BUFFER OVERFLOW EXPLOIT

Knowledge:
Buffer Overflow attacks are when the volume of data exceeds the allocated memory buffer for the specific application. This allows an attacker to point to their own malicious code to execute.

Application:
In this example, vulnserver, community debugger and a parrot machine is used to demonstrate the steps to exploit a buffer overflow.

Step 1: Fuzzing
The purpose of fuzzing is to send bytes and code to check for proper input sanitation. If this program crashes, it could determine if the program could potentially be vulnerable to a buffer overflow.

Once Vulnserver and community debugger are downloaded on the windows machine and the parrot machine is booted up, the fuzzing script can be made.

(The fuzzing script "**fuzzing.py**" can be found on my github under the Buffer Overflow Folder)

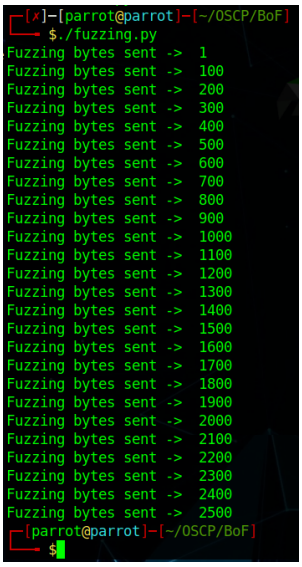


Figure 1: Fuzzing script output

In order to understand how fuzzing is done, we need to open the Immunity debugger on our Windows Machine. Then click File > Attach > Vulnserver.

To familiarize ourselves with the important registers, this link does a great job explaining the minimum.

This fuzzing script works by creating an array of increasing number of bytes and it keeps on sending larger and larger amounts until the program crashes

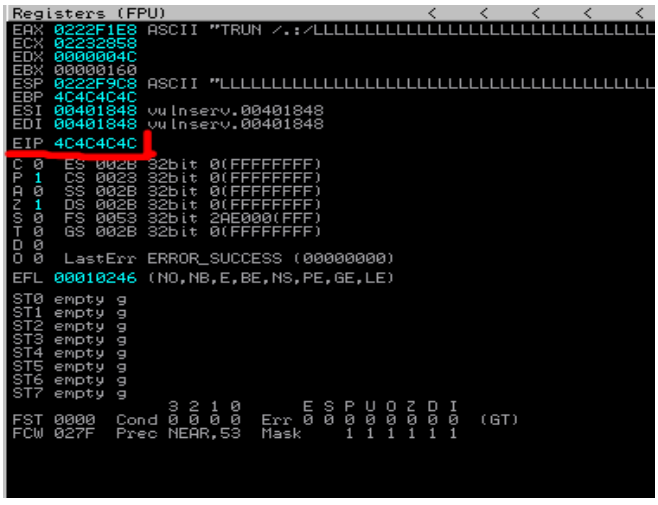


Figure 2. Immunity Debugger registers after fuzzing.py script

After running the fuzzing script, the EIP register is overwritten with "4C" which is the hex equivalent of the letter "L" which is what the server was fuzzed with.

Step 2: Finding the right amount of bytes

Upon fuzzing with 2500 bytes, the program crashed. In order to overwrite the EIP register, it is important to find out the exact amount of bytes to send and find where exactly the program crashes. To do that a string of unique non repeating characters will be used to fuzz.

To generate a string of unique non repeating characters, a tool from the metasploit framework known as **pattern create** will be used.

Figure 3: Unique pattern generated

This pattern will replace the L's in the Fuzzing script so that we can trace the EIP Value back and find where it is located in this string.

Upon replacing this in the fuzzing script, a new pattern.py script was created which can also be found on my github.

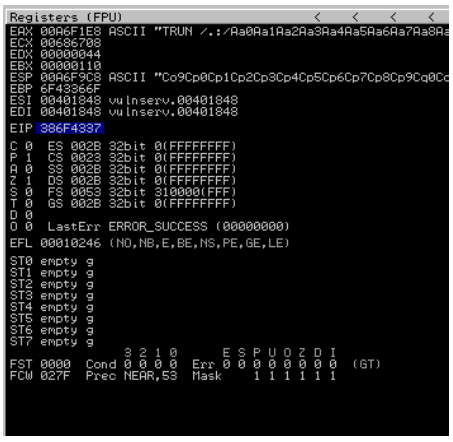


Figure 4: EIP Register values from unique pattern

This script was launched along with attaching and running vulnserver to Immunity Debugger. The values in the EIP Register read out to be 386F4337

Then another tool from the metasploit framework known as pattern offset was used to locate the exact match of "386F4337" as found on the EIP register. Pattern offset located the match at offset 2003

Figure 5: Finding offset

This means that in order to overwrite the EIP. We can continue to send in 2003 L's and the next 4 Letters we send after that should overwrite the EIP. In this case, using the same script, changed the unique string to "L" * 2003 + "A" * 4 and that gave the following result.

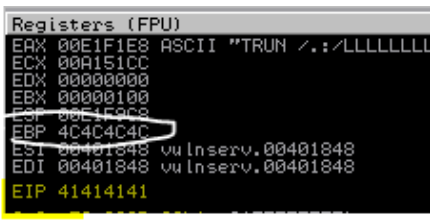


Figure 6: Successfully overwritten the EIP with 4 A's

Step 3: Finding bad characters

Bad characters are known to mess up exploits. So we have to make sure when we write our exploit we don't use any characters that could potentially mess up our exploit. The way to do this is to use a list of bad characters and make sure the program can accept and process all of them. One of the known bad characters is the Null Byte character " \x00".

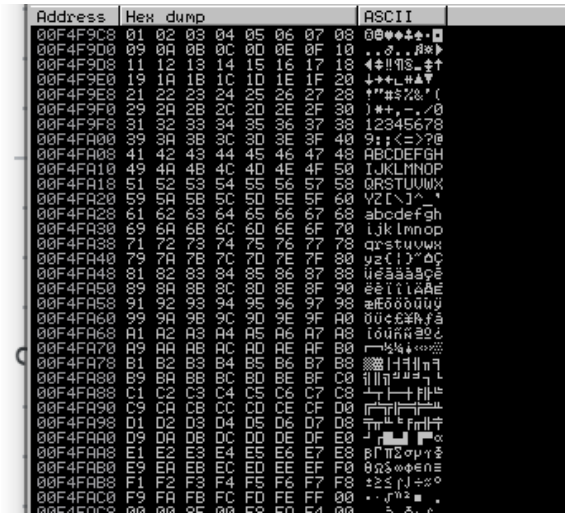
Using the same script, we can add the bad characters after the 4 A's and see if that triggers anything.

```
badChars = (
"\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"
)

uniqueString = "L" * 2003 + "A" * 4 + badChars
```

Figure 7: Bad characters list

After sending in the script, we can click open Immunity Debugger and click follow in dump on ESP. This will let us check if there are any bad characters. This script will be saved as badchars.py



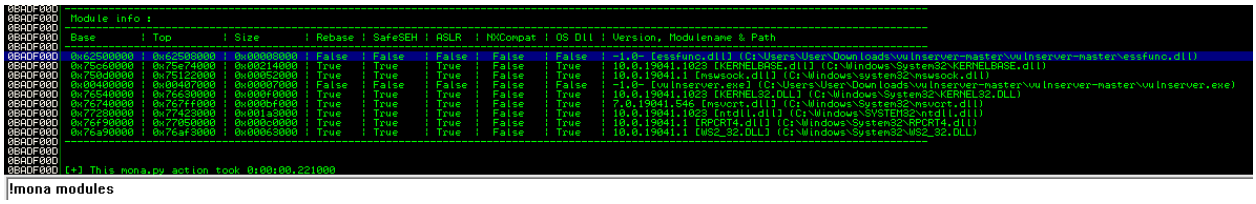
If there is any character missing or any character that is there that should not be there, that is a good indicator that there is a bad character in that position.

Step 5: Using mona module

For the next step, it is necessary to download the mona module. This could be found using this link.

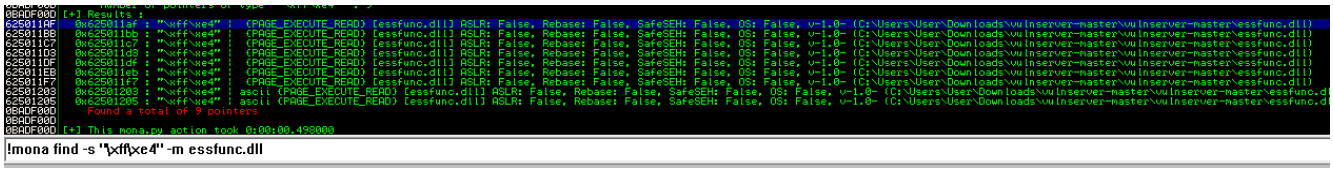
After it is downloaded and extracted, place it in C: > program files x86 > Immunity Inc > Immunity Debugger > PyCommands folder

After we attach vulnserver to the immunity debugger, we can type in !mona modules in the search bar at the bottom left of the screen to load the module.



Our goal here is to find preferably a dll file with mostly “false” so that there isn't memory protection and we can run malicious code. Since we already found one named essfunc.dll, our next step is to go back into our attacking machine and find the opt code equivalent (assembly to hex code).

There is a tool known as nasm shell that lets us do this. We are trying to find a return address for our EIP, to point to it. We can try using the nasm shell and type in JMP ESP, it provides the equivalent of FFE4.
WE can now go into the mona module and type in “mona find -s “\xff\xe4” -m essfunc.dll”.



This provides us with a bunch of return addresses that we can use.

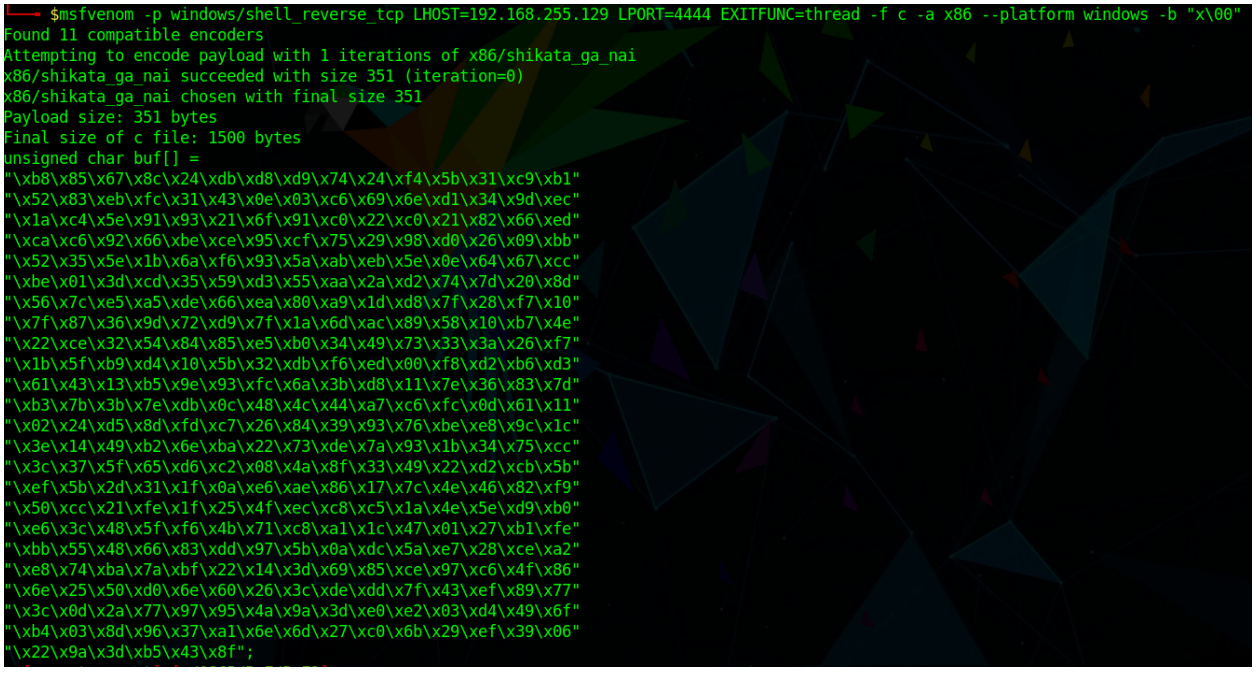
Now going back to our bad characters script, we can remove the bad characters and modify our code such that we have our random L’s for 2003 bytes and then instead of our 4 “A” ’s we have our address that we want our EIP to point towards (which was found using mona’s previous step). (make sure to type in return address in little indian notations (bytes reversed eg. abcdef would “\xef\xcd\xab”)

This script is saved as monaReturn.pyc

Now our script executes such that when it overwrites the EIP, its overwritten with a memory address that jumps to the ESP

Step 6: Malicious Code

Now, we can use msfvenom to create a reverse tcp shell payload.
-p is the payload flag, in the case we specified windows and reverse shell
L HOST/PORT is the attacker (our) ip/port to listen on
EXITFUNC=thread provides stability so we don't disconnect randomly
-f is for programming languages. We generated in c because it makes it easier to add to our code.
-a is architecture which is x86
-b is the flag for bad characters, in this case it is the null byte character



Now we can attach this to our code after the pointer for EIP and we can also add a little bit of padding before this code to make sure it runs smoothly. We can add a NOP which is a no operation character. This character is the op code of 0x90. So we can add approximately 32 bytes of "\x90" before our payload generated by msfvenom and after the EIP pointer. This script is saved as bufferExploit.py

After running this script and starting a netcat listener on port 4444, we should gain a shell