

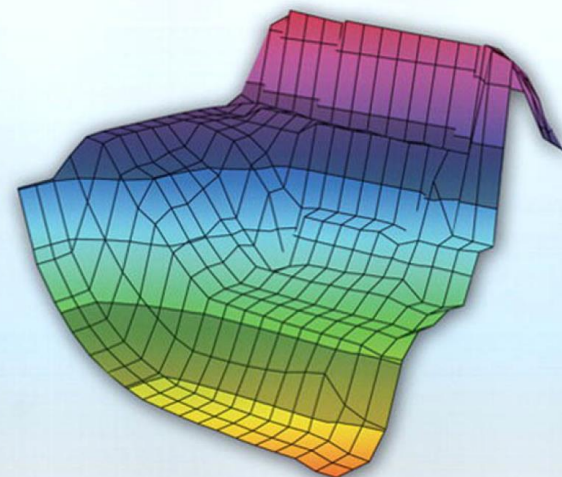
《计算机系统基础实验》

LAB3 - 缓冲区溢出攻击 2019 春季

华中科技大学

华中科技大学《计算机系统基础》课程组

2019-05



学术诚信

如果你确实无法完成实验，你可以选择不提交。如果抄袭，不管抄袭者还是被抄袭者，均以零分论处并按作弊处分。

■ 实验目的

加深对IA-32函数调用规则和栈结构的具体理解。

■ 实验内容

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**来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为。

■ 难度等级

分5个难度递增的等级，分别命名为Smoke (level 0)、Fizz (level 1)、Bang (level 2)、Boom (level 3) 和Nitro (level 4)，其中Smoke级最简单而Nitro级最困难。

■ 实验环境

C语言; linux

■ 实践技能

熟练运用gdb、objdump、gcc等工具。

二、实验数据

- 离线下载实验程序包： **lab3.tar**
- 在本地目录将程序包解压： **tar -xf lab3.tar**
- 数据包中至少包含下面四个文件：
 - * **bufbomb**: 可执行程序，攻击所用的目标程序bufbomb。
 - * **bufbomb.c**: C语言源程序，目标程序bufbomb的主程序。
 - * **makecookie**: 可执行程序，该程序基于你的学号产生一个唯一的由8个16进制数字组成的4字节序列（例如0x5f405c9a），称为“cookie”。
 - * **hex2raw**: 可执行程序，字符串格式转换程序（用法见后）。

另一个需要的文件是，用objdump工具反汇编bufbomb可执行目标程序，得到它的反汇编源程序：在后面的分析中，你将从这个文件中查找很多必需的信息。

三、目标程序BUFBOMB

bufbomb是一个可执行的目标程序，由bufbomb.c等源程序编译、链接后得到。

■ bufbomb的正常运行：

```
linux> ./bufbomb -u U201714557
```

其中，“-u U201714557” 是需要你提供的命令行参数，

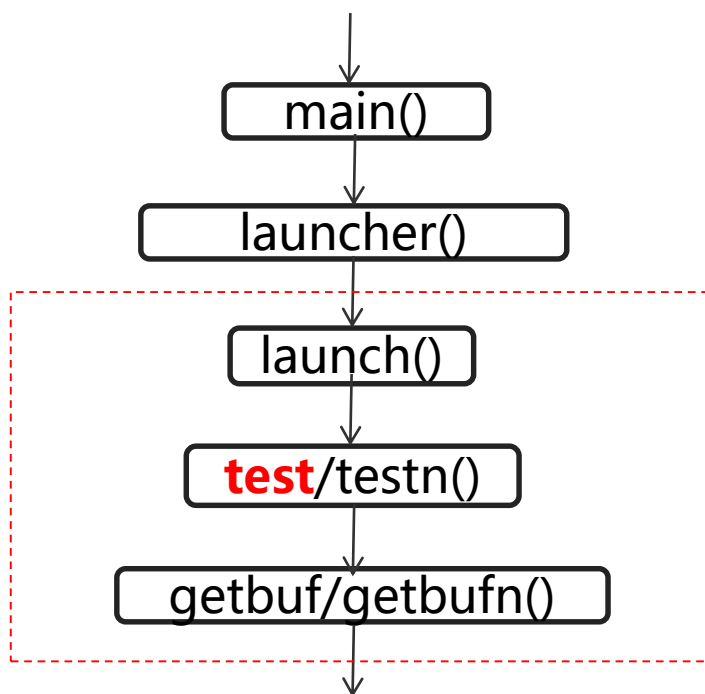
U201714XXX是你的学号。

注：ppt中的数据非实际数据

注：你的学号在bufbomb里将通过**getcookie**函数产生一个**cookie**（和使用makecookie完全一样的cookie），**cookie**将作为你的程序的**唯一标识**，而使得你的运行结果与其他同学不一样。

目标程序BUFBOMB (续)

- 为帮助你理解程序的行为，你可以简单分析一下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（!success），会看到信息“Better luck next time”，这时你就要继续尝试其它解了。

■ 本实验的主要内容从分析test函数开始。

test函数中调用了**getbuf**函数，getbuf函数的功能是从标准输入（stdin）读入一个字符串，正常时应返回test。

■ getbuf函数源程序如下（bufbomb.c里没有）：

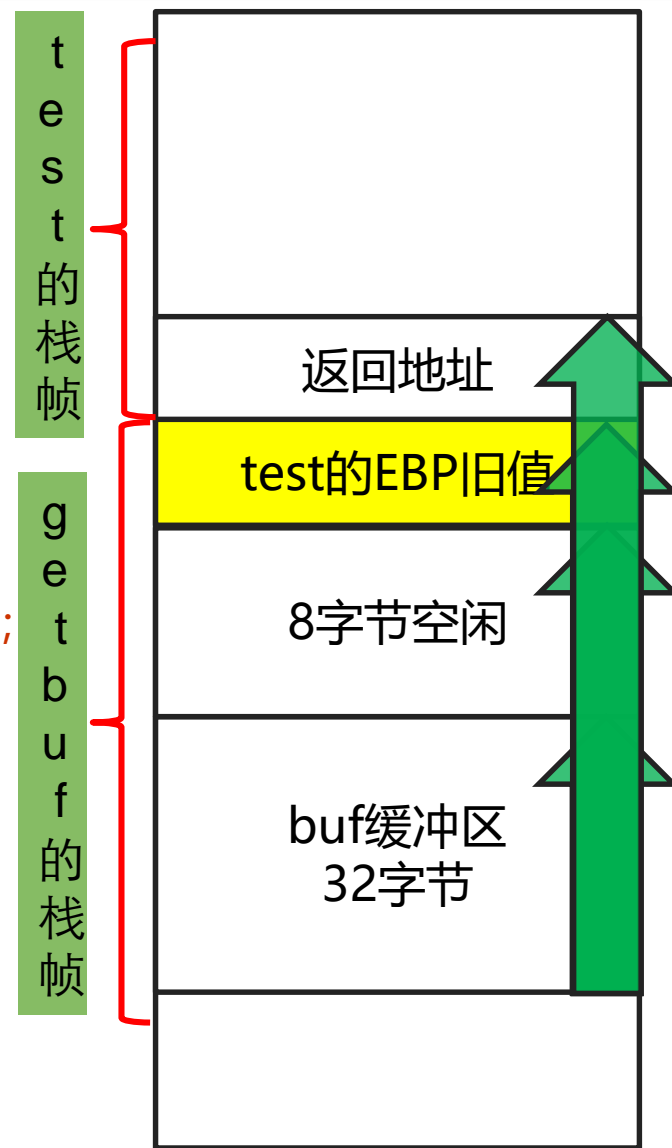
```
1 /* Buffer size for getbuf */
2 #define NORMAL_BUFFER_SIZE 32
3
4 int getbuf()
5 {
6     char buf[NORMAL_BUFFER_SIZE];
7     Gets(buf);
8     return 1; //正常时返回1
9 }
```

getbuf函数调用Gets函数从标准输入读入一个字符串，并将字符串存入指定的目标内存位置中，即getbuf中大小为32个字符的数组buf。

目标程序BUFBOMB (续)

■ test函数

```
void test()
{   int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval(); //getpid()
    val = getbuf(); //正常情况下getbuf返回值等于1
    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
        printf("Boom!: getbuf returned 0x%x\n", val);
        validate(3);
    } else {
        printf("Dud: getbuf returned 0x%x\n", val);
    }
}
```



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

■ 缓冲区攻击从getbuf函数开始

函数Gets()并不判断buf数组是否足够大，它只是简单地
向目标地址复制全部输入字符串，因此输入如果超出预先分配的
存储空间边界（32Byte），就会造成缓冲区溢出。

你可以尝试以下输入：

(a) 输入长度不超过31的字符串。运行示例如下：

linux>./ bufbomb -u U201714557

Type string: I love ICS2017

Dud: getbuf returned 0x1

(b) 输入超出31个字符的字符串，发生下列的错误：

```
unix> ./bufbomb -u U201714557
```

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

Ouch!: You caused a segmentation fault!

该错误信息的含义是缓冲区溢出导致程序状态被破坏，产生存储器访问错误，你可以从x86栈帧结构的组成来分析原因。

■ 本实验的任务从这里开始

需要你精心设计一些字符串输入给bufbomb，有意造成缓冲区溢出，而使bufbomb程序完成一些有趣的事情。

- **攻击字符串**：这些字符串称为“攻击字符串”（exploit string），由若干无符号字节数据构成，用**十六进制**表示，每两个十六进制数码组成一个字节，字节之间用空格隔开，如：

68 ef cd ab 00 83 c0 11 98 ba dc fe

- 每个攻击字符串一行，输入时最后以回车结束。
- **攻击字符串往往需要cookie作为其中的一部分**，所以每个同学的攻击字符串一般是独一无二的。

攻击字符串文件：

为了方便，你可以将攻击字符串写在一个文本文件中，攻击字符串文件的用法见后面（**六、攻击字符串文件和结果提交**）的说明。

四、Lab3 实验任务

本实验需要你构造5个攻击字符串，对目标程序bufbomb分别实施5次缓冲区溢出攻击。

5次攻击难度递增，分别命名为：

Smoke (level 0)

Fizz (level 1)

Bang (level 2)

Boom (level 3)

Nitro (level 4)

1) 任务一：Smoke

在bufbomb.c中查找smoke()函数，代码如下

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

任务：构造一个攻击字符串作为bufbomb的输入，而在getbuf()中造成缓冲区溢出，使得getbuf()返回时不是返回到test函数继续执行，而是**转向执行smoke**。

Lab3 实验任务（续）

1) 任务一：Smoke（续）

本阶段若成功，你将看到：

```
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!
```

2) 任务二：fizz

在bufbomb.c中查找fizz函数，其代码如下。

```
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);
}
```

fizz()有一个参数，这是与smoke不同的地方。函数中，该输入与系统的**cookie**（全局变量，里面含有根据你的学号生成的cookie）进行比较。

2) 任务二：fizz（续）

你的任务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，使得本次getbuf()返回时不是返回到test函数继续执行，而是**转向执行fizz()**。

与Smoke阶段不同和且较难的地方在于：**fizz函数需要一个输入参数**，因此你要设法将cookie值作为参数传递给fizz函数，以便于fizz中val与cookie的比较能够成功。所以，你需要仔细考虑**将cookie放置在栈中什么位置**。

Lab3 实验任务（续）

附：怎么生成你的cookie？

用makecookie工具，用法：

```
linux> makecookie U201714557
```

```
0x5f405c9a
```

注：ppt中的数据非实际数据

这里，“0x5f405c9a”即为根据学号U201714557生成的cookie。

2) 任务二：fizz（续）

本阶段成功你将看到：

```
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!
```

3) 任务三: Bang

在bufbomb.c中查找bang函数, 其代码如下。

```
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);
}
```

bang ()函数的功能大体和fizz()类似, 但val没有被使用, 而是一个**全局变量global_value与cookie进行比较**, 这里global_value的值应等于对应你的cookie才算成功, 所以**你要想办法将全局变量global_value设置为你的cookie值。**

3) 任务三：Bang（续）

本阶段的任务是**设计包含攻击代码的攻击字符串**，所含攻击代码首先将全局变量global_value设置为你的cookie值，然后转向执行bang()。

■ 任务的挑战：设计包含机器指令的攻击字符串。

本实验中，包含机器指令的攻击字符串在覆盖缓冲区时写入函数的栈帧，当被调用函数返回时，将转向执行这段攻击代码。

提示：你的攻击代码要实现：1) 将全局变量global_value设置为你的cookie值；2) 将bang函数的地址压入栈中；3) 附一条ret指令。而你还要做的是设法**将这段攻击代码置入栈中且将返回地址指向这段代码。**

3) 任务三：Bang（续）

提示2：如何构造含有攻击代码的攻击字符串？

所谓含有攻击代码的字符串，是指嵌有**二进制机器指令代码的字符串**，而二进制机器指令代码也就是一些无符号字节编码。

所以，要想构造攻击代码，可以手工编写这些指令的二进制字节编码，或者，可以首先编写一个汇编代码文件asm.s，然后使用gcc将该文件编译成机器代码，gcc命令格式：

```
gcc -m32 -c asm.s
```

然后再使用 “**objdump -d asm.o**” 命令将其反汇编，从中你可获得需要的**二进制机器指令字节序列**。

3) 任务三: Bang (续)

如果你成功了，你会看到一下结果

```
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!
```

4) 任务四：boom

前几阶段的实验实现的攻击都是使得程序跳转到不同于正常返回地址的其他函数中，进而结束整个程序的运行，因此，攻击字符串所造成的对栈中原有记录值的破坏、改写是可接受的。

然而，更高明的缓冲区溢出攻击是，除了执行攻击代码来改变程序的寄存器或内存中的值外，仍然使得程序能够返回到原来的调用函数继续执行——即调用函数感觉不到攻击行为。

挑战：这种攻击方式的难度相对更高，因为攻击者必须：

- (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) 还原对栈状态的任何破坏。

4) 任务四: boom (续)

本阶段的实验任务就是构造这样一个攻击字符串，使得 getbuf 函数不管获得什么输入，都能**将正确的cookie值返回给test函数**，而不是返回值1。除此之外，你的攻击代码应还原任何被破坏的状态，将正确返回地址压入栈中，并执行ret 指令从而真正返回到test函数。

本阶段如果你成功了，你会看到一下结果：

```
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

5) 任务五: Nitro

首先注意，本阶段你需要使用“-n”命令行开关运行bufbomb，以便开启Nitro模式，进行本阶段实验。

如下所示：

```
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
```

在Nitro 模式下，cnt=5（见目标程序BUFBOMB中相关说明）。亦即getbufn会连续执行了5次。

5) 任务五：Nitro（续）

为什么要连续5次调用getbufn呢？

通常，一个函数的栈的确切内存地址是随程序运行实例的不同而变化的。也就是一个函数的栈帧每次运行时都不一样。

之前实验中，bufbomb调用getbuf的代码经过了一定的处理，通过一些措施获得了稳定的栈地址，因此不同运行实例中，你观察到的getbuf函数的栈帧地址保持不变（自己去观察）。这使得你在之前实验中能够基于buf的已知的确切起始地址构造攻击字符串。

但是，如果将这样的攻击手段用于一般的程序时，你会发现你的攻击有时奏效，有时却导致段错误（segmentation fault）。

5) 任务五：Nitro（续）

实验任务：与阶段四类似，构造一攻击字符串使得getbufn函数（注，在**Nitro**阶段，bufbomb将调用testn函数和getbufn函数，见bufbomb.c）返回cookie值至testn函数，而不是返回值1。

此时，需要将cookie值设为函数返回值，复原/清除所有被破坏的状态，并将正确的返回位置压入栈中，然后执行ret指令以正确地返回到testn函数。

挑战：与boom不同的是，本阶段的每次执行栈（ebp）均不同，所以你要想办法保证每次都能够正确复原栈被破坏的状态，以使得程序每次都能够正确返回到test。

五、实验工具和技术

本次实验要求你要能较熟练地使用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：bufbomb没有调试信息，所以你基本上无法通过单步跟踪观察程序的执行情况。但你依然需要设置断点（b命令）来让程序暂停，并进而观察断点处必要的内存单元内容、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。gdb查看寄存器内容的指令：**info r**。

gcc：在阶段2~4，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反汇编成二进制字节数据和汇编代码，以此来构造具有攻击代码的攻击字符串。

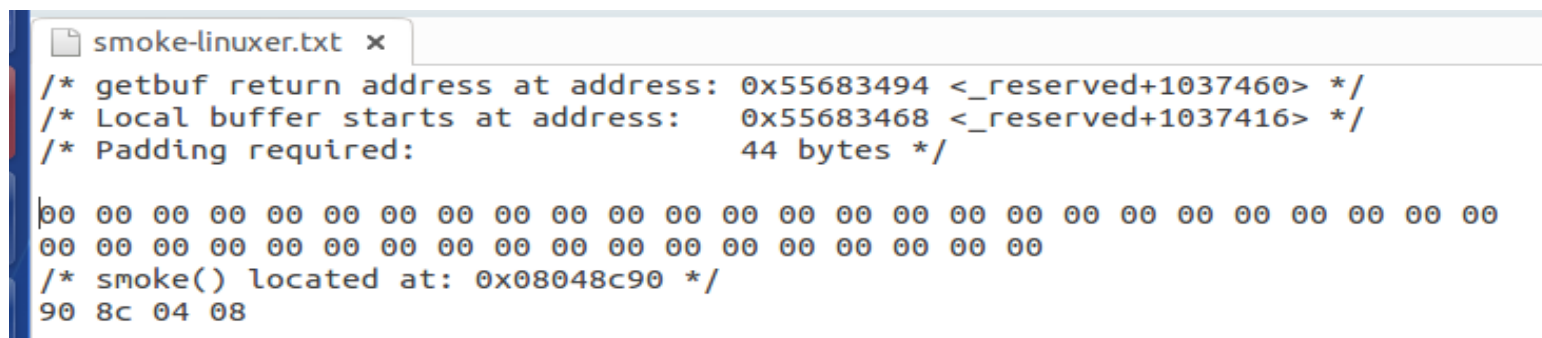
返回地址：**test函数调用getbuf后的返回地址是getbuf后的下一条指令的地址**（通过观察bufbomb反汇编代码可得）。而带有攻击代码的攻击字符串所包含的**攻击代码地址**，则需要你在深入理解地址概念的基础上，找到它们所在的位置并正确使用它们实现程序控制的转向。

六、攻击字符串文件和结果的提交

攻击字符串文件：

为了使用方面，你可以将攻击字符串写在一个文本文件中，该文件称为攻击文件（**exploit.txt**）。该文件中可以加入类似C语言的注释，只是需要在使用之前用hex2raw工具将注释去掉，生成相应的raw攻击字符串文件（**exploit_raw.txt**）。

例：学号U201714557的smoke阶段的攻击字符串文件命名为smoke_U201714557.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 00 00 00 00 00
/* smoke() located at: 0x08048c90 */
90 8c 04 08
```

六、攻击字符串文件和结果的提交（续）

攻击字符串文件的使用：

以smoke_ U201714557.txt为例说明攻击字符串文件的使用：首先将攻击字符串写入smoke_ U201714557.txt中（可带有注释）。然后用hex2raw进行转换，得到smoke_ U201714557_raw.txt。

方法一：命令行执行bufbomb时使用

使用I/O重定向将其输入给bufbomb：

```
linux> ./hex2raw smoke_ U201714557.txt smoke_ U201714557-raw.txt
```

```
linux> ./bufbomb -u U201714557 < smoke_ U201714557_raw.txt
```

六、攻击字符串文件和结果的提交（续）

方法二：在gdb中运行bufbomb时使用：

```
linux> gdb bufbomb
```

```
(gdb) run -u U201714557 < smoke_U201714557_raw.txt
```

方法三：如果你不想单步进行raw文件格式转换再代入bufbomb使用攻击字符串文件，也可以借助linux操作系统管道操作符和cat命令，按如下方式直接使用smoke_U201714557.txt：

```
linux> cat smoke_U201714557.txt |./ hex2raw | ./ bufbomb -u U201714557
```

六、攻击字符串文件和结果的提交（续）

对应本实验5个阶段的exploit.txt，请分别命名为：

- ◆ smoke_学号.txt 如：smoke_ U201714557 .txt
- ◆ fizz_学号.txt 如：fizz_ U201714557.txt
- ◆ bang_学号.txt 如：bang_ U201714557.txt
- ◆ boom_学号.txt 如：boom_ U201714557.txt
- ◆ nitro_学号.txt 如：nitro_ U201714557.txt

六、攻击字符串文件和结果的提交（续）

实验结果提交：

做为实验结果，你需要提交最多5个solution文件（即上面的五个txt文件），一起打包为zip文件，命名为：

班级_学号.zip，如CS1601_U201714557.zip

◆ 物联网 IT 计算机 CS 卓越班 ZY ACM班 ACM 校交班 IE

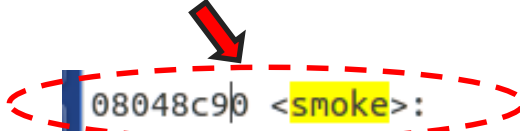
注： **5个文件分别包含5个阶段的攻击字符串**。每个文件包含一个字符序列，序列格式严格定义为：**两个16进制值作为一个16进制对，每个16进制对代表一个字节，每个16进制对之间用空格分开**，如

“68 ef cd ab 00 83 c0 11 98 ba dc fe”

（可以加注释和分行）。

七、任务一的实验示例

- 以任务一smoke为例介绍阶段一实验的详细操作步骤。
- 任务一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。为此，
 - 1) 在bufbomb的反汇编源代码中找到smoke函数，记下它的开始地址：



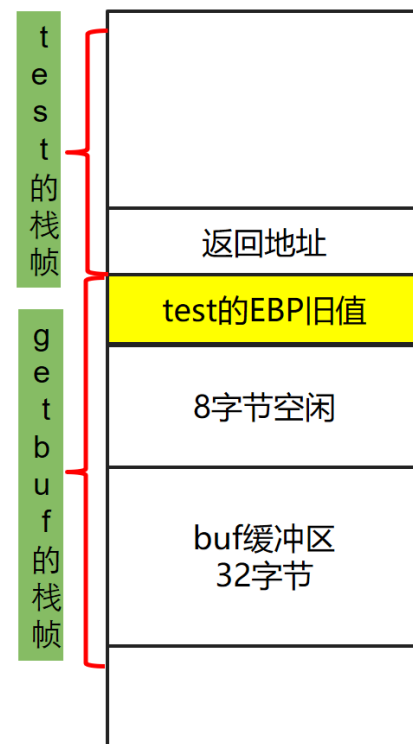
```
08048c90: <smoke>:
8048c90: 55                push    %ebp
8048c91: 89 e5             mov     %esp,%ebp
8048c93: 83 ec 18          sub     $0x18,%esp
8048c96: c7 04 24 13 a1 04 08 movl    $0x804a113,(%esp)
8048c9d: e8 ce fc ff ff    call   8048970 <puts@plt>
8048ca2: c7 04 24 00 00 00 00 movl   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

任务一实验示例（续）

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
```

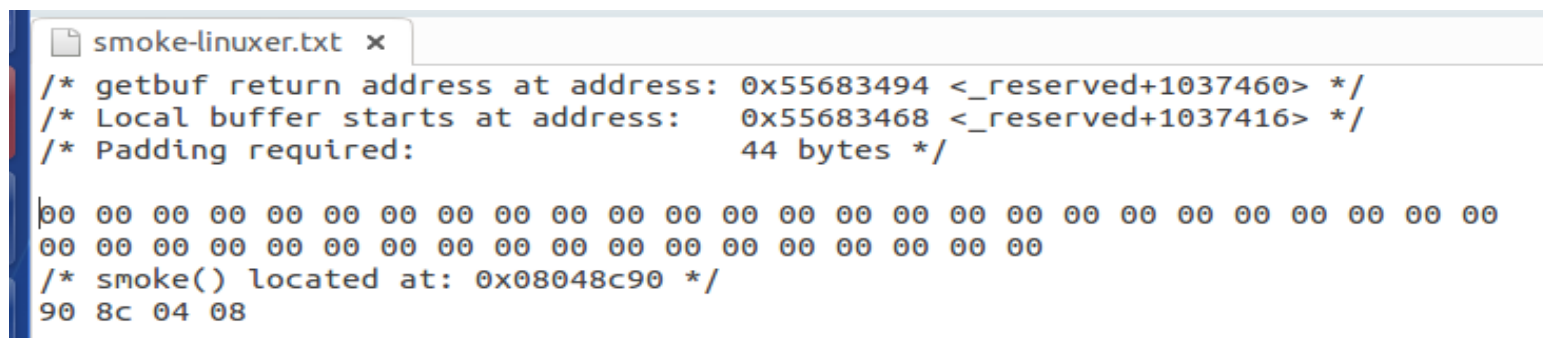


可以看到getbuf的栈帧是0x38+4个字节，buf缓冲区的大小是0x20个字节，还有8个空闲字节（共40个字节）。

注：smoke的地址是0x8048c90，而作为字节数据且为小端格式，字节值为“90 8c 04 08”。

任务一实验示例（续）

4) 可以将上述攻击字符串写在攻击字符串文件中，命名为 smoke_U201714557.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 00 00
/* smoke() located at: 0x08048c90 */
90 8c 04 08
```

可以看到，里面加了一些C风格的注释，以便于阅读和理解。smoke_U201714557.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而代入bufbomb中使用。

注：smoke_U201714557.txt文件中的注释，/*和*/与其后或前的字符之间必须要用空格隔开，否则会发生解析异常。

任务一实验示例（续）

5) 测试。

生成smoke_U201714557.txt后，可以用以下命令进行测试：

```
linux> cat smoke_U201714557.txt | ./ hex2raw | ./ bufbomb -u U201714557
```

如果正确无误，则显示如下：

```
Userid:U201714557
```

```
Cookie:0x5f405c9a
```

```
Type string:Smoke!: You called smoke()
```

```
VALID
```

```
NICE JOB!
```

至此，任务一成功完成。

八、实验报告和结果文件

- ◆ 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
- ◆ 最后以班为单位集中打包发送至1097412466@qq.com