華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础实验

实验名称： 二进制程序分析

院　　系 ： 计算机科学与技术

专业班级 ： 计算机图灵班 202301 班

学　　号 ： U202311239

姓　　名 ： 刘星佳

指导教师 ： 王多强

2024 年 10 月 8 日

一、实验目的与要求

通过逆向分析一个二进制程序（称为“二进制炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示各方面知识点的理解，增强反汇编、跟踪、分析、调试等能力。

实验环境：Ubuntu，GCC，GDB 等。

二、实验内容

作为实验目标的二进制炸弹（binary bombs）可执行程序由多个“关”组成。每一个“关”（阶段）要求输入一个特定字符串，如果输入满足程序代码的要求，该阶段即通过，否则程序输出失败。实验的目标是设法得到得出解除尽可能多阶段的字符串。

为了完成二进制炸弹的拆除任务，需要通过反汇编和分析跟踪程序每一阶段的机器代码，从中定位和理解程序的主要执行逻辑，包括关键指令、控制结构和相关数据变量等等，进而推断拆除炸弹所需要的目标字符串。

实验源程序及相关文件：

bomb.c　　主程序

phases.o　各个阶段的目标程序

support.c 完成辅助功能的目标程序

phases.h　support.h　公共头文件

阶段 1：　串比较 phase_1(char *input);

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。

阶段 2：循环 phase_2(char *input);

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。

阶段 3：条件分支 phase_3(char *input);

要求输入一个整数数据，该数据与程序自动生成的 一个数据比较，相等则过关。提示：在自动生成数据时，使用了 switch … case 语句。

阶段 4：递归调用和栈 phase_4(char *input);

要求在一行中输入两个数，第一个数表示在一个有序的数组（或者 binary search tree）中需要搜索到的数，该数是在一定范围之内的；第二个数表示找到搜索数的路径（在树的左边搜索编码为二进制位 0，在树的有边搜索编码为二进制位 1）。

阶段 5：指针和数组访问 phase_5(char *input);

要求在一行中输入一个串，该串与程序自动生成的串相同。在生成串和比较串时，使用了数组和指针。

阶段 6：链表、结构、指针的访问 phase_6(char *input);

要求在一行中输入 6 个数，这 6 个数是一个链表中结点的顺序号（从 1 到 6）。按照输

入的顺序号，将对应链表结点中的值形成一个数组。若该数组是按照降序排列的，则过关。

三、实验记录及问题回答

拆除所有炸弹的密码字符串 ans.txt 如下：

Brownie, you are doing a heck of a job.
1 2 4 7 11 16
3 -814
36 3 DrEvil
ioapeg
5 6 3 4 2 1
40

1. phase_1

```
14e2:   50                      push   %eax
14e3:   ff 74 24 1c             push   0×1c(%esp)
14e7:   e8 6a 05 00 00          call   1a56 <strings_not_equal>
14ec:   83 c4 10                add    $0×10,%esp
14ef:   85 c0                   test   %eax,%eax
14f1:   75 05                   jne    14f8 <phase_1+0×2b>
14f3:   83 c4 08                add    $0×8,%esp
14f6:   5b                      pop    %ebx
14f7:   c3                      ret
14f8:   e8 71 06 00 00          call   1b6e <explode_bomb>
14fd:   eb f4                   jmp    14f3 <phase_1+0×26>
```

phase_1 的核心主要是上面的汇编代码，调用了 strings_not_equal 函数判断两个字符串是否相等。通过 gdb 调试工具查看压入栈中的两个函数参数的值，发现其分别是输入的字符串和一个常量字符串“Brownie, you are doing a heck of a job.”调用完成后使用 test 对返回值进行判断，如果返回值为 1 则直接触发炸弹，此时字符串不相等。所以只要让输入的字符串与该常量字符串相等即可。

2. phase_2

```
1520:   50                      push   %eax
1521:   ff 74 24 3c             push   0×3c(%esp)
1525:   e8 79 06 00 00          call   1ba3 <read_six_numbers
152a:   83 c4 10                add    $0×10,%esp
152d:   83 7c 24 04 00          cmpl   $0×0,0×4(%esp)
1532:   78 0b                   js     153f <phase_2+0×40>
1534:   be 01 00 00 00          mov    $0×1,%esi
1539:   8d 7c 24 04             lea    0×4(%esp),%edi
153d:   eb 0f                   jmp    154e <phase_2+0×4f>
153f:   e8 2a 06 00 00          call   1b6e <explode_bomb>
1544:   eb ee                   jmp    1534 <phase_2+0×35>
```

phase_2 先是调用 read_six_numbers 函数，说明这个炸弹的密码应当为六个数字，设为 a1~a6。函数返回后，用 gdb 工具查找对应地址发现输入的六个整数存储在栈指针 esp+4 之后的连续 24 个字节处。接着程序判断如果 a1<0 就直接触发炸弹，说明首先 a1>=0。

```
1546:   83 c6 01              add    $0×1,%esi
1549:   83 fe 06              cmp    $0×6,%esi
154c:   74 12                 je     1560 <phase_2+0×61>
154e:   89 f0                 mov    %esi,%eax
1550:   03 44 b7 fc           add    -0×4(%edi,%esi,4),%eax
1554:   39 04 b7              cmp    %eax,(%edi,%esi,4)
1557:   74 ed                 je     1546 <phase_2+0×47>
1559:   e8 10 06 00 00        call   1b6e <explode_bomb>
```

上述判断之后跳转至一个循环结构，该循环执行 6 次，寄存器%esi 从 1 递增到 5（记为 i），将%eax 寄存器的值设为 i+ai，然后与 a(i+1)判断是否相等，如果不相等就会执行 explode_bomb 发生爆炸。

综上所述，该炸弹的密码是一个长度为 6 的数列，满足首项>=0 且 a(i+1)=i+ai，即密码不唯一，只要是 x,x+1,x+3,x+6,x+10,x+15(x>=0)的形式即可。

3. phase_3

```
1594:   8d 44 24 08           lea    0×8(%esp),%eax
1598:   50                    push   %eax
1599:   8d 44 24 08           lea    0×8(%esp),%eax
159d:   50                    push   %eax
159e:   8d 83 77 d3 ff ff     lea    -0×2c89(%ebx),%eax
15a4:   50                    push   %eax
15a5:   ff 74 24 2c           push   0×2c(%esp)
15a9:   e8 92 fb ff ff        call   1140 <__isoc99_sscanf@plt>
```

该阶段首先是执行 sscanf 函数进行读入，通过 gdb 查看压栈的元素获得 sscanf 的第二个参数为“%d %d”，表明第三个炸弹的密钥是以空格隔开的两个整数。后面还判断了一下 sscanf 的返回值是否>=2（至少输入了两个数）。

```
15b6:   83 7c 24 04 07        cmpl   $0×7,0×4(%esp)
15bb:   0f 87 93 00 00 00     ja     1654 <.L37+0×7>
15c1:   8b 44 24 04           mov    0×4(%esp),%eax
15c5:   89 da                 mov    %ebx,%edx
15c7:   03 94 83 38 d2 ff ff  add    -0×2dc8(%ebx,%eax,4),%edx
15ce:   ff e2                 jmp    *%edx
15d0:   e8 99 05 00 00        call   1b6e <explode_bomb>
15d5:   eb df                 jmp    15b6 <phase_3+0×3d>
```

接着程序将读入的第一个数与 7比较，如果大于 7 就会跳转到 explode_bomb，因此第一个数<=7。将第一个数赋值给 eax 寄存器后，后面是一个类似于 switch-case 的结构，1～7 正好对应着 7 个标签 L31~L37。

```
00001623 <.L31>:
    1623:   b8 00 00 00 00          mov    $0×0,%eax
    1628:   eb b2                   jmp    15dc <.L26+0×5>

0000162a <.L32>:
    162a:   b8 00 00 00 00          mov    $0×0,%eax
    162f:   eb b0                   jmp    15e1 <.L26+0×a>

00001631 <.L33>:
    1631:   b8 00 00 00 00          mov    $0×0,%eax
    1636:   eb ae                   jmp    15e6 <.L26+0×f>

00001638 <.L34>:
    1638:   b8 00 00 00 00          mov    $0×0,%eax
    163d:   eb ac                   jmp    15eb <.L26+0×14>

0000163f <.L35>:
    163f:   b8 00 00 00 00          mov    $0×0,%eax
    1644:   eb aa                   jmp    15f0 <.L26+0×19>

00001646 <.L36>:
    1646:   b8 00 00 00 00          mov    $0×0,%eax
    164b:   eb a8                   jmp    15f5 <.L26+0×1e>

0000164d <.L37>:
    164d:   b8 00 00 00 00          mov    $0×0,%eax
    1652:   eb a6                   jmp    15fa <.L26+0×23>
    1654:   e8 15 05 00 00          call   1b6e <explode_bomb>
    1659:   b8 00 00 00 00          mov    $0×0,%eax
    165e:   eb 9f                   jmp    15ff <.L26+0×28>
    1660:   e8 1b 13 00 00          call   2980 <__stack_chk_fail_local>
```

从图中可以看出，七个标签都会先跳转至 L26 标签，只是位置不同。

```
000015d7 <.L26>:
    15d7:   b8 84 00 00 00          mov    $0×84,%eax
    15dc:   2d e0 03 00 00          sub    $0×3e0,%eax
    15e1:   05 e2 03 00 00          add    $0×3e2,%eax
    15e6:   2d 2e 03 00 00          sub    $0×32e,%eax
    15eb:   05 2e 03 00 00          add    $0×32e,%eax
    15f0:   2d 2e 03 00 00          sub    $0×32e,%eax
    15f5:   05 2e 03 00 00          add    $0×32e,%eax
    15fa:   2d 2e 03 00 00          sub    $0×32e,%eax
    15ff:   83 7c 24 04 05          cmpl   $0×5,0×4(%esp)
    1604:   7f 06                   jg     160c <.L26+0×35>
    1606:   39 44 24 08             cmp    %eax,0×8(%esp)
    160a:   74 05                   je     1611 <.L26+0×3a>
    160c:   e8 5d 05 00 00          call   1b6e <explode_bomb>
    1611:   8b 44 24 0c             mov    0×c(%esp),%eax
    1615:   65 2b 05 14 00 00 00    sub    %gs:0×14,%eax
    161c:   75 42                   jne    1660 <.L37+0×13>
    161e:   83 c4 18                add    $0×18,%esp
    1621:   5b                      pop    %ebx
    1622:   c3                      ret
```

以第 3 个标签为例，跳转到 L26 标签之后，eax 寄存器变为 0-0x32e=-814。后面将标签数按 5 进行分类，小于 5 的部分会把第二个数和 eax 寄存器的值比较，不同的话就会进入 explode_bomb，而相同能顺利拆除炸弹，如 3 -814 是其中一组可行的密钥。

4. phase_4

phase_4 的主要部分也是先以 sscanf 开始，发现读入格式仍然是“%d %d”，说明 phase_4 的密钥也是两个整数（设为 a,b）。

```
    16f7:   ff 74 24 0c             push   0×c(%esp)
    16fb:   6a 05                   push   $0×5
    16fd:   e8 63 ff ff ff          call   1665 <func4>
    1702:   83 c4 10                add    $0×10,%esp
    1705:   39 44 24 08             cmp    %eax,0×8(%esp)
    1709:   75 12                   jne    171d <phase_4+0×77>
```

后面可以看出程序主体是先调用 func4(5,b)，把返回值和第一个输入值 a 相比较，如果不同就会触发 explode_bomb。因此我们只需要探究 func(5,b)的值即可。

```
1668:   8b 5c 24 10             mov    0×10(%esp),%ebx
166c:   8b 7c 24 14             mov    0×14(%esp),%edi
1670:   b8 00 00 00 00          mov    $0×0,%eax
1675:   85 db                   test   %ebx,%ebx
1677:   7e 29                   jle    16a2 <func4+0×3d>
1679:   89 f8                   mov    %edi,%eax
167b:   83 fb 01                cmp    $0×1,%ebx
167e:   74 22                   je     16a2 <func4+0×3d>
1680:   83 ec 08                sub    $0×8,%esp
1683:   57                      push   %edi
1684:   8d 43 ff                lea    -0×1(%ebx),%eax
1687:   50                      push   %eax
1688:   e8 d8 ff ff ff          call   1665 <func4>
168d:   83 c4 08                add    $0×8,%esp
1690:   8d 34 38                lea    (%eax,%edi,1),%esi
1693:   57                      push   %edi
1694:   83 eb 02                sub    $0×2,%ebx
1697:   53                      push   %ebx
1698:   e8 c8 ff ff ff          call   1665 <func4>
169d:   83 c4 10                add    $0×10,%esp
16a0:   01 f0                   add    %esi,%eax
```

func4(x,y)首先判断 x 的值是否<=0，如果是的话就直接返回 eax 的值即 0。接着将 eax 的初值设置为 y，比较 x 和 1 的大小关系。如果 x=1 也会直接返回，返回值为 y。x>1 时开始压栈说明要进一步递归下去，eax 的值变为 x-1，即递归 func(x-1,y)。call 之后又开始压栈，说明有二次递归，同理可以看出是 func(x-2,y)。1690 代码处 esi 寄存器被赋值了 func(x-1,y)+y，16a0 处又将 eax 变为 esi 的值加上 func(x-2,y)，因此我们可以得到递推式 func(x,y)=func(x-1,y)+func(x-2,y)+y，初值 func(1,y)=y,fun(0,y)=0。因此 func(2,y)=2y,func(3,y)=4y,func(4,y)=7y,func(5,y)=12y。因此密钥中的 a,b 满足 a=12b 即可，如 36,3 即为一组合法的密钥。

5. phase_5

phase_5 先对一个字符串调用了 string_length 函数，通过 gdb 查看传入参数的值得知该函数计算我们输入的字符串的长度，如果长度不为 6 就直接跳转到爆炸函数，说明 phase_5 的输入应当是一个长度为 6 的字符串。

```
1757:   b8 00 00 00 00          mov    $0×0,%eax
175c:   8d 8b 58 d2 ff ff       lea    -0×2da8(%ebx),%ecx
1762:   0f b6 14 06             movzbl (%esi,%eax,1),%edx
1766:   83 e2 0f                and    $0×f,%edx
1769:   0f b6 14 11             movzbl (%ecx,%edx,1),%edx
176d:   88 54 04 05             mov    %dl,0×5(%esp,%eax,1)
1771:   83 c0 01                add    $0×1,%eax
1774:   83 f8 06                cmp    $0×6,%eax
1777:   75 e9                   jne    1762 <phase_5+0×39>
```

接着是一个循环结构，发现寄存器 esi 中存储着输入字符串的地址，也就是循环会依次将字符串中的字符取出，将其 ASCII 码对 16 取模获得一个 0～15 的数存在寄存器 edx 中。然后似乎是以 edx 为下标取得某个数组中的值。

```
(gdb) x/s $ecx
0x565581bc <array.0>:   "maduiersnfotvbyl
```

使用 gdb 工具在 1762 设置断点查看寄存器 ecx，发现其恰好是一个长度为 16 的字符串的首地址。也就是输入的字符串的字符转化下标后在这个字符串常量中取值，最后又产生了一个新的字符串。

```
(gdb) x/s $eax
0x56558192:     "flames"
```

继续看汇编代码，发现到了后面又调用了 string_not_equal 函数，同样使用 gdb 在参数入栈时查看，发现一个字符串是刚刚的重组字符串，还有一个是字符串常量“flames”。也就是我们需要构造字符串使重组字符串为“flames”即能拆除炸弹。通过反推，我们找到这六个字符在 array.0 中的位置分别为 9，15，1，0，5，7，假设输入的字符串是前 16 个小写字母，则这分别表示了我们解密字符串在字母表中的顺序，即获得最终答案 ioapeg。

6. phase_6

phase_6 开始时调用 read_six_numbers 函数表明拆弹密钥为六个整数。

```
1805:   83 c6 01              add    $0x1,%esi
1808:   83 fe 06              cmp    $0x6,%esi
180b:   74 0f                 je     181c <phase_6+0x5d>
180d:   8b 44 b5 00           mov    0x0(%ebp,%esi,4),%eax
1811:   39 07                 cmp    %eax,(%edi)
1813:   75 f0                 jne    1805 <phase_6+0x46>
1815:   e8 54 03 00 00        call   1b6e <explode_bomb>
181a:   eb e9                 jmp    1805 <phase_6+0x46>
181c:   83 44 24 08 04        addl   $0x4,0x8(%esp)
1821:   8b 44 24 08           mov    0x8(%esp),%eax
1825:   89 c7                 mov    %eax,%edi
1827:   8b 00                 mov    (%eax),%eax
1829:   89 44 24 0c           mov    %eax,0xc(%esp)
182d:   83 e8 01              sub    $0x1,%eax
1830:   83 f8 05              cmp    $0x5,%eax
1833:   77 c9                 ja     17fe <phase_6+0x3f>
1835:   83 44 24 04 01        addl   $0x1,0x4(%esp)
183a:   8b 74 24 04           mov    0x4(%esp),%esi
183e:   83 fe 05              cmp    $0x5,%esi
1841:   7e ca                 jle    180d <phase_6+0x4e>
```

读入后程序进入一个循环结构，遍历输入的六个数，如果这个数-1>5 就会触发炸弹，所以这六个数都不能超过 6。进行完上述判断之后会进入第二重循环，该循环作用是判断这六个数是否互不相同，否则触发炸弹。因此可以猜测输入是一个 1～6 的排列。

```
1843:   be 00 00 00 00          mov    $0×0,%esi
1848:   89 f7                   mov    %esi,%edi
184a:   8b 4c b4 1c             mov    0×1c(%esp,%esi,4),%ecx
184e:   b8 01 00 00 00          mov    $0×1,%eax
1853:   8d 93 68 01 00 00       lea    0×168(%ebx),%edx
1859:   83 f9 01                cmp    $0×1,%ecx
185c:   7e 0a                   jle    1868 <phase_6+0×a9>
185e:   8b 52 08                mov    0×8(%edx),%edx
1861:   83 c0 01                add    $0×1,%eax
1864:   39 c8                   cmp    %ecx,%eax
1866:   75 f6                   jne    185e <phase_6+0×9f>
1868:   89 54 bc 34             mov    %edx,0×34(%esp,%edi,4)
186c:   83 c6 01                add    $0×1,%esi
186f:   83 fe 06                cmp    $0×6,%esi
1872:   75 d4                   jne    1848 <phase_6+0×89>
```

紧接着又是一个循环结构，中间有一个载入 edx 的操作，通过 gdb 查看。

```
(gdb) x/20wx $edx
0x5655b0cc <node1>:     0x0000004a      0x00000001      0x5655b0d8      0x0000006f
0x5655b0dc <node2+4>:   0x00000002      0x5655b0e4      0x000001a7      0x00000003
0x5655b0ec <node3+8>:   0x5655b0f0      0x0000014a      0x00000004      0x5655b0fc
0x5655b0fc <node5>:     0x0000030f      0x00000005      0x5655b068      0x00000000
0x5655b10c:     0x00000000      0x00000000      0x00000000      0x00000000
```

发现有很整齐的数据，初步可以判断每个 node 占 4*3 字节。继续阅读汇编，发现这应当是链表的节点，第一个位置存储节点的值，第二个位置存储编号，第三个位置存储下一个节点的首地址。1853～1866 处的代码是根据寄存器 ecx 的值依次将链表对应的第 ecx 个节点存到连续的地址段中，而 ecx 依次是读入的六个数。相当于按我们输入的顺序将链表中的节点进行了重新排序并存储到连续的地址段。

```
1874:   8b 74 24 34             mov    0×34(%esp),%esi
1878:   8b 44 24 38             mov    0×38(%esp),%eax
187c:   89 46 08                mov    %eax,0×8(%esi)
187f:   8b 54 24 3c             mov    0×3c(%esp),%edx
1883:   89 50 08                mov    %edx,0×8(%eax)
1886:   8b 44 24 40             mov    0×40(%esp),%eax
188a:   89 42 08                mov    %eax,0×8(%edx)
188d:   8b 54 24 44             mov    0×44(%esp),%edx
1891:   89 50 08                mov    %edx,0×8(%eax)
1894:   8b 44 24 48             mov    0×48(%esp),%eax
1898:   89 42 08                mov    %eax,0×8(%edx)
189b:   c7 40 08 00 00 00 00    movl   $0×0,0×8(%eax)
18a2:   bf 05 00 00 00          mov    $0×5,%edi
```

这段连续的 mov 指令也很好分析，每个 0x8(...)表示修改了相应节点的 next，可以看出是将刚刚重排的节点重新连接成一个链表。

```
18a9:   8b 76 08              mov    0x8(%esi),%esi
18ac:   83 ef 01              sub    $0x1,%edi
18af:   74 10                 je     18c1 <phase_6+0x102>
18b1:   8b 46 08              mov    0x8(%esi),%eax
18b4:   8b 00                 mov    (%eax),%eax
18b6:   39 06                 cmp    %eax,(%esi)
18b8:   7d ef                 jge    18a9 <phase_6+0xea>
18ba:   e8 af 02 00 00        call   1b6e <explode_bomb>
18bf:   eb e8                 jmp    18a9 <phase_6+0xea>
```

最后的一个循环结构，当前节点为存在 esi 中，下一个节点存在 eax 中，然后比较大小，如果当前节点值大于等于下一个节点就进行下一次判断，否则炸弹爆炸。至此真相大白，我们需要保证重排后的节点权值从大到小排序。

```
(gdb) x/3wx $edx
0x5655b0cc <node1>:     0x0000004a      0x00000001      0x5655b0d8
(gdb) x/3wx 0x5655b0d8
0x5655b0d8 <node2>:     0x0000006f      0x00000002      0x5655b0e4
(gdb) x/3wx 0x5655b0e4
0x5655b0e4 <node3>:     0x000001a7      0x00000003      0x5655b0f0
(gdb) x/3wx 0x5655b0f0
0x5655b0f0 <node4>:     0x0000014a      0x00000004      0x5655b0fc
(gdb) x/3wx 0x5655b0fc
0x5655b0fc <node5>:     0x0000030f      0x00000005      0x5655b068
(gdb) x/3wx 0x5655b068\
0x5655b068 <node6>:     0x00000257      0x00000006      0x00000000
```

因此我们需要先从最开始发现第一个节点的位置起，用 gdb 看相应地址的 node 权值，将所有节点的权值即编号都查找出来。为了从大到小排序，我们输入的编号顺序应当为 5 6 3 4 2 1.

7. 隐藏关

发现程序中有 fun7 这一函数还没被调用过，其在 secret_phase 中调用，而 secret_phase 在 phase_defused 中调用。

在 phase_defused 中有这样的跳转判断：

```
1d27:   83 bb 2c 04 00 00 06    cmpl   $0x6,0x42c(%ebx)
1d2e:   74 16                   je     1d46 <phase_defused+0x3a>
```

通过 gdb 访问 0x42c(%ebx)，发现这是字段<num_input_strings>的地址，这个变量存储着目前已经读入了几行（也就是现在是现在完成到了第几关），发现如果第 6 关完成，就会跳转到一个特殊的代码段。

紧接着，代码调用了 sscanf 函数，用 gdb 查看压入栈中的参数发现读入格式为"%d %d %s"，读入来源是 phase_4 的读入，这启发我们要开启隐藏关需要在 phase_4 的两个数后再加上一个字符串。

```
1d87:   83 ec 08              sub    $0x8,%esp
1d8a:   8d 83 da d3 ff ff     lea    -0x2c26(%ebx),%eax
1d90:   50                    push   %eax
1d91:   8d 44 24 18           lea    0x18(%esp),%eax
1d95:   50                    push   %eax
1d96:   e8 bb fc ff ff        call   1a56 <strings_not_equal>
```

接着代码判断了 phase_4 后添加的字符串和某个常量字符串是否相同，相同才会执行隐藏关相关代码。用 gdb 查看得知该常量字符串为“DrEvil”，也就是在 phase_4 后应当添加的字符串。

进入 secret_phase，程序先用 read_line 读入一行，并调用了 strtol 将字符串转换为整数，意味着隐藏关需要我们输入一个整数。

```
1949:   e8 62 f8 ff ff        call   11b0 <strtol@plt>
194e:   89 c6                 mov    %eax,%esi
1950:   8d 40 ff              lea    -0x1(%eax),%eax
1953:   83 c4 10              add    $0x10,%esp
1956:   3d e8 03 00 00        cmp    $0x3e8,%eax
195b:   77 32                 ja     198f <secret_phase+0x63>
```

要求输入的整数必须不超过 0x3e8。

```
195d:   83 ec 08              sub    $0x8,%esp
1960:   56                    push   %esi
1961:   8d 83 14 01 00 00     lea    0x114(%ebx),%eax
1967:   50                    push   %eax
1968:   e8 6e ff ff ff        call   18db <fun7>
```

传入 fun7 的第一个参数是刚刚读入的整数，用 gdb 得知第二个参数是某个变量 n1 的地址。

接下来看 int fun7(void *xp,int y)的运行逻辑（记 x=*(int *)xp）

```
18ed:   39 cb                 cmp    %ecx,%ebx
18ef:   7f 0c                 jg     18fd <fun7+0x22>
18f1:   b8 00 00 00 00        mov    $0x0,%eax
18f6:   75 18                 jne    1910 <fun7+0x35>
18f8:   83 c4 08              add    $0x8,%esp
18fb:   5b                    pop    %ebx
18fc:   c3                    ret
```

如果 x=y 就返回 0，否则根据 x 和 y 的大小关系跳转到不同分支去执行。

如果 x<y：

```
1910:   83 ec 08              sub    $0x8,%esp
1913:   51                    push   %ecx
1914:   ff 72 08              push   0x8(%edx)
1917:   e8 bf ff ff ff        call   18db <fun7>
191c:   83 c4 10              add    $0x10,%esp
191f:   8d 44 00 01           lea    0x1(%eax,%eax,1),%eax
1923:   eb d3                 jmp    18f8 <fun7+0x1d>
```

调用 fun7(xp+8,y),然后把返回值*2+1 后作为自身的返回值。

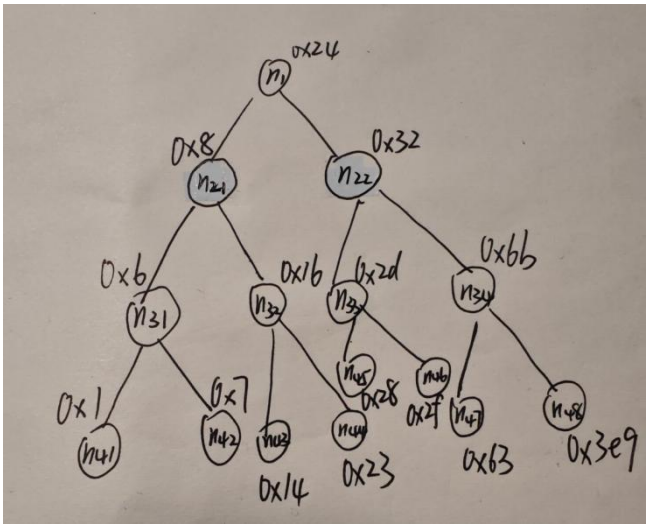
当 x>y 时也类似，调用 fun7(xp+4,y)，然后把返回值*2 后作为自身的返回值。

用 gdb 查看最开始传入 fun7 的第一个地址参数及附近的内存：

```
(gdb) x/20wx $edx
0x5655b078 <n1>:        0x00000024      0x5655b084      0x5655b090      0x00000008
0x5655b088 <n21+4>:     0x5655b0b4      0x5655b09c      0x00000032      0x5655b0a8
0x5655b098 <n22+8>:     0x5655b0c0      0x00000016      0x5655b044      0x5655b02c
0x5655b0a8 <n33>:       0x0000002d      0x5655b008      0x5655b050      0x00000006
0x5655b0b8 <n31+4>:     0x5655b014      0x5655b038      0x0000006b      0x5655b020
```

这和 phase_6 的结构很像，应该是某种数据结构。根据标签的提示又可以看出这个数据结构每个节点由 12 个字节组成，包括节点的值和两个指针，可以猜想这个数据结构是二叉树。

从 n1 开始递归地根据指针值用 gdb 去访问，依次把整棵二叉树的结构和每个节点的值分析出来：



可以发现这棵二叉树同时还是一棵二叉查找树，因此 fun7 的功能就是通过当前节点 xp 的值 x 与固定值 y 比较，从而确认节点值为 y 的点在左子树还是右子树内，并继续递归查询。

最终 fun7 返回值的含义可以理解为从根节点到值为 y 的节点的路径上，向左儿子走记为 0，向右儿子走记为 1，形成到一个二进制串，翻转一下得到的二进制串转化为整数即为返回值。

得知了 fun7 的功能后，再回到 secret_phase 处看看它利用返回值做了什么判断：

```
1968:   e8 6e ff ff ff          call   18db <fun7>
196d:   83 c4 10                add    $0x10,%esp
1970:   83 f8 01                cmp    $0x1,%eax
1973:   75 21                   jne    1996 <secret_phase+0x6a>
```

如果返回值不为 1 就会跳转到爆炸，那么要想返回值为 1（二进制的 001），图中的节点 n45（0x28）就是满足条件的。这个节点的值也就是隐藏关的密钥，转换成十进制即为 40。

四、体会

在此次实验中，通过拆解二进制炸弹的各个阶段，我对程序的机器级表示有了更深入的理解。这不仅帮助我巩固了反汇编与调试工具（如 GDB）的使用技巧，还增强了我对复杂程序逻辑的分析和解决能力。

在每一个阶段的拆解过程中，我学会了如何有效地利用 GDB 工具逐步跟踪程序的运行，并通过观察汇编代码来推断出程序的逻辑与输入要求。通过分析每个阶段的逻辑，我体会到机器代码虽低级但非常严谨，精确的每一步运算和控制流都由底层指令来完成。这让我更加深刻理解了高级语言编写的程序最终是如何通过汇编语言指令来实现的。

此外，实验让我深刻体会到细心与耐心在逆向工程中的重要性。二进制炸弹的拆解需要非常细致的观察和分析，稍有不慎就可能导致误解程序逻辑，最终引发错误的结果。每个关卡都需要在理解基础上制定有效的解决策略，而不仅仅是盲目地进行调试与测试。

这次实验不仅加强了我对计算机系统底层机制的理解，也让我积累了更多的实际操作经验。通过这个实验，我对如何使用反汇编工具、调试工具，如何分析复杂程序的底层实现有了更清晰的认识，这为我日后处理更复杂的系统和应用程序奠定了坚实的基础。