LAB3 - 缓冲区溢出攻击

一、实验提交

请认真阅读以下内容，若有违反，后果自负

截止时间：请在规定的时间内完成实验并提交结果，如无特殊原因，迟交实验报告造成的成绩损失自负（即使迟了 1 秒），请大家合理分配时间。

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。.

提交方式：见后。

请你在实验截止前务必确认你提交的内容符合要求（格式及内容等）。如果由于你的原因给我们造成了不必要的麻烦，视情况而定，在本次实验中你将会被扣除一定的分数，最高可达 50%。

二、实验概述

本实验的目的在于加深对 IA-32 函数调用规则和栈结构的具体理解。实验的主要内容是对一个可执行程序“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，例如将给定的字节序列插入到其本不应出现的内存位置等。本次实验需要你熟练运用 gdb、objdump、gcc 等工具完成。

实验中你需要对目标可执行程序 BUFBOMB 分别完成 5 个难度递增的缓冲区溢出攻击。5 个难度级分别命名为 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。

实验语言：c；实验环境：linux

三、实验数据

本实验的数据包含于一个文件包 lab3.tar 中。下载该文件到本地目录中，然后利用“tar –xf lab3.tar”命令将其解压，至少包含下列四个文件：

* bufbomb：实验需要攻击的目标程序 bufbomb。

* bufbomb.c：目标程序 bufbomb 的主源程序。

* makecookie：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：字符串格式转换程序。

另一个需要的文件是，需要你如同 Lab2 一样，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，你将要从这个文件中查找很多信息。

四、目标程序 BUFBOMB

bufbomb 是一个可执行的目标程序，是通过编译、链接 bufbomb.c 等源程序得到的。你可以打开 bufbomb.c 分析一下。

1）bufbomb 在运行时需要你提供命令行参数，这里你要用到的命令行参数只有一个-u，方法如下：

unix> ./bufbomb -u U201614557

这里，-u U201614557 就是 bufbomb 的命令行参数，U201614557 是你的学号。你输入的学号在 bufbomb 里通过 getcookie(userid)，将如同 makecookie 一样，产生一个由 8 个 16 进制数字组成的 4 字节序列 cookie。Cookie 将作为你程序的唯一标示，而使得你的运行结果与其他同学不一样。这里你所关心的仅是按照命令行参数的要求输入学号，别的你可以不用关心。

2）进一步分析，你可以看到，bufbomb 的 main 函数里通过调用 launcher 函数进行后续的处理，launcher 函数被调用 cnt 次，但除了最后 Nitro 阶段，cnt 都只是 1。这也不是你关心的重点。

进入 launcher 程序，你会看到它又调用 launch 函数，你不用关心细节，而仅到 launch 函数里找到对 test()或 testn()函数的调用（testn 仅在 Nitro 阶段被调用，其余阶段均调用 test）。同时你会看到，以后各阶段，如果你的操作不符合预期（也就是执行不成功，!success）时，会打印“Better luck next time”，这是告诉你这一次没做对。如果你的输入产生这样的提示，你就要继续尝试其它解了。

3）**本实验的主要内容从分析 test 函数开始**。这需要你认真分析一下 test 函数的功能。提示你的是，test 函数调用一个名字叫 getbuf 的函数，该函数的功能是从标准输入（stdin）读入一个字符串。bufbomb.c 里没有 getbuf 函数的源程序，但告诉你如下：

```
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;
9  }
```

可见该函数的功能很简单：调用另一个函数 Gets（类似于标准库函数 gets，在 bufbomb.c 里面有它的源程序），从标准输入读入一个字符串（以换行‘\n’或文件结束 end-of-file 字符结尾），并将字符串（以 null 空字符结尾）存入指定的目标内存位置。在 getbuf 函数代码中，目标内存位置是具有 32 个字符大小的数组 buf。

getbuf 函数之所以重要，是因为所谓的缓冲区攻击就是从这里开始的。根本原因是，函数 Gets()并不判断 buf 数组是否足够大而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，就会造成缓冲区溢出。你可以尝试以下输入：

（a）输入到 getbuf()的字符串长度不超过 31 个字符长度。很明显，这时 getbuf()将正常返回 1，运行示例如下：

linux>./ bufbomb -u U201614557（回车，以下是屏幕上的显示）

Type string: I love ICS2016（回车，“I love ICS2016”为你输入的字符串）.

Dud: getbuf returned 0x1（这一行输出的由来你可以看一看 test 函数的源程序）

（b）但是，如果你输入一个超出 31 个字符的字符串，通常会发生类似下列的错误：

unix> ./bufbomb -u U201614557（回车）

Type string: It is easier to love this class when you are a TA.（输入一个长字符串，回车）

Ouch!: You caused a segmentation fault!（这一行输出的由来你也可以看一看 bufbomb.c，但不需要你关心细节，该错误信息所指为缓冲区溢出导致程序状态被破坏，产生存储器访问错误，至于为什么会产生一个段错误，你可以思考以下 x86 栈结构的组成来分析原因）

到这里为止，上述过程中，不管是缓冲区溢出还是不溢出，程序的反应都是程序控制下的“正常”行为。**下面才开始本实验的任务**，就是你精心设计一些字符串输入给 bufbomb（当然，这些字符串都是传送给了 getbuf），有意造成缓冲区溢出，而使 bufbomb 程序完成一些有趣的事情。这些字符串称为“攻击字符串”（exploit string）。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：（包括 cookie）是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，最后以回车结束（注：字符串中间不能包含其它回车），因为类似 gets()，Gets()函数以回车作为一行输入的结束。

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**），如 smoke_ U201614557.txt。为了便于阅读，该文件中可以加入类似 C 语言的注释，但需要在使用之前用 hex2raw 工具将注释去掉，生成仅包含攻击字符串而不含注释的 raw 攻击字符串文件（**exploit_raw.txt**），如 smoke_ U201614557_raw.txt。此时的用法如下：

首先将攻击字符串保写入文本文件 smoke_ U201614557.txt 中。然后 hex2raw 进行转换，得到 smoke_ U201614557_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_ U201614557.txt    smoke_ U201614557-raw.txt
linux> ./bufbomb -u U201614557    < smoke_ U201614557_raw.txt
```

方法二：在 gdb 中运行 bufbomb 时使用：

```
linux> gdb bufbomb
(gdb) run -u U201614557 < smoke_ U201614557_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_ U201614557.txt：

```
linux> cat smoke_ U201614557.txt |./ hex2raw    | ./ bufbomb –u U201614557
```

对应本实验 5 个阶段的 exploit.txt，请分别命名为：

smoke_学号.txt

fizz_学号.txt

bang_学号.txt

boom_学号.txt

nitro_学号.txt

如：

smoke_ U201614557 .txt

fizz_ U201614557.txt

bang_ U201614557.txt

boom_ U201614557.txt

nitro_ U201614557.txt

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“<userid>.zip”文件提交，如 U201614557.zip。

这 5 个文件分别包含完成 5 个阶段的攻击字符串，文件命名方式为“<level>_<userid>.txt”，其中 level 建议采取小写形式，例如“smoke_U201614557.txt”。每个文件包含一个字符串序列，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，例如“68 ef cd ab 00 83 c0 11 98 ba dc fe”。（可以加注释和分行）

五、实验任务

本实验需要你构造一些攻击字符串，对目标可执行程序 BUFBOMB 分别造成不同的缓冲区溢出攻击。实验分 5 个难度级分别命名为 Smoke（level 0）、Fizz（level 1）、Bang（level 2）、Boom（level 3）和 Nitro（level 4）。

1）任务一：Smoke

在 bufbomb.c 中查找 smoke()函数，其代码如下。简单分析一下 smoke()函数的功能（这个不重要）。

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

然后，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。如果你成功了，你会看到一下结果：

```
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!
```

注：你的攻击字符串可能会同时破坏了与本阶段无关的栈结构部分，但这不会造成问题，因为 smoke 函数会使程序直接结束。

本任务将作为示例，后面详细讲解如何操作。

2）任务二：fizz

在 bufbomb.c 中查找 fizz 函数，其代码如下。同样，简单分析一下 fizz ()函数的功能，但这个不重要。

```
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。在函数内部，该输入与系统的 cookie（里面含有根据你的 userid 生成的 cookie）进行比较。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，因此你要设法将使用 makecookie 得到的 cookie 值作为参数传递给 fizz 函数，你需要仔细考虑将 cookie 放置在栈中什么位置。

makecookie 的用法：

```
linux> makecookie U201614557
0x5f405c9a
```

“0x5f405c9a”即为根据学号 U201614557 生成的 cookie。

本阶段如果你成功了，你会看到一下结果：

```
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!
```

2）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);
}
```

分析一下 bang ()函数的功能，大体和 fizz()类似，但 bang()中，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 userid 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。

本阶段任务将带来挑战：更复杂的缓冲区攻击将在攻击字符串中包含实际的机器指令，并在攻击字符串覆盖缓冲区时写入函数（这里是 getbuf()）的栈帧，并进而将原返回地址指针改写为位于栈帧内的攻击机器指令的开始地址。这样，当被调用函数（getbuf()）返回时，将转向这段攻击代码执行，而攻击代码将使得程序转向执行 bang()，而不是返回上层的 test()函数。你可以想象到，使用这种攻击方式，就可以使被攻击程序做任何事了。

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 的值设置为你的 cookie 值，然后转向执行 bang()。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!
```

提示 1：此类攻击具有一定难度，你的攻击代码首先应将全局变量 global_value 设置为你的 cookie 值，同时将 bang 函数的地址压入栈中，最后附一条 ret 指令（从而可以跳至 bang 函数的代码继续执行。这里不要试图利用 jmp 或者 call 指令跳到 bang 函数的代码中，因为这些指令使用相对 PC 的寻址，很难正确达成执行 bang()函数目标）。你必须设法将这段攻击代码置入栈中且将返回地址指针指向这段代码的起始位置。

你可以使用 gdb 获得构造攻击字符串所需的信息。例如，在 getbuf 函数里设置一个断点并执行到该断点处，进而确定 global_value 和缓冲区等变量的地址。

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，如 objdump 反汇编后左侧第二列的机器指令代码。

```
804887c:        83 ec 04                sub    $0x4,%esp
804887f:        e8 00 00 00 00          call   8048884 <_init+0xc>
8048884:        5b                      pop    %ebx
```

所以，要想构造攻击代码，你可以手工编写指令的二进制字节编码，或者，你可以首先编写一个可以实现相应功能的汇编代码文件 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

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。

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（例如 test）继续执行——即调用函数感觉不到攻击行为。

然而，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。

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!
```

提示：boom 不是一个函数，你的任务仅是设计一个可以使 getbuf()能够返回正确 cookie 值的攻击字符串。你要保证 getbuf()能够正确返回 test()中继续执行，而不是像前几个阶段一旦转向，就不能回到 test()继续执行。

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 Nigro 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 linux
er
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（见四、目标程序 BUFBOMB 中相关说明）。亦即 getbufn 会连续执行 5 次。为什么要连续 5 次调用 getbufn 呢？

通常，一个给定的函数的栈的确切内存地址随程序运行实例（特别是运行用户）的不同而不同。其中一个原因是当程序开始执行时，所有环境变量（evn）的值所在内存位置靠近栈的基地址，而环境变量的值是做为字符串存储的，视值的不同需要不同数量的存储空间。因此，为一特定运行用户分配的栈空间取决于其环境变量的设置。

此外，当在 gdb 中运行程序时，程序的栈地址也会存在差异，因为 gdb 使用栈空间保存其自己的状态。之前实验中，在 bufbomb 调用 getbuf()的代码是经过一定的处理，通过一定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。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，如果你尝试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）。

本阶段的实验任务与阶段四类似，即构造一攻击字符串使得 getbufn 函数（注，在 kaboom 阶段，bufbomb 将调用 testn 函数和 getbufn 函数，源程序代码见 bufbomb.c）返回 cookie 值至 testn 函数，而不是返回值 1。此时，这需要你的攻击字符串将 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。

与 boom 不同的是，本阶段的每次执行栈（ebp）均不同，分析 bufbomb.c 函数可知，程序使用了 random 函数造成栈地址的随机变化，使得栈的确切内存地址每次都不相同。所以此时，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。

提示 1：在本阶段，程序将调用与 getbuf 略有不同的 getbufn 函数，区别在于 getbufn 函数使用如下 512 字节大小的缓冲区，以方便你利用更大的存储空间构造可靠的攻击代码：

```
/* Buffer size for getbufn */
#define KABOOM_BUFFER_SIZE 512
```

而且调用 getbufn 函数的代码会在栈上分配一随机大小的内存块（也就是利用 random 函数模拟栈地址的随机变化），使得如果你跟踪前后两次调用 getbufn 时%ebp 寄存器中的值，会发现它们呈现一个随机的差值。

提示 2：本阶段的技巧在于合理使用 nop 指令，该指令的机器代码只有一个字节（0x90）。

六、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 等工具，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。然后查看该文件，以确定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，可以通过观察 bufbomb 反汇编代码得到。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。

七、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（Smoke）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，你需要：

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>
```

如以上实例中，smoke 的开始地址是 0x8048c90

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                      push   %ebp
 80491ed:       89 e5                   mov    %esp,%ebp
 80491ef:       83 ec 38                sub    $0x38,%esp
 80491f2:       8d 45 d8                lea    -0x28(%ebp),%eax
 80491f5:       89 04 24                mov    %eax,(%esp)
 80491f8:       e8 55 fb ff ff          call   8048d52 <Gets>
 80491fd:       b8 01 00 00 00          mov    $0x1,%eax
 8049202:       c9                      leave
 8049203:       c3                      ret
```

如以上实例，你可以看到 getbuf 的栈帧是 0x38+4 个字节，而 buf 缓冲区的大小是 0x28（40 个字节）。

3）设计攻击字符串。攻击字符串的功能是用来覆盖 getbuf 函数内的数组 buf（缓冲区），进而溢出并覆盖 ebp 和 ebp 上面的返回地址，所以攻击字符串的大小应该是 0x28+4+4=48 个字节。并且其最后 4 个字节应是 smoke 函数的地址，正好覆盖 ebp 上方的正常返回地址。这样再从 getbuf 返回时，取出的根据攻击字符串设置的地址，就可实现控制转移。

所以，这样的攻击字符串为：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 90 8c 04 08，总共 48 个字节，并且前面 44 个字节可以为任意值，对程序的执行没有任何影响，只要最后四个字节正确地设置为“90 8c 04 08”即可。注：smoke 的地址是 0x8048c90，作为字节数据且为小端格式，所以字节值为“90 8c 04 08”。

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_U201614557.txt，内容可为：

smoke-linuxer.txt ×

```
/* 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
/* 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_U201614557.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_U201614557.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。

5）测试。

生成 smoke_U201614557.txt 后，可以用以下命令进行测试：

```
linux> cat smoke_U201614557.txt |./ hex2raw    | ./ bufbomb –u U201614557
```

如果正确无误，则显示如下：

```
Userid:U201614557
Cookie:0x5f405c9a
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

至此，任务一成功完成。