Lab Tecniche di Protezione del Software

Massimo Perego

Indice

1 Introduzione **2**

2 Buffer Overflow **3**
2.1 Randomizzazione degli indirizzi ASLR 4
2.2 Canary 4

3 Shellcode Injection **5**

4 Return Oriented Programming ROP **6**
4.1 Defeat Dynamic Linking 7

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 (forse) leggibile.

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 si ha a disposizione un buffer e bisogna 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 il punto subito dopo l'indirizzo di ritorno della funzione) si trova `0xa8` bit prima del buffer (i.e., devo sovrascrivere 176 byte, 168 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

```python
# Per definire l'eseguibile
elf = ELF(path)
# Con suid attivo non fornisce core dump
io = elf.process(setuid=False)
# Genera stringa ciclica, 512 byte in blocchi da 8
io.sendline(cyclic(512, 8))
# Aspetta il crash
io.wait()
# Estrapola l'offset da inizio stack,
#    in base all'indirizzo da cui è crashato il programma,
#    con indirizzi da 8 byte
print(cyclic_find(io.corefile.fault_addr, n=8))
```

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, ma dopo il check

2.1 Randomizzazione degli indirizzi ASLR

Generalmente attivo di default, l’ASLR **randomizza la posizione del programma in memoria**, di conseguenza rendendo casuali return address e indirizzo del buffer.

Ci possono essere diverse implementazioni, con le relative difficoltà, ma in generale è implementato a livello **kernel**: ogni volta che un programma viene lanciato gli viene assegnata una pagina diversa; gli indirizzi vengono calcolati a runtime tramite offset.

Rimane noto l’offset all’interno del programma, ovvero le ultime 3 cifre in esadecimale dell’indirizzo (12 bit, le pagine sono allineate a `0x1000`). Si può sfruttare questa cosa sovrascrivendo solo gli ultimi 2 byte dell’indirizzo di ritorno e presupponendo che la funzione da eseguire non sia “troppo lontana” dal resto. In questo modo sarà casuale solo l’ultima cifra in esadecimale dovuta al paging, basta ripetere l’esecuzione “qualche volta” (probabilità 1/16) per indovinare anche l’ultimo carattere.

2.2 Canary

Se presente una canary, dobbiamo leakarla per poi sovrascriverla. Generalmente gli esercizi hanno un `%s` e una “backdoor” che fa ripartire la challenge.

Facendo overflow del buffer fino alla canary (+1 byte per sovrascrivere il primo `0x00` della canary, sempre presente) e non inserendo un terminatore di stringa il `%s` stamperà il valore della canary, inseribile all’esecuzione successiva della challenge.

3 Shellcode Injection

L'obiettivo è iniettare e far eseguire al programma una shellcode (o comunque codice arbitrario). Lo schema più semplice possibile è:

- ho un buffer in cui posso fare overflow
- scrivo all'interno del buffer una shellcode/codice
- sovrascrivo il return address con l'indirizzo del buffer

Possibili problemi con l'esecuzione della shellcode:

- **stack non eseguibile**: protezione solitamente disabilitata per gli esercizi
- **shellcode più piccolo del buffer**: devo calcolare quanto padding serve per terminare il buffer ed arrivare al return address
- **non abbastanza spazio per lo shellcode**: più opzioni
  - dividere lo shellcode in due parti, la prima avrà un'istruzione "finale" che fa saltare 16 byte in avanti per "skippare" `SFP` e `return_address`, per poi continuare con il codice dello shellcode. Per saltare avanti
    - `jmp 0x10` in avanti
    - `mv rax, rip; add rax, 0x10; mov rip, rax`
  - anziché dividere lo shellcode, si inizia con del padding fino a sovrascrivere l'indirizzo di ritorno, per poi scrivere lo shellcode completo al di sotto (o sopra, punti di vista; di solito sopra lo stack frame corrente c'è tutto lo spazio necessario); padding di `dim_buffer+8` byte di `SFP + 8` byte `return_address`

Per costruire lo shellcode, msfvenom, a mano, exploitdb, ma per l'esame pwntools permette di crearne con shellcraft. Esempio pwntools:

```
1 context.arch = 'amd64'  # imposta l'architettura
2 shellcode = shellcraft.sh()
3 print(asm(shellcode))
```

Esistono diversi comandi per generare diversi tipi di payload, `visibili sulla documentazione`.

4 Return Oriented Programming ROP

Voglio creare una catena di gadget per eseguire codice arbitrario. La sfida è capire *cosa fare* con i gadget a disposizione. In generale, nelle challenge, la catena va creata a partire da:

```
payload = b'A'*buf_offset + gadget1 + param1 + gadget2 +
                        param2 + …
```

Fino ad avere tutti i gadget e parametri sullo stack.

Trovare i gadget: Esistono tool come ropper, ropgadgets, …(cerca). Altrimenti si può fare a mano (psycho behaviour).

Esempio

```
ropper -f file
```

Così li sputa tutti, per specificare quale gadget: `--search "pop rdi"`.

Pwntools contiene un modulo ROP

```
rop = ROP(binario)
```

e permette di trovare i gadget con (esempio come sopra):

```
rop.rdi.address
```

Potrebbe trovarne più di uno, in tal caso è da specificare quale usare.

Pwntools permette anche di creare catene di rop: dopo aver caricato il modulo rop sopra, posso chiamare una funzione con dei parametri specifici:

```
rop.win_stage_4(4)
```

per passare il parametro `4` alla funzione `win_stage_4()`.

Per vedere la catena creata

```
rop.dump()
```

Per mandarla

```
rop.chain()
```

Se non ho una `win()` da chiamare:

- inietto una shellcode sullo stack: ma potrebbe essere non eseguibile

- uso le syscall: potrei avere gadget per fare chiamate di sistema, ce ne sono per tutti i gusti

4.1 Defeat Dynamic Linking

Il dynamic linking è una tecnica utilizzata per permettere di eseguire codice che non fa direttamente parte dell'eseguibile (es: tutte le funzioni di `libc`). Le librerie esterne vengono caricate runtime al posto di essere incorporate nel binario.

Ogni libreria viene mappata in memoria con il proprio spazio di indirizzi (e relativi metodi di difesa attivi). Per ciascun simbolo (funzione) usato dall'eseguibile o da altre librerie, alla prima chiamata:

- entra nella **PLT (Procedure Linkage Table)**, la quale conterrà uno stub che porta al resolver
- il resolver riceve la richiesta e trova la funzione nella shared library corretta
- l'indirizzo reale della funzione viene scritto nella **GOT (Global Offset Table)**

Per le chiamate successive, la GOT conterrà l'indirizzo definitivo, quindi l'indirizzo della PLT passa direttamente alla funzione senza passare dal resolver. La GOT è essenzialmente composta da un array di puntatori a funzioni.

Leak libc: La libc, quando dinamically linked, avrà sempre ASLR e tutte le altre protezioni attive, quindi dobbiamo scoprirne l'indirizzo a runtime. Per farlo vogliamo ottenere l'indirizzo reale di una delle funzioni di libc, ovvero stampare l'indirizzo di una qualsiasi funzione. Un modo semplice è fare `puts` di `puts@got` tramite `puts@plt`.

Una volta noto l'indirizzo di una funzione e l'offset di tale funzione dall'inizio della libc si può calcolare la base.