Capture the Flag Manual

Alessio Gjergji
Nicolò Piccoli
Davide Rossignolo
Author 4
Author 5

Indice

Capitolo 1

Software Security OlyCyber

1.1 Introduzione

I numeri interi, in questo caso int32, possono essere rappresentati in due maniera, Big-Endian(cifra significativa a sinistra) e Little-Endian(cifra significativa a destra).

1.2 Assembly x86_64

Ogni istruzione assembly ha degli operandi(registri) e un'operazione.
La notazione per architetture intel è del tipo <op><destinazione><sorgente>, i registri hanno la seguente struttura: AH,AL -¿ 8 bit, AX -¿ 16 bit, EAX -¿ 32 bit, RAX -¿ 64 bit.
Tra le operazioni di base che troviamo ci sono:
MOV <dst><src>
PUSH <src>or POP<src>
ADD or SUB <dst><src>
CALL <pc>or RET
Tra i salti condizionali invece abbiamo: CMP <opn1><opn2>-¿ confronta due valori e imposta delle flag
J <condizione><pc>-¿ salta a PC se le flag soddisfano la condizione

1.3 Buffer Overflow

Consideriamo di aver dichiarato in c un'istruzione del tipo "char name[100];", cosa succede se l'utente ha un nome che supera i 100 caratteri?. Può succedere che con scanf() andiamo a scrivere caratteri oltre la fine di name, andando a sovrascrivere la memoria che lo segue, generando quindi un buffer overflow che va a corrompere la memoria, ossia scriviamo dati in posizioni che il programmatore non aveva previsto fossero modificate.
Se corrompiamo abbastanza bene la memoria possiamo addirittura prendere il controllo del processo(arbitrary code execution).

1.3.1 Accessi out of bound

Se abbiamo una struct che dichiara un int a[2] e un int b[3], nel momento in cui scrivo in a[2] o in b[0] non cambia niente, sono equivalenti.

1.4 Reverse Engineering

1.4.1 Binary

Gli eseguibili nativi sono file che contengono codice macchina eseguibile dal processore, contengono anche informazioni usate dal sistema operativo per caricarlo in memoria.
Il formato **elf** è flessibile e serve a rappresentare i file binari, in linux è usato per rappresentare eseguibili e librerie condivise, ad alto livello invece risulta come un insieme di strutture che descrivono come caricare in memoria i dati salvati nello stessso file.
Per analizzare file ELF abbiamo alcuni strumenti tra cui:

- **readelf**: stampa le informazioni contenute nei file elf
- textbfnm: stampa tutti i simboli contenuti nel file elf
- textbfobjdump: stampa le informazioni contenute nel file oggetto, è più specifico rispetto a readelf
- textbflld: stampa gli oggetti condivisi necessari all'esecuzione del programma
- textbflief: libreria python per analizzare e modificare file elf.

1.4.2 Memoria a basso livello

Se vogliamo considerare un'astrazione della memoria troviamo vari livelli:

- **dati tipati**: byte interpretati
- linguaggi di programmazione
- textbfmemoria virtuale: sequenza di byte indirizzabili, spazio indipendente per processo, solo alcune aree mappate(con la fisica)
- sistema operativo
- textbfmemoria fisica: sequenza di byte indirizzabili

1.4.3 Spazio virtuale linux user space

- Text
- Data -¿ dati globali inizializzati
- BSS -¿ dati globali azzerati
- Heap -¿ allocazioni dinamiche
- Librerie -¿ binari librerie dinamiche

- Stack -¿ var locali, record di attivazione

1.4.4 metodologie di base di reverse

Tool per analisi statica: Ghidra, IDA, Binary Ninja, Radare2 -¿ analizzano qualsiasi binario in input

- JADX: reverse bytecode java
- dnSpy, ILSpy: reverse bytecode .NET
- uncompyle6/unpyc: reverse bytecode python
- Iuadec: reverse bytecode LUA.

Tool per analisi dinamica:

- gdb: da usare con GEF o PWNDBG
- radare2 integra features comode per il reversing
- rr timeless debugging
- frida inietta codice js in qualsiasi punto del programma
- ida debugger debugger gui disponibile free.

Capitolo 2

Sw Sec Lab Univr

2.1 Elf File

È il file standard per gli eseguibili Unix, Elf sta per Executable and Linkable Format, è essenzialmente un file binario che continene varie informazioni tra cui:

- **header**: descrive il contenuto del file per l'esecuzione
- **Pht(program header table)**: da informazioni su come si crea l'immagine del processo
- **Sequenza di sezioni**: contengono ciò che serve per il linking
- **Section header table**: descrizione delle sezioni precedenti

Per windows l'equivalente è PE, per mac Mach-O

2.2 Memoria

2.2.1 Storage Size

WORD = 2 BYTES, DWORD = 4 BYTES, QUADWORD = 8 BYTES. La cifra più significativa è a sinistra, quella meno è a destra.

2.2.2 Hexadecimal

Si usa come forma compatta dei numeri binari, si va da 0 a 9 e da A a F.
Ogni digit in hex rappresenta 4 bits, quindi 2 digit un bytes, in C sono scritti come 0X FA 1D .... .

2.2.3 Registri x86_32

Ha registri general purpose a 32 bit.
Sono strutturati nella seguente maniera: AH,AL 8 bit ciascuno, AX 16 bit, EAX 32 bit.
EAX è storicamente usato come accumulatore, ECX come counter, inoltre ci sono ESP(stack

pointer) e EBP(base pointer).
Se passiamo a x86_64 si aggiungono i registri a 64 bit che si nominano del tipo: RAX.

2.2.4 x86 Memory managment

La memoria è semplicemente una sequenza di bytes, ognuno con un indirizzo unico, i compilatori potrebbero introdurre padding per cambiare l'ordine dei dati per ottimizzare, per questo motivo ci torna comodo vederla come una matrice di tot righe con n bytes(n=processor word), se sono a 64 bit n=8.
Un indirizzo è una locazione in memoria, un **puntatore** è un oggetto mantiene un indirizzo.
I bytes possono essere ordinati in memoria in due maniere:

- **BigEndian**: il bytes meno significativo ha indirizzo più alto
- **LittleEndian**: il bytes meno significativo ha indirizzo più basso

2.3 intel x86 instruction set

La notazione per architetture intel è del tipo <op><destinazione><sorgente>
Vediamo ora alcune istruzioni:

- mov: muove dati da un src a un registro dst
- push: mette l'operando sullo stack
- pop: rimuove l'operando dallo stack
- lea: carica ciò che c'è all'indirizzo [] in dst
- add: fa la somma e salva in op1
- sub: fa la differenza e salva in op1
- inc: incrementa di 1
- and/or/xor: fa l'operazione e salva in op1
- jump: salto non condizionale a label
- cmp: compara contenuto di op1 con op2
- je/jne/jz/jg/jge/jl/jle: in base a cmp fa salto condionale
- call: usato per chiamare funzioni
- ret: implementa il ritorno da funzione

Per quanto riguarda lo stack, questo è formato da degli stack frames, uno per ogni funzione chiamata, **stack pointer** punta all'ultimo elemento nello stack(il primo inserito).
Per quanto riguarda ogni singolo frame, abbiamo che **base pointer(frame pointer)** punta l'indirizzo dal quale partono le variabili locali.

2.4 Debugging con GDB

Posso usare gdb in varie maniere:

- gdb >program <
- gdb >program <>pid <
- gdb -p >pid <

2.4.1 Comandi di GDB

- run: fa partire il programma in gdb
- set args: imposta gli argomenti del programma
- show args: mostra gli argomenti del programma
- help: comandi disponibili
- break: mette breakpoint alla prossima istruzione
- break location: mette un breakpoint alla locazione location(preceduta da *)
- break [location] if >condition <: mette un breakpoint data una condizione
- continue(c): va al prossimo breakpoint(se esiste)
- nexti(ni): esegue solo un'istruzione
- frame [>selection <]: stampa una descrizione dello stack frame selezionato
- info frame [>selection <]: stampa una descrizione più informativa rispetto a frame
- disas: disassembla una funzione
- print(p): stampa il contenuto di un indirizzo o registro(*0x092a3e or $eax)
- x: non ho ben capito
- call: chiama una funzione
- set: modifica il valore di una locazione di memoria o di un registro

Capitolo 3

Comandi terminale

3.1 Introduzione

Durante una ctf potremmo trovarci di fronte ad alcune challenge in cui è necessario l'utilizzo di vari comandi della shell per recuperare la flag richiesta e quindi passare alla challenge successiva. Di seguito vedremo alcuni comandi della shell per sistemi UNIX o macOS che possono tornare utili.

3.2 Comandi di base

Se si conosce un comando ma non si sa come utilizzarlo è bene consultare il manuale scrivendo sul terminale **man** ⟨*command*⟩. Se non è presente la pagina del manuale provare a specificare il flag –**help**.

3.3 Operazioni sulle directory

3.3.1 cd

Per navigare attraverso il filesystem utilizziamo il comando **cd dir**.

- **cd esempio** (ci spostiamo nella cartella *esempio*)
- **cd Dekstop/esempio** (ci spostiamo nella cartella *esempio* identificata dal suo path)
- **cd ..** (ci permette di spostarci nella cartella superiore)
- **cd** ∼ (ci spostiamo nella home directory)
- **cd /** (ci spostiamo nella root directory)

Per creare una cartella utilizziamo il comando **mk dir**, se invece vogliamo vedere la cartella corrente utlizziamo il comando **pwd**.

3.3.2 ls

Se vogliamo vedere i file all'interno di una cartella utilizziamo il comando **ls**.

- **ls** (mostra i file nella cartella corrente)
- **ls Desktop/esempio** (mostra i file all'interno della cartella *esempio* identificata dal path)

Tra le opzioni del comando **ls** possiamo trovare:

- **-a** (mostra tutti i file, inclusi quelli nascosti)
- **-r** (inverte l'ordine della lista)
- **-t** (ordina in base all'ultimo modificato)
- **-S** (ordina per dimensione del file)

3.4 Operazioni sui file

3.4.1 touch

Per creare un file utilizzare il comando **touch file**.

3.4.2 cat

Il comando **cat** permette di concatenare file e stampare il loro contenuto sullo standard output.

- **cat file** (stampa il contenuto del file, se vengono specificati più file li concatena e stampa il contenuto, e.g. cat file file2)
- **cat < -file** (permette di stampare il contenuto di un file con il nome che inizia con un dash)
- **cat** "$nomefileconspazi$" (permette di stampare il contenuto di un file che contiene spazi nel nome)
- **cat .file** (stampa il contenuto del file nascosto)

3.4.3 cp, mv, rm, file

- **cp file file2** (copia file in file2)
- **mv file file2** (rinomino file in file2)
- **rm file** (elimino file)
  - **rm -r** (rimuovo le directory e i loro contenuti)
  - **rm -d** (rimuovo direcotry vuote)
- **file file1** (ritorna il tipo di file1)

3.4.4 head, tail

- **head file1** (ritorna le prime 10 linee di file1)
- **tail file1** (ritorna le ultime 10 linee di file1)

3.4.5 strings

Il comando **strings** stampa una sequenza di stringhe leggibili all'interno di un file.

- **strings file**
  - con il flag **-n number-of-lines** (specifichiamo la lunghezza minima delle stringhe)
  - con il flag **-e encoding** (specifichiamo la codifica)
  - con il flag **-w** (includiamo gli spazi bianchi)
  - con il flag **-s** (il separatore per l'output)

3.5 Ricerca

3.5.1 sort, unique

Il comando **sort file** permette di ordinare le linee all'interno di un file, il comando **unique -u** permette di mostrare le linee uniche non duplicate, questi due comandi possono essere comodi da usare in combinazione attraverso l'utilizzo di una pipe: **sort nomefile — unique -u**.

3.5.2 grep

Il comando **grep pattern files** permette di cercare un determinato pattern in ogni file, i pattern andrebbero specificati sempre compresi tra doppi apici.

- **grep -i** (ricerca case-insensitive)
- **grep -r** (ricerca ricorsiva)
- **grep -v** (ricerca invertita)
- **grep -o** (mostra solo la parte di file che ha matchato il pattern)

3.5.3 find

Il comando **find** permette di cercare dei file all'interno del filesystem.

- **find /percorso -name** "$filename$" (ricerca per nome)
- **find /percorso -name** "$*.txt$" (ricerca per estensione)
- **find /percorso -type f -size +1M** (ricerca per dimensione)
- **find /percorso -user utente -group gruppo** (ricerca per proprietario e gruppo)
- **find /percorso -mtime -7** (ricerca per data di modifica)

3.6 Connessioni ssh o tcp

Connessione ad una risorsa in ssh:

- **ssh -p numero-porta utente@indirizzo-del-server**

Connessione tramite tcp:

- **nc host port**

3.7 Gestione dei processi

- **ps** (mostra uno snapshot dei processi)
- **top** (mostra i processi real-time)
-
- **kill pid** (termina un processo con il pid=pid)
- **pkill name** (termina un processo col nome=name)
- **killall name** (termina tutti i processi con il nome che inizia per name)