

华中科技大学 计算机学院

# [计算机系统基础]

[实验任务书 V1.2]

2017-5-15

# LAB1、数据表示实验

## 1. 实验简介

本实验的目的是更好地熟悉和掌握计算机中整数和浮点数的二进制编码表示。实验中，你需要解开一系列编程“难题”——使用有限类型和数量的运算操作实现一组给定功能的函数，在此过程中你将加深对数据二进制编码表示的了解。

实验语言：c； 实验环境： linux

## 2. 实验数据文件

实验所需要的代码和相关文件已打包成一个 tar 文件 lab1-handout.tar 供下载，下载后的文件存放到 linux 主文件夹 (/home/目录下)，进入 terminal，输入如下命令：

```
$ ./cd ~ #进入用户 home 目录，注意中间有空格
$ ./ls #显示当前目录下的文件，请学习 ls 用法
$ tar vxf lab1-handout.tar # 解压代码文件，输入时可以输入 tar vxf lab +tab 键
$ cd lab1-handout # 进入代码目录，输入时可以输入 cd lab1- +tab 键
$ ls # 显示当前目录下的文件，会看到如下文件
```

(不要用图形界面的 gzip 工具解开，会损失文件执行权限等属性，造成文件无法运行)

- README 有关实验细节的说明文件，请在开始实验前仔细阅读
- bits.c 包含一组用于完成指定功能的函数的代码框架，需要你按要求补充完成其函数体代码并“作为实验结果提交”。函数的功能与实现要求详细说明在相应函数和文件首部的注释中（务必认真阅读和遵照说明完成实验）。
- bits.h 头文件
- btest.c 实验结果测试工具，用于检查作为实验结果的 bits.c 中函数实现是否满足实验的功能正确性要求。
- btest.h, decl.c, tests.c 生成 btest 程序的源文件
- dlc 实验结果检查工具，用于判断作为实验结果的 bits.c 中函数实现是否满足实验的语法规则要求。
- Makefile 生成 btest、fshow、ishow 等工具的 Make 文件。
- ishow.c 整型数据表示查看工具，需要查看机器码时可以用

- fshow.c 浮点数据表示查看工具，需要查看机器码时可以用

```
$ gedit bits.c # 编辑工作文件
```

用户还可以用 vi, vim, emacs 甚至安装 sublime 来编辑文件，建议大家有时间可以学习一下 vi 等文本编辑命令，无需鼠标即可完成所有编辑，是所有 linux 程序员必须掌握的技能，至少要了解一下，能简单修改配置文件。

### 3. 实验内容

需要完成 bits.c 中下列函数功能，具体分为三大类：**位操作、补码运算和浮点数操作**。

#### 1) 位操作

表 1 列出了 bits.c 中一组操作和测试位组的函数。其中，“级别”栏指出各函数的难度等级（对应于该函数的实验分值），“功能”栏给出函数应实现的输出（即功能），“约束条件”栏指出你的函数实现必须满足的编码规则（具体请查看 bits.c 中相应函数注释），“最多操作符数量”指出你的函数实现中允许使用的操作符的最大数量。你也可参考 tests.c 中对应的测试函数来了解所需实现的功能，但是注意这些测试函数并不满足目标函数必须遵循的编码约束条件，只能用做关于目标函数正确行为的参考。

表 1 位操作题目列表

级别	函数名	功能	约束条件	最多操作符数
1	lsbZero	将 x 的最低有效位（LSB）清零	仅能使用! ~ & ^   + << >>	5
2	byteNot	将 x 的第 n 个字节取反（字节从 LSB 开始到 MSB 依次编号为 0-3）	仅能使用! ~ & ^   + << >>	6
2	byteXor	比较 x 和 y 的第 n 个字节（字节从 LSB 开始到 MSB 依次编号为 0-3），若不同，则返回 1；若相同，则返回 0	仅能使用! ~ & ^   + << >>	20
3	logicalAnd	x && y	仅能使用! ~ & ^   + << >>	20
3	logicalOr	x    y	仅能使用! ~ & ^   + << >>	20
3	rotateLeft	将 x 循环左移 n 位	仅能使用! ~ & ^   + << >>	25

4	parityCheck	若 x 有奇数个 1，则返回 1；否则，返回 0	仅能使用! ~ & ^   + << >>	20
---	-------------	--------------------------	--------------------------	----

## 2) 补码运算

表 2 列出了 bits.c 中一组使用整数的补码表示的函数。可参考 bits.c 中注释说明和 tests.c 中对应的测试函数了解其更多具体信息。

表 2 补码运算题目列表

级别	函数名	功能	约束条件	最多操作符数
2	mul2OK	计算 $2 * x$ ，如果不溢出，则返回 1，否则，返回 0	仅能使用 ~ & ^   + << >>	20
2	mult3div2	计算 $(x * 3) / 2$ ，朝零方向取整	仅能使用! ~ & ^   + << >>	12
3	subOK	计算 $x - y$ ，如果不溢出，则返回 1，否则，返回 0	仅能使用! ~ & ^   + << >>	20
4	absVal	求 x 的绝对值	仅能使用! ~ & ^   + << >>	10

## 3) 浮点数操作

表 3 列出了 bits.c 中一组浮点数二进制表示的操作函数。可参考 bits.c 中注释说明和 tests.c 中对应的测试函数了解其更多具体信息。**注意 float\_abs 的输入参数和返回结果（以及 float\_f2i 函数的输入参数）均为 unsigned int 类型，但应作为单精度浮点数解释其 32 bit 二进制表示对应的值。**

表 3 浮点数操作题目列表

级别	函数名	功能	约束条件	最多操作符数
2	float_abs	返回浮点数 'f' 的二进制表示，当输入参数是 NaN 时，返回 NaN	仅能使用任何整型/无符号整型操作，包括  , &&以及 if, while 控制结构	10
4	float_f2i	返回浮点数 'f' 的强制整型转换 "(int)f" 表示	仅能使用任何整型/无符号整型操作，包括  , &&以及 if, while 控制结构	30

## 4. 实验要求

实验前请认真阅读本文档和 bits.c 中的代码及注释，然后根据要求相应完成 bits.c 中的各函数代码。

实验中实现的函数代码必须满足下述基本条件（更多具体要求见函数的注释）：

- 除关于浮点数的函数实现外，只能使用顺序程序结构，不得使用循环或条件分支控制程序结构，例如 `if`, `do`, `while`, `for`, `switch` 等。
- 仅能使用有限类型和数量的 C 语言算术和逻辑操作，例如如下的操作符，但注意每个题目可能有不同的可用操作符列表，详见具体函数说明。`! ~ & ^ | + << >>`
- 不得使用超过 8 位表示的常量（即其值必须位于[0,255]中）。
- 不得使用任何形式的强制类型转换。
- 不得使用除整型外的任何其它数据类型，如数组、结构、联合等。
- 不得定义和使用宏。
- 不得定义除已给定的框架函数外的其他函数，不得调用任何函数。
- 具体的函数功能和实现要求可参看 `bits.c` 各函数框架的注释，以注释为准。
- 特定于浮点数操作函数的额外限制条件：
  - ◆ 可以使用循环和条件控制；
  - ◆ 可以使用整型和无符号整型常量及变量（取值不受[0,255]限制）；
  - ◆ 不使用任何浮点数据类型、操作及常量。
  - ◆ 关于浮点数的函数实现可以使用标准的程序结构(选择、循环均可使用)，可以使用 `int` 和 `unsigned` 两种整型数据，不得使用浮点数据类型、`struct`、`union` 或数组结构。关于浮点数的函数均使用 `unsigned` 型数据表示浮点数据。
  - ◆ `float_abs` 等函数必须能处理全范围的变量值，包括(NaN)和 `infinity`.为简化问题，若要返回 NaN 值，可使用 `0x7FC0000` 表示。

## 5. 代码检查

如前所述，实验数据包中包含两个工具程序可帮助检查你的代码的正确性。**必须通过这两个工具检查后才能提交代码，否则无法自动评分，将按零分计算成绩。**

### 1) 使用 `dlc` 检查函数实现代码是否符合实验要求的编码规则

完成 `bits.c` 后，调用如下命令进行检查：

```
$ ./dlc bits.c
```

如果 `dlc` 发现了错误，例如出现不允许使用的操作符、过多数量的操作符或者非顺序的代码结构，则将返回错误信息。如果程序代码满足要求，`dlc` 将不输出任何提示。

使用 `-e` 选项调用 `dlc`

```
$ ./dlc -e bits.c
```

可使 `dlc` 打印出每个函数使用的操作符数量。

输入 “`./dlc -help`” 可打印出 `dlc` 的可用命令行选项列表。

## 2) 使用 `btest` 检查函数实现代码的功能正确性

首先使用如下命令编译生成 `btest` 可执行程序：

```
$ make # 编译
```

如下调用 `btest` 命令检查 `bits.c` 中所有函数的功能正确性：

```
$ ./btest # 功能检查
```

注意每次修改 `bits.c` 后都必须使用 `make` 命令重新编译生成 `btest` 程序。

为方便依次检查测试每一函数的正确性，可如下在命令行使用 “`-f`” 选项跟上函数名，以要求 `btest` 只测试所指定的函数：

```
$ ./btest -f byteNot #制定函数功能检查
```

进一步可如下使用 “`-1, -2, -3`” 等选项在函数名后输入特定的函数参数：

```
$ ./btest -f byteNot -1 0xf -2 1
```

（`README` 文件中有关于 `btest` 程序的使用说明）

## 3) 挑战教授记录

当所有任务完成后可以试试和教授提交的标准答案进行 PK，运行如下命令

```
$ linux> ./driver.pl -u username          #用户名用 CS1405_tanzhihu  班级_姓名
```

访问挑战排行榜网站 <http://211.69.198.69:8080/> 即可查看自己与其他人提交的记录

### Scoreboard for the Data Lab "Beat the Prof" Contest

This page shows the operator counts for the students who have submitted entries to the Data Lab "Beat the Prof" contest.

- To enter the contest, run the driver with the -u option: `./driver.pl -u "nickname"`.
- Enter as often as you like. The page will show only your most recent submission.
- Blue entries match the instructor. Red entries beat the instructor. Incorrect entries are denoted by "-".
- Entries are sorted by score, defined as *(total instructor operations - total student operations)*. Higher scores are better.
- If all of your puzzle entries are correct and they each match or beat the instructor, then you're a **winner!**

Puzzle key: 1=lsbZero, 2=byteNot, 3=byteXor, 4=logicalAnd, 5=logicalOr, 6=rotateLeft, 7=parityCheck, 8=mul2OK, 9=mult3div2, 10=subOK, 11=absVal, 12=float\_abs, 13=float\_f2i

Last updated: Wed Aug 17 22:47:00 2016 (updated every 30 secs)

1	2	3	4	5	6	7	8	9	10	11	12	13	Winner?	Score	Nickname
3	3	9	8	5	16	11	4	6	12	5	4	13	Winner!	0	tanzhihu
3	3	9	8	5	16	11	4	6	12	5	4	13	Winner!	0	totalcontrol
3	3	9	8	5	16	11	4	6	12	5	4	13		0	tiger

红色表示挑战成功，蓝色表示对应函数挑战教授成功，短横线表示挑战失败！

## 6. 建议与提示

1) 如果你的代码不能完全满足相应函数的操作符使用限制，你可以获得部分得分，但是往往这样的次优解总能找到改进它的方法，从而获得正确解答。

2) 在 `bits.c` 文件中不要包含 `<stdio.h>` 头文件，因为这样将给 `dlc` 程序造成困难并产生一些难以理解的错误信息。注意尽管未包含 `<stdio.h>` 头文件，你仍然可以在 `bits.c` 中调用 `printf` 函数进行调试，`gcc` 将打印警告信息但你可以忽略它们。

3) 注意 `dlc` 程序使用比 `gcc` 和 `C++` 更严格的 C 变量声明形式。在由 “{ }” 包围的一个代码块中，所有变量声明必须出现在任何非声明语句之前。例如，针对下述代码，`dlc` 将报错

```
int foo(int x)
```

```
{    int a = x;
    a *= 3;    /* Statement that is not a declaration */
    int b = a; /* ERROR: Declaration not allowed here */
}
```

你必须类似如下代码将变量声明放在最前：

```
int foo(int x)
```

```
{    int a = x;
    int b;
    a *= 3;
    b = a;
}
```

## 7. 实验结果提交

请将完成函数体代码后的 `bits.c` 按以下命名规范命名后作为实验结果提交。**本次实验不需要提交实验报告，但各函数实现中应有详细注释描述解题思路。**按班为单位提交电子版本压缩文件。

### ◆ 专业命名规范

信安 IS    物联网 IT    计算机 CS    卓越班 ZY    ACM 班 ACM

### ◆ 文件命名规范

CS1201\_U201214795\_姓名.c

### ◆ 电子版按班为单位集中打包发送至 130757@qq.com 谭志虎老师处归档



## LAB2、二进制拆弹实验

### ➤ 请认真阅读以下内容，若有违反，后果自负

- 截止时间：在规定的时间内提交作业。（如无特殊情况，迟交的作业将损失 50% 的成绩（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，作为学术诚信的奖励，你将会获得 10% 的分数。
- 请你在实验截止前务必确认你提交的内容符合要求（格式、相关内容等），你可以下载你提交的内容进行确认。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%。

### 1. 实验简介

本实验中，你要使用课程所学知识拆除一个“binary bombs”来增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。

一个“binary bombs”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，包含了 6 个阶段（phase1~phase6）。炸弹运行的每个阶段要求你输入一个特定的字符串，若你的输入符合程序预期的输入，该阶段的炸弹就被“拆除”，否则炸弹“爆炸”并打印输出 "BOOM!!!" 字样。实验的目标是拆除尽可能多的炸弹层次。

每个炸弹阶段考察了机器级语言程序的一个不同方面，难度逐级递增：

- \* 阶段 1：字符串比较
- \* 阶段 2：循环
- \* 阶段 3：条件/分支
- \* 阶段 4：递归调用和栈
- \* 阶段 5：指针
- \* 阶段 6：链表/指针/结构

另外还有一个隐藏阶段，但只有当你在第 4 阶段的解之后附加一特定字符串后才会出现。

为了完成二进制炸弹拆除任务，你需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代

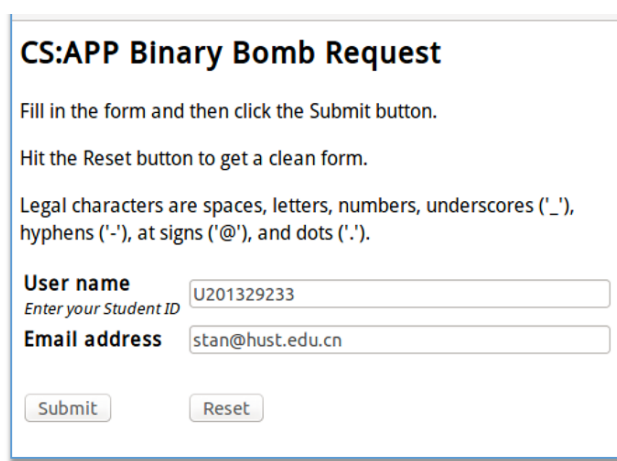
码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。这可能需要你在每一阶段的开始代码前和引爆炸弹的函数前设置断点，以便于调试。

实验语言：C 语言 at&t 汇编

实验环境：linux

## 2. 实验数据文件

本次实验中，每位同学会得到一个不同的 **binary bomb** 二进制可执行程序及其相关文件，访问 <http://211.69.198.69:15213/>，输入学号，email，点击 submit 即可获得你个人的专属代码包，可多次下载，但每次代码均不同。



**CS:APP Binary Bomb Request**

Fill in the form and then click the Submit button.

Hit the Reset button to get a clean form.

Legal characters are spaces, letters, numbers, underscores ('\_'), hyphens ('-'), at signs ('@'), and dots ('.').

**User name**  
Enter your Student ID

**Email address**

其中包含如下文件：

- **bomb**: bomb 的可执行程序。
- **bomb.c**: bomb 程序的 main 函数。

**bomb** 是一个可执行程序，需要 0 或 1 个命令行参数（详见 **bomb.c** 源文件中的 **main()** 函数）。如果运行时不指定参数，则该程序打印出欢迎信息后，期待你按行输入每一阶段用来拆除炸弹的字符串，并根据你当前输入的字符串决定你是通过相应阶段还是炸弹爆炸导致任务失败。

你也可将拆除每一阶段炸弹的字符串按行组织在一个文本文件中（比如：**result.txt**），然后作为运行程序时的唯一一个命令行参数传给程序（**./bomb result.txt**），程序会自动读取文本文件中的字符串，并依次检查对应每一阶段的字符串来决定炸弹拆除成败。

### 3. 实验提交要求

提交文件名：**学号.txt**（也就是上面的 result.txt，需要用你的学号改个名称）

提交文件格式：每个炸弹字符串一行，除此之外不要包含任何其它字符。

范例如下。

```
string1
string2
.....
string6
string7
```

### 4. 实验工具

本实验所需使用的一些软件工具和用法：

#### 1) Gdb

为了从二进制可执行程序“./bomb”中找出触发 bomb 爆炸的条件，可使用 gdb 来帮助你对程序进行分析。

GDB 是 GNU 开源组织发布的一个强大的交互式程序调试工具。GDB 主要帮忙你完成下面几方面的功能（更详细描述可参看 GDB 文档和相关资料）：

- 装载、启动被调试的程序。
- 让被调试的程序在你指定的调试断点处中断执行，方便查看程序变量、寄存器、栈内容等运行现场数据。
- 动态改变程序的执行环境，如修改变量的值。

#### 2) objdump 反汇编工具

##### ➤ objdump -t

该命令可以打印出 bomb 的符号表。符号表包含了 bomb 中所有函数、全局变量的名称和存储地址。你可以通过查看函数名得到一些目标程序的信息。

##### ➤ objdump -d

该命令可用来对 bomb 中的二进制代码进行反汇编。通过阅读汇编源代码可以发现 bomb 是如何运行的。但是，objdump -d 不能告诉你 bomb 的所有信息，例如

一个调用 `sscanf` 函数的语句可能显示为：

```
8048c36: e8 99 fc ff ff call 80488d4 <_init+0x1a0>
```

所以，你还需要 `gdb` 来帮助你确定这个语句的具体功能。

### 3) strings

该命令可以显示二进制程序中的所有可打印字符串。

## 5. 实验步骤提示

下面以 `phase1` 为例介绍一下基本的实验步骤：

**第一步：**调用 “**`objdump -d bomb > disassemble.txt`**” 对 `bomb` 进行反汇编并将汇编源代码输出到 “`disassemble.txt`” 文本文件中。

**第二步：**查看该汇编源代码文件 `disassemble.txt`，你可以在 `main` 函数中找到如下语句（通过查找 “`phase_1`”），从而得知 `phase1` 的处理程序包含在 “`main()`” 函数所调用的函数 “`phase_1()`” 中：

```
8048a4c: c7 04 24 01 00 00 00    movl    $0x1,(%esp)
8048a53: e8 2c fd ff ff          call    8048784 <__printf_chk@plt>
8048a58: e8 49 07 00 00          call    80491a6 <read_line>
8048a5d: 89 04 24                mov     %eax,(%esp)
8048a60: e8 a1 04 00 00          call    8048f06 <phase_1>
8048a65: e8 4a 05 00 00          call    8048fb4 <phase_defused>
8048a6a: c7 44 24 04 40 a0 04    movl    $0x804a040,0x4(%esp)
```

**第三步：**，在反汇编文件中继续查找 `phase_1`，找到其具体定义的位置，如下所示：

```
08048f06 <phase_1>:
8048f06: 55                      push    %ebp
8048f07: 89 e5                   mov     %esp,%ebp
8048f09: 83 ec 18                sub     $0x18,%esp
8048f0c: c7 44 24 04 fc a0 04    movl    $0x804a0fc,0x4(%esp)
8048f13: 08                      nop
8048f14: 8b 45 08                mov     0x8(%ebp),%eax
```

```

8048f17: 89 04 24          mov     %eax,(%esp)
8048f1a: e8 2c 00 00 00    call    8048f4b <strings_not_equal>
8048f1f: 85 c0            test    %eax,%eax
8048f21: 74 05            je      8048f28 <phase_1+0x22>
8048f23: e8 49 01 00 00    call    8049071 <explode_bomb>
8048f28: c9              leave
8048f29: c3              ret
8048f2a: 90              nop
8048f2b: 90              nop
8048f2c: 90              nop
8048f2d: 90              nop
8048f2e: 90              nop
8048f2f: 90              nop

```

从上面的语句中我们可以看出<strings\_not\_equal>所需要的两个变量是存在于%esp 所指向的堆栈存储单元里（strings\_not\_equal 是程序中已经设计好的一个用于判断两个字符串是否相等的函数）。

**第四步：**从前面的 main()函数中，可以进一步找到：

```

8048a58: e8 49 07 00 00    call    80491a6 <read_line>
8048a5d: 89 04 24          mov     %eax,(%esp)

```

这两条语句告诉我们%eax 里存储的是调用 read\_line()函数后返回的结果，也就是用户输入的字符串（首地址），所以可以很容易地推断出和用户输入字符串相比较的字符串的存储地址为 0x804a0fc：

在 phase1 的反汇编代码中有语句：

```

8048f0c: c7 44 24 04 fc a0 04    movl    $0x804a0fc,0x4(%esp)

```

指出了 0x804a0fc,因此我们可以使用 gdb 查看这个地址存储的数据内容。具体过程如下：

**第五步：执行：./bomb/bomblab/src\$ gdb bomb，显示如下：**

```
GNU gdb (GDB) 7.2-ubuntu
```

Copyright (C) 2010 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <<http://gnu.org/licenses/gpl.html>>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying" and "show warranty" for details.

This GDB was configured as "i686-linux-gnu".

For bug reporting instructions, please see:

<<http://www.gnu.org/software/gdb/bugs/>>...

./bomb/bomblab/src/bomb...done.

**(gdb) b main**

Breakpoint 1 at 0x80489a5: file bomb.c, line 45.

**(gdb) r**

Starting program: ./bomb/bomblab/src/bomb

Breakpoint 1, main (argc=1, argv=0xbffff3f4) at bomb.c:45

45 if (argc == 1) {

**(gdb) ni**

0x080489a8 45 if (argc == 1) {

**(gdb) ni**

46 infile = stdin;

**(gdb) ni**

0x080489af 46 infile = stdin;

**(gdb) ni**

0x080489b4 46 infile = stdin;

**(gdb) ni**

67 initialize\_bomb();

**(gdb) ni**

printf (argc=1, argv=0xbffff3f4) at /usr/include/bits/stdio2.h:105

105 return \_\_printf\_chk (\_\_USE\_FORTIFY\_LEVEL - 1, \_\_fmt, \_\_va\_arg\_pack ());

**(gdb) ni**

```
0x08048a38 105    return __printf_chk (__USE_FORTIFY_LEVEL - 1, __fmt, __va_arg_pack ());
```

**(gdb) ni**

```
0x08048a3f 105    return __printf_chk (__USE_FORTIFY_LEVEL - 1, __fmt, __va_arg_pack ());
```

**(gdb) ni**

**Welcome to my fiendish little bomb. You have 6 phases with**

```
0x08048a44 in printf (argc=1, argv=0xbffff3f4)
```

```
    at /usr/include/bits/stdio2.h:105
```

```
105    return __printf_chk (__USE_FORTIFY_LEVEL - 1, __fmt, __va_arg_pack ());
```

**(gdb) ni**

```
0x08048a4c 105    return __printf_chk (__USE_FORTIFY_LEVEL - 1, __fmt, __va_arg_pack ());
```

**(gdb) ni**

```
0x08048a53 105    return __printf_chk (__USE_FORTIFY_LEVEL - 1, __fmt, __va_arg_pack ());
```

**(gdb) ni**

which to blow yourself up. Have a nice day!

```
main (argc=1, argv=0xbffff3f4) at bomb.c:73
```

```
73    input = read_line();                /* Get input                */
```

**(gdb) ni** */\*如果是命令行输入，这里输入你的拆弹字符串\*/*

```
74    phase_1(input);                    /* Run the phase                */
```

**(gdb) x/2s 0x804a0fc** *#查看地址 0x804a0fc 处两个字符串：*

```
0x804a0fc: " I am just a renegade hockey mom "
```

```
0x804a132: ""
```

**(gdb) q** *#退出 gdb*

**Objdump --start-address=0x804a0fc -s bomb #方法 2**

I am just a renegade hockey mom.”就是密码，从而完成了第一个密码的破译。

## 6. 实验结果提交

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 ans.txt，重新命名如下：

班级\_学号.txt，如 CS1201\_U201214795.txt

信安 IS 物联网 IT 计算机 CS 卓越班 ZY ACM 班 ACM

实验报告：Word 文档。

在实验报告中，对你拆除了炸弹的每一道题，用文字详细描述分析求解过程。

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

以班为单位集中打包发送至 130757@qq.com



## LAB3、黑客攻击实验

### 1. 实验简介

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。实验的主要内容扮演黑客对一个可执行程序“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

### 2. 实验数据文件

本次实验中，所有同学的代码包一样，但根据攻击密钥根据自己的学号不一样，访问 <http://211.69.198.69:18213/>，可得到图中的代码包，可以从其他同学处复制该代码包。



本实验的数据包含于一个文件包 buflab-handout.tar 中。下载该文件到本地目录

中，然后利用“tar vxf buflab-handout.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。

### 3. 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 U201414557
```

这里，**-u U201414557** 就是 **bufbomb** 的命令行参数，**U201414557** 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 U201414557 (回车，以下是屏幕上的显示)
```

```
Type string: I love ICS2014 (回车，“I love ICS2014”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，通常会发生下列错误：

```
unix> ./bufbomb -u U201414557 (回车)
```

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

```
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)

tiger123

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：**为了使用方便，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_U201414557.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_U201414557_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_U201414557.txt` 中。然后 `hex2raw` 进行转换，得到 `smoke_U201414557_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_U201414557.txt smoke_U201414557-raw.txt
```

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

#### 方法二：在 `gdb` 中运行 `bufbomb` 时使用：

```
linux> gdb bufbomb
```

```
(gdb) run -u U201414557 < smoke_ U201414557_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\_U201414557.txt：

```
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**

**fizz\_学号.txt**

**bang\_学号.txt**

**boom\_学号.txt**

**nitro\_学号.txt**

如：

**smoke\_ U201414557 .txt**

**fizz\_ U201414557.txt**

**bang\_ U201414557.txt**

**boom\_ U201414557.txt**

**nitro\_ U201414557.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 U201414557.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\_U201414557.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）

## 4. 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

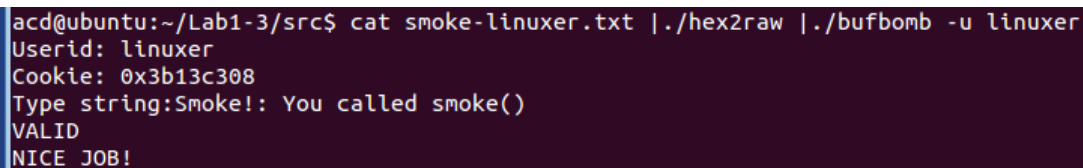
同的缓冲区溢出攻击。实验分 5 个难度级分别命名为 Smoke (level 0)、Fizz (level 1)、Bang (level 2)、Boom (level 3) 和 Nitro (level 4)。

### 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。

### 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 U201414557  
0x5f405c9a
```

“0x5f405c9a”即为根据学号 U201414557 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 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（env）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`）。

## 5. 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。

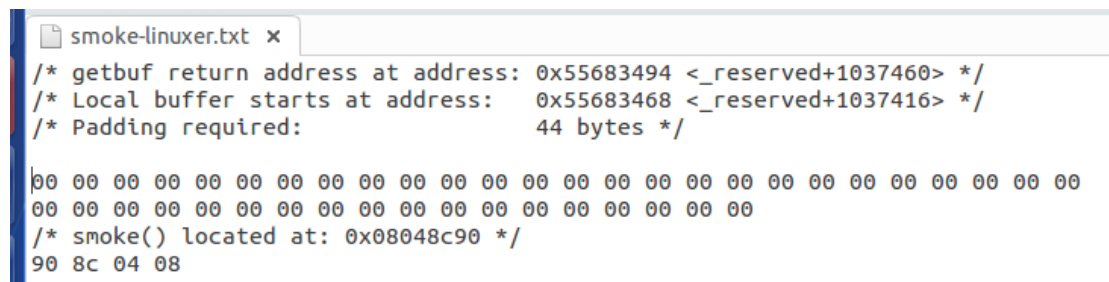
## 6. 实验步骤示例

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：



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\_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
/* 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\_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而代入 bufbomb 中使用。

注: smoke\_U201414557.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\_U201414557.txt 后, 可以用以下命令进行测试:

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

如果正确无误, 则显示如下:

Userid:U201414557

Cookie:0x5f405c9a

Type string:Smoke!: You called smoke()

VALID

NICE JOB!

至此, 任务一成功完成。

```
cat smoke |./hex2raw |./bufbomb -u U201414557
```

global 0x804c218 设置为零

## 7. 实验结果提交

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（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 0~4 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

以班为单位集中打包发送至 130757@qq.com

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%。