BÁO CÁO THỰC HÀNH

**Môn học: Lập trình an toàn & Khai thác lỗ hổng phần mềm**

**Lab 6: Bài tập tổng hợp**

*GVHD: Nguyễn Hữu Quyền*

1. **THÔNG TIN CHUNG:**

*(Liệt kê tất cả các thành viên trong nhóm)*

Lớp: NT521.O12.ATCL - Nhóm 3

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **MSSV** | **Email** |
| 1 | Bùi Hoàng Trúc Anh | 21521817 | [21521817@gm.uit.edu.vn](mailto:21521817@gm.uit.edu.vn) |
| 2 | Lê Hoàng Oanh | 21521253 | 21521253@gm.uit.edu.vn |
| 3 | Nguyễn Ngọc Trà My | 21520353 | [21520353@gm.uit.edu.vn](mailto:21520353@gm.uit.edu.vn) |
| 4 | Huỳnh Minh Tân Tiến | 21521520 | 21521520@gm.uit.edu.vn |

1. **NỘI DUNG THỰC HIỆN:**

|  |  |  |
| --- | --- | --- |
| **STT** | **Công việc** | **Kết quả tự đánh giá** |
| 1 | **Stack Architect** | 100% |
| 2 | **Shellcode** | 100% |
| 3 | **Autofmt** | 100% |
| 4 | **Ropchain** | 100% |

**Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.**

BÁO CÁO CHI TIẾT

1. **Stack Architect**

Đầu tiên, ta chạy gdb đối với file stack\_architect.

Khi này, ta có thể xem được checksec của file. Trong checksec có bật NX, vì vậy sẽ

không thể truyền shellcode vào chạy được.

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, màn hình

Description automatically generated

Tìm địa chỉ của các hàm: func1, func2 và win trong file compile để thực hiện việc exploit.

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, số

Description automatically generated

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, số

Description automatically generated

Ảnh có chứa văn bản, phần mềm, ảnh chụp màn hình, Biểu tượng máy tính

Description automatically generated

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm

Description automatically generated

Cần tìm tiếp địa chỉ của pop;ret. Vì vậy ta sẽ cần dùng đến ROPgadget với command bên dưới.

Lệnh sau sử dụng công cụ `ROPgadget` để tìm kiếm các chuỗi lệnh (gadget) trong một tệp thực thi (binary) được gọi là "stack\_architect". Cụ thể, nó tìm kiếm các gadget có dạng "pop" hoặc "ret" trong mã máy (assembly). Đây là giải thích từng phần của lệnh:

* ROPgadget: Đây là công cụ được thiết kế để tìm kiếm và phân tích các gadget ROP (Return-Oriented Programming) trong các tệp thực thi.
* “binary stack\_architect”: Chỉ định tệp thực thi cụ thể mà ROPgadget sẽ phân tích. Trong trường hợp này, tệp thực thi có tên "stack\_architect".
* ”--only 'pop|ret'”: Thực hiện lọc các gadget dựa trên các chuỗi lệnh chỉ định. Trong trường này, chỉ có các gadget chứa các lệnh "pop" hoặc "ret" sẽ được hiển thị. Điều này hữu ích khi bạn chỉ quan tâm đến các gadget có thể sử dụng để xây dựng các chuỗi tấn công ROP.

Kết quả của lệnh này sẽ là danh sách các gadget trong tệp "stack\_architect" mà ROPgadget xác định là có thể sử dụng trong một cuộc tấn công ROP, đặc biệt là các gadget chứa lệnh "pop" hoặc "ret".

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Description automatically generated

Sau khi có đủ các địa chỉ cần thiết, ta tạo file exploit để chuẩn bị và truyền payload đến máy vul.

Ảnh có chứa văn bản, đồ điện tử, ảnh chụp màn hình, máy tính

Description automatically generated

Như vậy stack sau khi bị ghi đè sẽ có nội dung như sau

|  |
| --- |
| Địa chỉ win |
| Địa chỉ func2 |
| Địa chỉ func2 |
| 0x20010508 |
| Địa chỉ popret |
| Địa chỉ func1 |
| Địa chỉ func1 |
| \x00  …  \x00 |
| ‘I’m sorry, don’t leave me, I want you here with me ~~’ |
| AAAA |

Sau khi chạy file exploit, ta sẽ lấy được flag.

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Description automatically generated

1. **Shellcode**

Câu lệnh file shellcode được sử dụng để xác định loại file của "shellcode". Kết quả cho thấy đây là một tập tin thực thi ELF 64-bit, được động liên kết và không được loại bỏ thông tin định danh (not stripped).

Câu lệnh checksec --file=shellcode được sử dụng để kiểm tra các cài đặt bảo mật của file shellcode. Kết quả cho thấy RELRO không đầy đủ (Full RELRO), không có canary (No canary found), NX (No eXecute) bị tắt, PIE (Position Independent Executable) đã được kích hoạt.





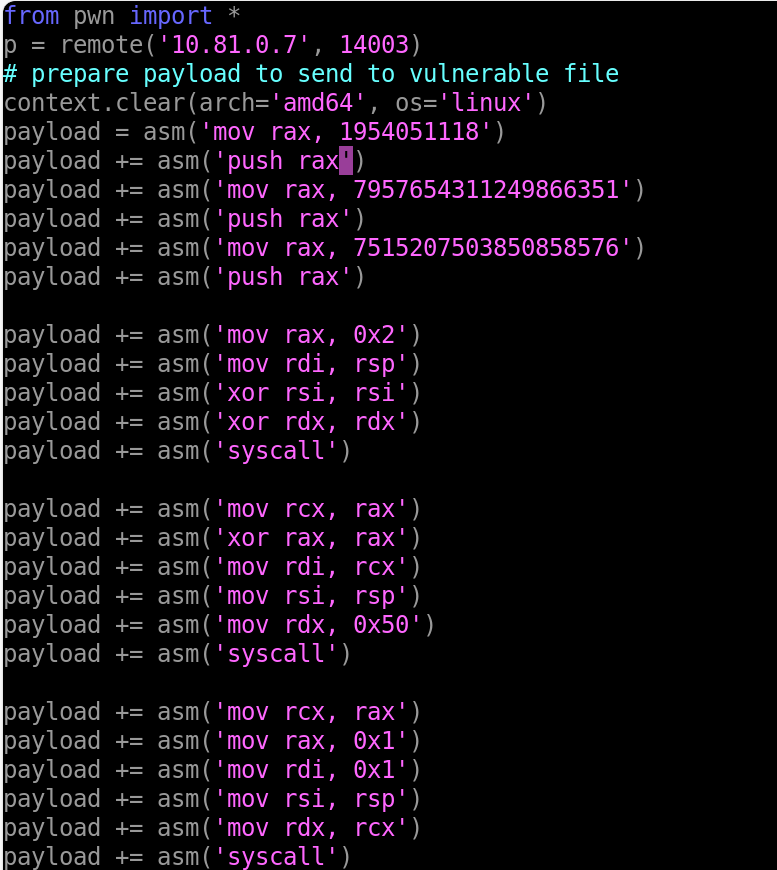
A screen shot of a computer

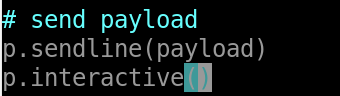
Description automatically generated

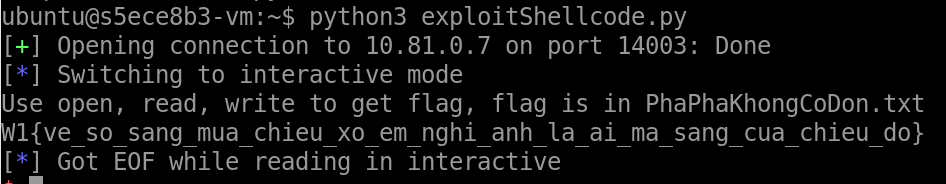
Một đoạn mã Python được sử dụng để tạo ra một payload chứa các hàm hệ thống như open, read, write để đọc nội dung của file "PhaPhaKhongCoDon.txt" và trả về flag.

File python để exploit shellcode

* Kết nối đến máy chủ từ xa:
* p = remote('10.81.0.7', 14003): Tạo một kết nối đến máy chủ có địa chỉ IP là '10.81.0.7' và cổng là 14003 sử dụng thư viện pwn.
* Giai đoạn chuẩn bị payload:
* context.clear(arch='amd64', os='linux'): Xác định kiến trúc là 64-bit và hệ điều hành là Linux để chuẩn bị mã shellcode.
* payload = asm('mov rax, 1954051118'): Di chuyển giá trị 1954051118 vào thanh ghi rax.
* payload += asm('push rax'): Thực hiện việc đẩy giá trị của thanh ghi rax vào đỉnh của ngăn xếp.
* Tương tự, các bước tiếp theo là thực hiện đẩy các giá trị khác vào ngăn xếp.
* Gọi các hàm hệ thống sử dụng syscall:
* payload += asm('mov rax, 0x2'): Đặt giá trị 2 vào thanh ghi rax, đây là mã syscall cho open.
* payload += asm('mov rdi, rsp'): Đặt con trỏ đến chuỗi cấu trúc đường dẫn (file name) trong thanh ghi rdi.
* payload += asm('xor rsi, rsi') và payload += asm('xor rdx, rdx'): Đặt rsi và rdx là 0, đánh dấu việc không có quyền truy cập và không có cờ mở rộng.
* payload += asm('syscall'): Gọi hệ thống open để mở file.
* Tiếp theo, payload tiếp tục thực hiện read để đọc nội dung của file đã mở và sau đó write để hiển thị nội dung ra màn hình.
* Chuyển sang chế độ tương tác và hiển thị kết quả:
* p.interactive(): Chuyển sang chế độ tương tác, cho phép người dùng tương tác trực tiếp với kết nối.
* Trong trường hợp này, đầu ra của chương trình mục tiêu (bao gồm cả flag) được hiển thị trên màn hình.



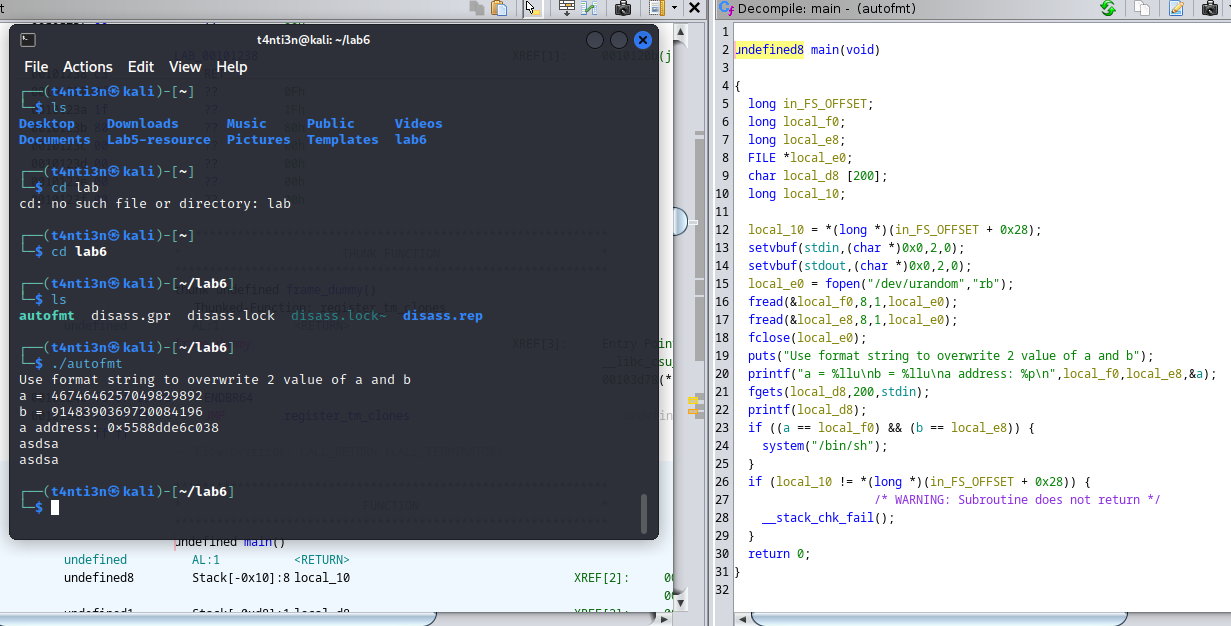




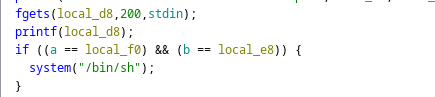
Vậy flag là: W1{ve\_so\_sang\_mua\_chieu\_xo\_em\_nghi\_anh\_la\_ai\_ma\_sang\_cua\_chieu\_do}

1. **Autofmt**

Flag: W1{do\_cac\_ban\_tren\_the\_gian\_nay\_khoang\_cach\_nao\_la\_xa\_nhat}  
Overview source file autofmt

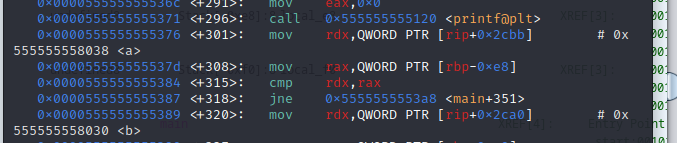


Đoạn codition code có gọi hàm system

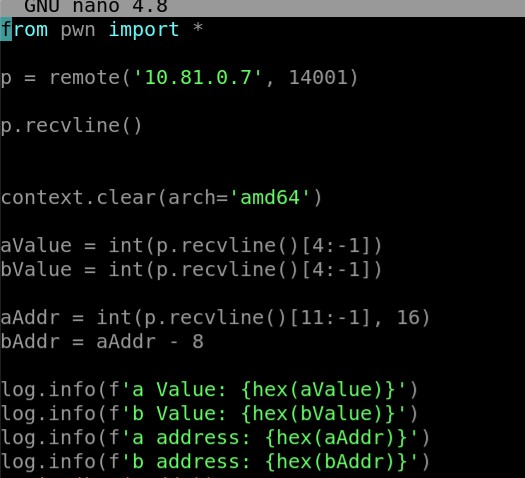


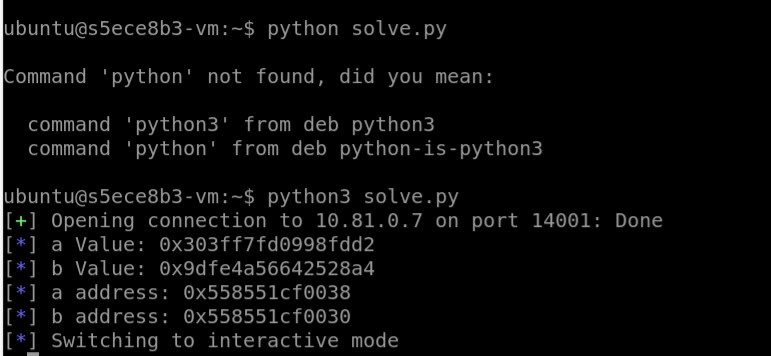
Mục tiêu phải overwrite được giá trị của a và b để các giá trị nhất định để có thể gọi được system bin/sh

Gdp disass main



Địa chỉ của a cách b là 0x8 byte, thử lại trên máy vul để kiểm chứng





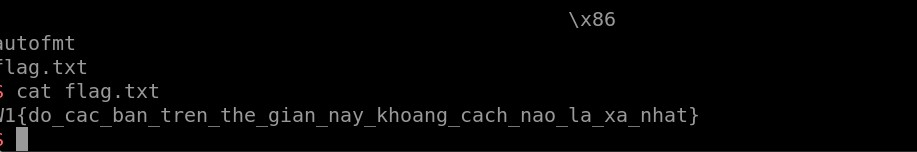
Mục tiêu ghi lại giá trị ở a address và b address thành giá trị của a value và b value



hàm fmtstr\_payload() của thư viện pwntools để tạo một payload. Payload này sẽ được gửi tới máy chủ và sẽ được sử dụng để sửa đổi giá trị của các biến a và b. Hàm fmtstr\_payload() nhận ba đối số:

* Length: chiều dài của payload.
* Writes: một dict chứa các cặp khóa-giá trị, trong đó khóa là địa chỉ của biến và giá trị là giá trị của biến.
* Write\_size: kích thước của mỗi lần ghi. Trong trường hợp này, kích thước của mỗi lần ghi là 2 byte.

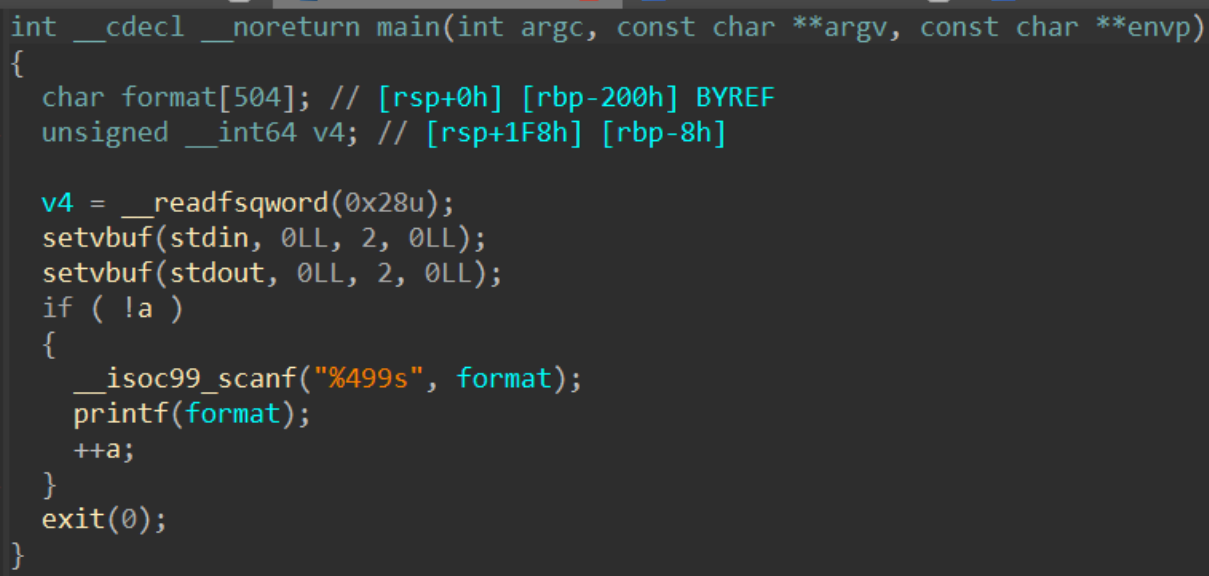
Kết quả



1. **Ropchain**

W1{biet\_yeu\_em\_la\_lam\_day\_nhung\_tinh\_cam\_nay\_day\_lam}

Overview source file



* Chương trình kiểm tra giá trị của biến a, nếu a = 0 sẽ đọc input từ người

dùng và lưu vào biến format. Chương trình sẽ gọi printf(format) → Lỗi Format string. Tăng biến a thêm 1 và gọi hàm exit

* Giải quyết như sau
* Payload1: Dùng format string để leak địa chỉ libc và overwrite *exit@got*

thành địa chỉ hàm main → mỗi khi chương trình gọi exit sẽ quay lại

hàm main → có vòng lặp vô tận, ngoài ra ta cần ghi đè giá trị của *a*

thành *-1* để thỏa điều kiện cho phép người dùng nhập input

* Payload2: Dùng format string để ghi đè *printf@got* thành địa chỉ hàm

*system* trong libc (ở lần lặp kế tiếp chương trình sẽ gọi *system(format)*

thay vì *printf(format)*), ghi đè giá trị của *a* thành -*1*

* Payload3: truyền vào chương trình chuỗi *“/bin/sh\x00”* → chương trình

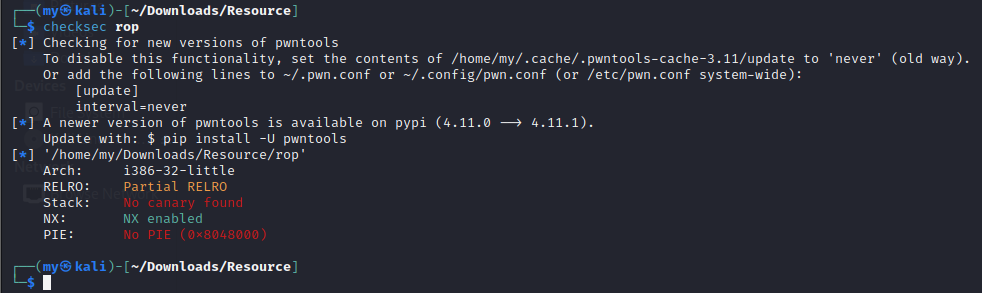
gọi *system(“/bin/sh\x00”)* và ta sẽ có shell để đọc flag

* Khai thác như sau:

Sử dụng checksec để kiểm tra, ta thấy chương trình chỉ có Partial RELRO và

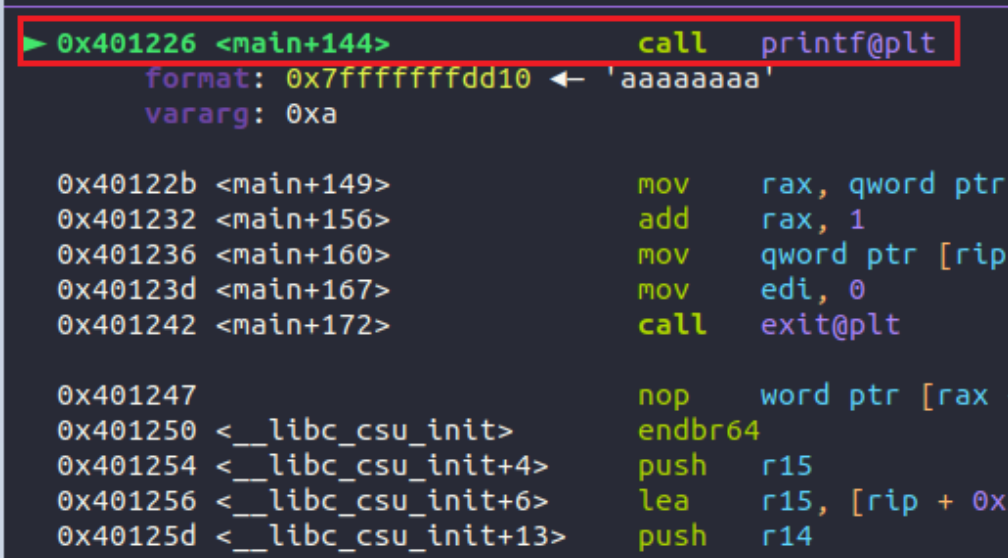
NO PIE, tức là ta dễ dàng có được các địa chỉ cần tìm cũng như ghi đè được

*exit@got / printf@got*



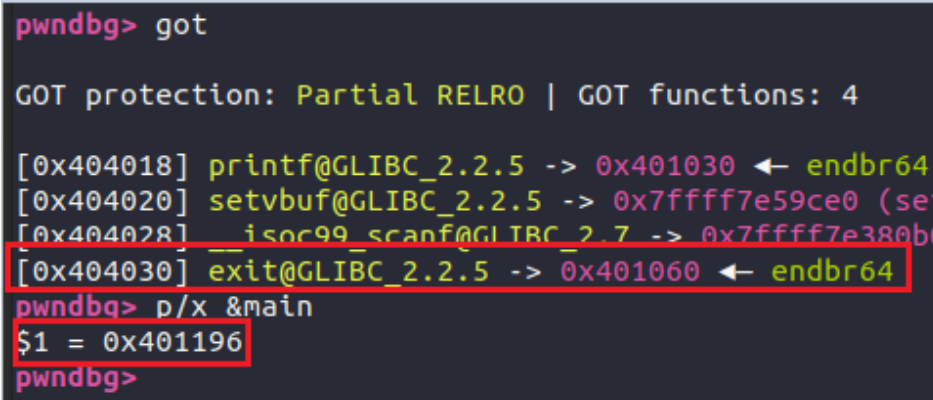
Debug với gdb, ta thấy input người dùng nhập vào sẽ nằm tại quadword đầu

tiên trên stack



Do đó, để trỏ tới input, ta sẽ sử dụng *%(6+0)$ = %6$\_*

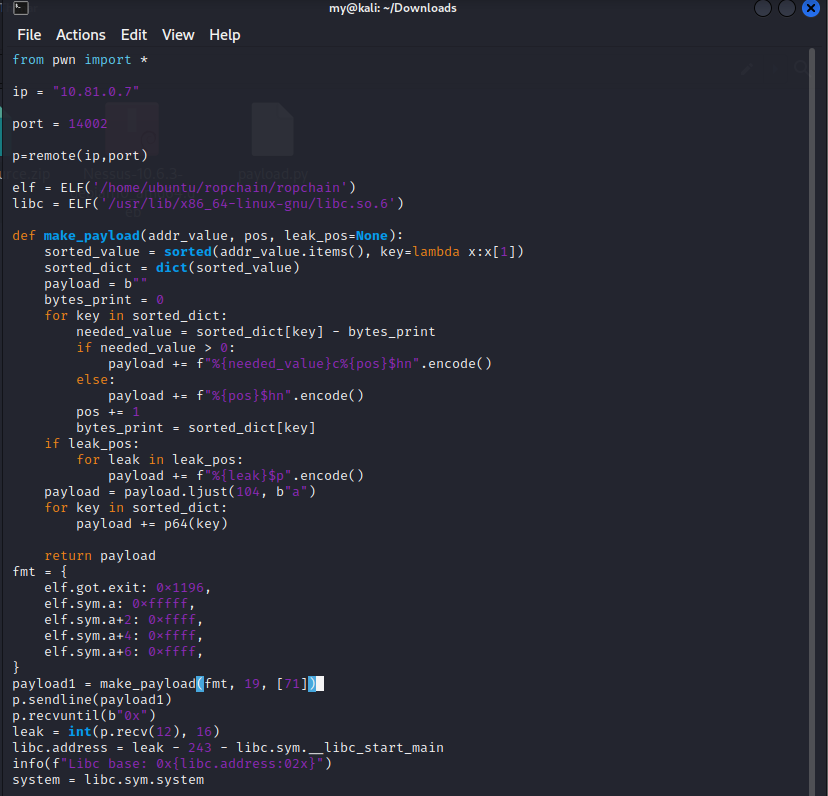
* Taoj payload
* Tạo payload 1 với tham số là dictionary chứa key là địa chỉ biến *a, exit@got* và value là giá trị *0xffffffffffffffff = -1*, địa chỉ hàm main *(0x401196)*
* Do giá trị tại exit@got và địa chỉ hàm main chỉ khác nhau 2 bytes đầu tiên, ta chỉ cần ghi 2 bytes vào exit@got



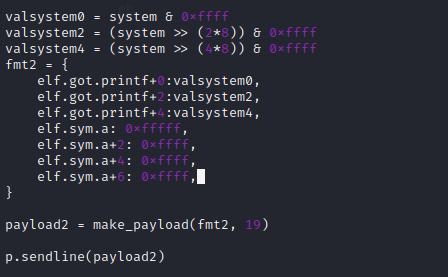
* Để có được địa chỉ hàm system, ta cần phải leak một địa chỉ libc trên stack
* Trên stack, quan sát thấy tại quadword thứ 65 có một địa chỉ thuộc libc *(\_\_libc\_start\_main)*



* Vậy ta sẽ dùng %(0x41+6)$p = %71$p để đọc được giá trị libc này



* Ta sẽ đọc giá trị libc được leak ra và cập nhật địa chỉ hàm system tương ứng
* Tạo payload 2 với tham số là dictionary chứa key là địa chỉ biến a, *printf@got* và value là giá trị 0xffffffffffffffff (-1), địa chỉ hàm system vừa tìm được



* Sau khi overwrite printf@got thành địa chỉ hàm system, ta chỉ cần gửi “/bin/sh\x00” sẽ có được shell

