Lab Tecniche di Protezione del Software

Massimo Perego

Contents

1 Introduzione

Debugger: Permette di analizzare un programma riga per riga, impostando break point ed analizzando la memoria durante l'esecuzione.

Alcuni comandi:

- `run r`: esegue il programma
- `step`: istruzione successiva (entrando nelle funzioni)
- `next`: istruzione successiva, ma le funzioni sono trattate come istruzione singola
- `info registers`: per vedere stato dei registri
- `x/nfu`: per esaminare la memoria, dove `n` è il numero di unità da stampare, `f` il formato, `u` il tipo di unità
- `b addr`: imposta un breakpoint all'indirizzo `addr`, dove l'indirizzo può essere relativo, ad esempio `*foo+123` mette un breakpoint all'indirizzo `+123` della funzione `foo`
- `disassemble foo`: mostra le istruzioni della funzione `foo`
- `set what=val`: setta `what` al valore `val`

Ghidra: Decompiler prende l'output di un disassembler e cerca di ricostruire istruzioni di più alto livello in base a pattern noti. Dall'assembly cerca di ricostruire un codice leggibile (forse).

Calling convention 64 bit: Su architettura 64 bit, quando viene chiamata una funzione i valori vengono caricati all'interno dei registri, nell'ordine

`rdi, rsi, rdx, rcx, r8, r9`

Se ne sono presenti di più allora vengono caricati sullo stack.

2 Buffer Overflow

Per gli esercizi di pwncollege, in generale abbiamo a disposizione un buffer e dobbiamo far eseguire una funzione `win()`, in qualche modo (dipende dall'esercizio, ma generalmente sovrascrivendo l'indirizzo di ritorno della funzione `challenge()` con l'indirizzo di `win()`), al fine di ottenere una flag.

Per capire la dimensione del buffer:

- Ghidra: la notazione di ghidra fa riferimento all'inizio dello stack, se il buffer è chiamato `local_a8`, l'inizio dello stack (e di conseguenza l'inizio dell'indirizzo di ritorno della funzione) si trova `0xa8` bit prima del buffer (i.e., devo sovrascrivere 168 byte, 160 di buffer e cose varie poi 8 di indirizzo di ritorno)
- gdb: guardo l'indirizzo del buffer, cercando dove viene popolato rispetto al RBP (poi ci sono 8 byte di Saved Frame Pointer SFP, poi il return address). Non so come si faccia tbh, dovrei scoprirlo.
- Stringhe cicliche: scrivo n indirizzi di memoria diversi (valori casuali) e guardo il core dump: lo stato dei registri mostra il valore dell'instruction pointer, ovvero quale dei n diversi indirizzi ha fatto crashare il programma, se è il 10^o indirizzo allora la distanza è $10 \cdot 8$, potremmo farlo a mano, oppure usare pwntools:
  - `elf = ELF('')` per definire l'eseguibile
  - `io = elf.process(setuid=False)` per dire di non usare il `suid` (altrimenti il programma non fornisce il core dump)
  - `io.sendline(cyclic(512, n=8))` genera la stringa ciclica, 512 byte in blocchi 8
  - `io.wait()` aspetta il crash
  - `print(cyclic_find(io.corefile.fault_addr, n=8))`: dall'indirizzo da cui è crashato il programma estrapola la distanza dall'inizio dello stack, usando indirizzi da 8 byte

Quest'ultimo meccanismo non funziona con canary o stack protection. Checksec sul binario (da pwntools) restituisce le protezioni sull'eseguibile.

Se all'interno della funzione `win()` c'è il check di un parametro all'inizio della funzione, posso:

- usare dei "gadget" (to be continued)
- andare ad un istruzione dopo il controllo, al posto dell'indirizzo iniziale della funzione `win()` scrivo l'indirizzo di una istruzione all'interno della funzione dopo il check