哈爾濱工業大學

实验报告

实 验（三）

题　　目 Binary Bomb

二进制炸弹

专　　业 计算学部

学　　号 1190200910

班　　级 1903012

学　　生 严幸

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2021-4-21

计算机科学与技术学院

目 录

第 1 章 实验基本信息 - 3 -

1.1 实验目的 - 3 -
1.2 实验环境与工具 - 3 -
1.2.1 硬件环境 - 3 -
1.2.2 软件环境 - 3 -
1.2.3 开发工具 - 3 -
1.3 实验预习 - 3 -

第 2 章 实验环境建立 - 5 -

2.1 UBUNTU 下 CODEBLOCKS 反汇编（10 分） - 5 -
2.2 UBUNTU 下 EDB 运行环境建立（10 分） - 5 -

第 3 章 各阶段炸弹破解与分析 - 7 -

3.1 阶段 1 的破解与分析 - 7 -
3.2 阶段 2 的破解与分析 - 7 -
3.3 阶段 3 的破解与分析 - 10 -
3.4 阶段 4 的破解与分析 - 10 -
3.5 阶段 5 的破解与分析 - 13 -
3.6 阶段 6 的破解与分析 - 14 -
3.7 阶段 7 的破解与分析(隐藏阶段) - 17 -

第 4 章 总结 - 23 -

4.1 请总结本次实验的收获 - 23 -
4.2 请给出对本次实验内容的建议 - 23 -

参考文献 - 24 -

第 1 章 实验基本信息

1.1 实验目的

- 熟练掌握计算机系统的 ISA 指令系统与寻址方式
- 熟练掌握 Linux 下调试器的反汇编调试跟踪分析机器语言的方法
- 增强对程序机器级表示、汇编语言、调试器和逆向工程等的理解

1.2 实验环境与工具

1.2.1 硬件环境

- X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

- Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；
- GDB/OBJDUMP；EDB；KDD 等

1.2.3 开发工具

Visual Studio 2019

1.3 实验预习

- 上实验课前，必须认真预习实验指导书（PPT 或 PDF）
- 了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。
- 请写出 C 语言下包含字符串比较、循环、分支（含 switch）、函数调用、递归、指针、结构、链表等的例子程序 sample.c。
- 生成执行程序 sample.out。
- 用 gcc －S 或 CodeBlocks 或 GDB 或 OBJDUMP 等，反汇编，比较。

- 列出每一部分的 C 语言对应的汇编语言。
- 修改编译选项-O (缺省 2)、O0、O1、O3、Og、-m32/m64。再次查看生成的汇编语言与原来的区别。
- 注意 O1 之后缺省无栈帧，RBP 为普通寄存器。用 -fno-omit-frame-pointer 加上栈指针。
- GDB 命令详解 –tui 模式 ^XA 切换 layout 改变等等
- 有目的地学习: 看 VS 的功能，GDB 命令用什么？

第 2 章 实验环境建立

2.1 Ubuntu 下 CodeBlocks 反汇编（10 分）

CodeBlocks 运行 hellolinux.c。反汇编查看 printf 函数的实现。

要求：C、ASM、内存(显示 hello 等内容)、堆栈（call printf 前）、寄存器同时在一个窗口。

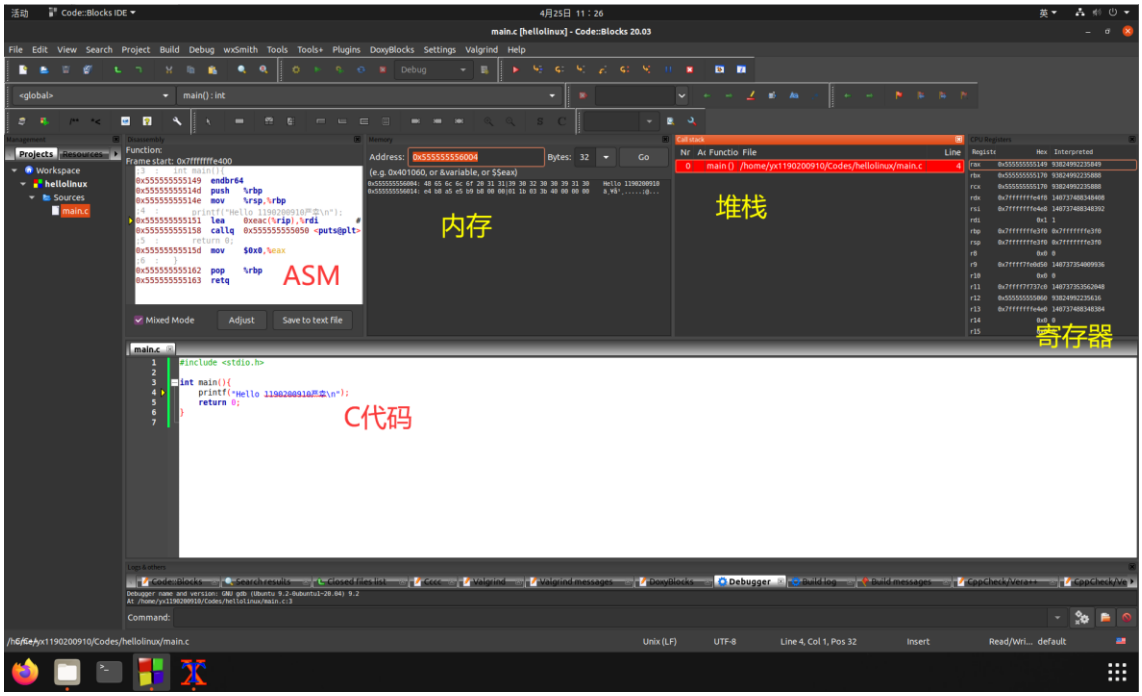


图 2-1 Ubuntu 下 CodeBlocks 反汇编截图

2.2 Ubuntu 下 EDB 运行环境建立（10 分）

用 EDB 调试 hellolinux.c 的执行文件，截图，要求同 2.1

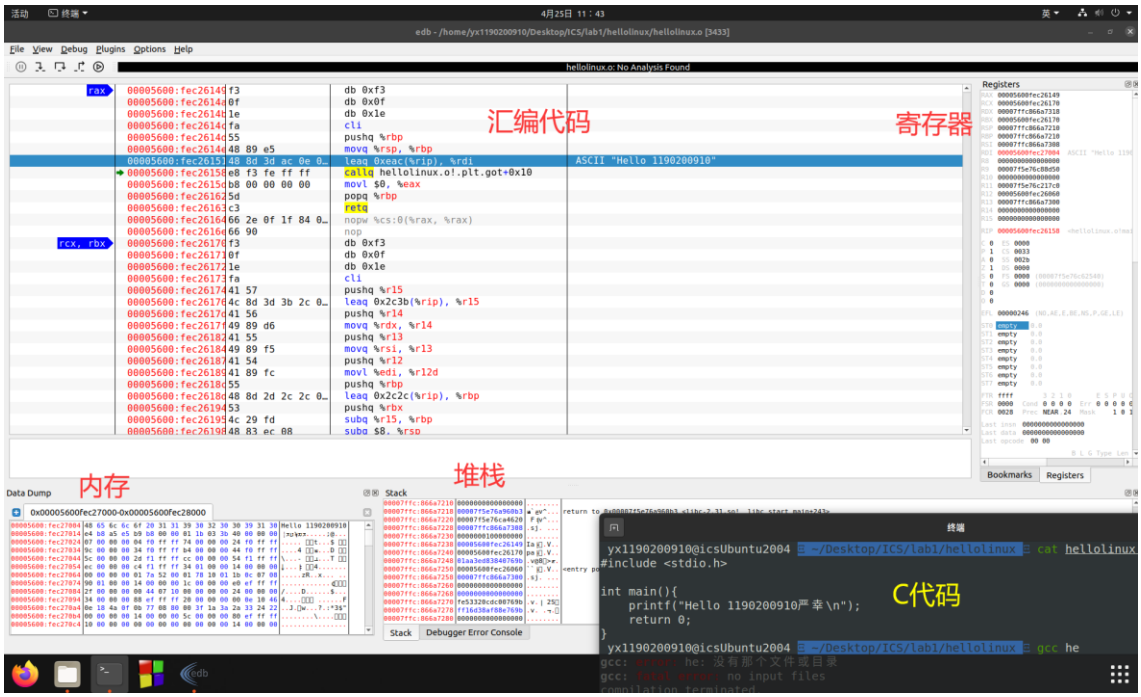


图 2-2　Ubuntu 下 EDB 截图

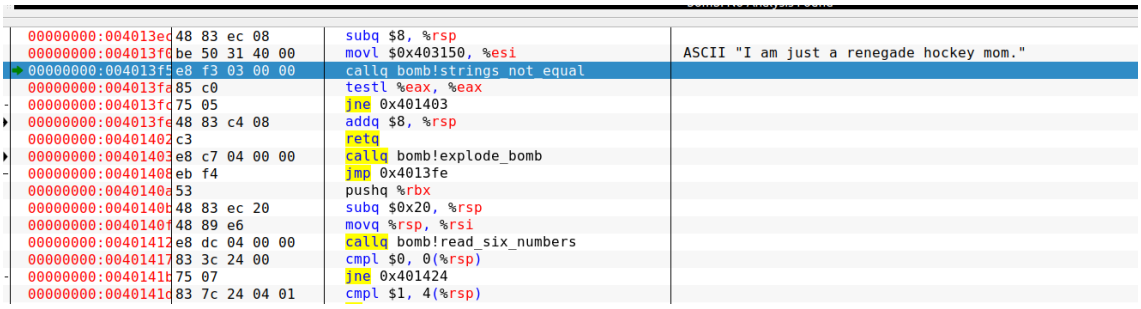
第 3 章 各阶段炸弹破解与分析

每阶段 40 分，密码 20 分，分析 20 分，总分不超过 80 分

3.1 阶段 1 的破解与分析

密码如下：I am just a renegade hockey mom.

破解过程：

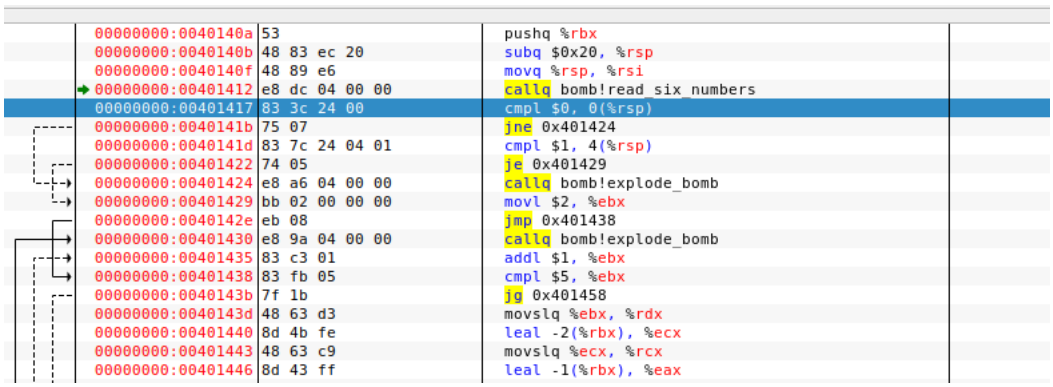


进入第一关 phase_1 过程后，可以看到程序将用户输入的字符串与程序内部保存的一个字符串常量进行了对比，如果两串不相等则执行爆炸函数，因此这里要求输入的字符串即为程序保存的字符串 I am just a renegade hockey mom.

3.2 阶段 2 的破解与分析

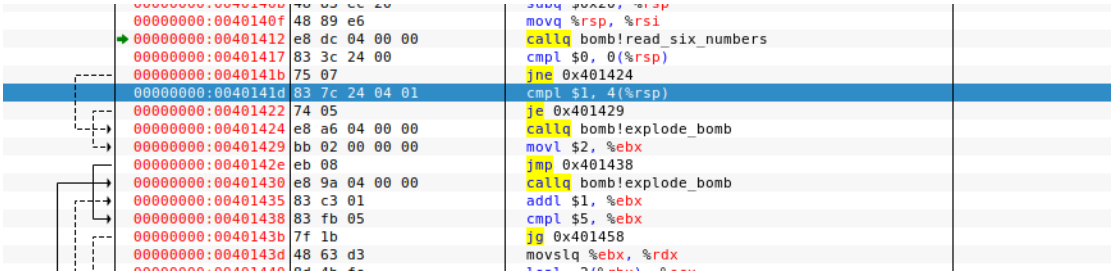
密码如下：0 1 1 2 3 5

破解过程：

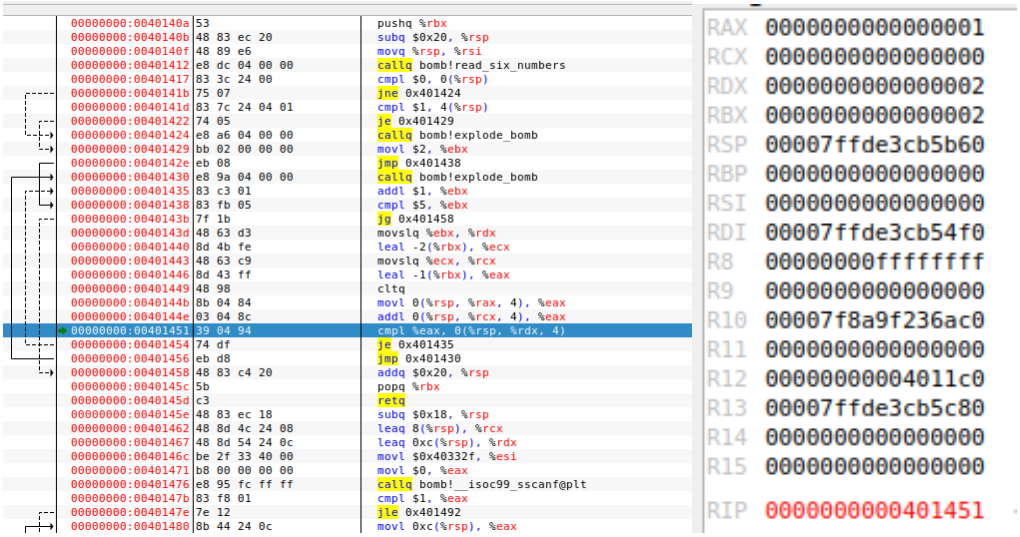


通过 EDB 动态调试。进入第二关函数后，可以看到调用了 read_six_numbers 函数，在这个函数内用格式化字符串“%d %d %d %d %d %d”调用了 scanf，可以推测这里是要求输入 6 个整数，分别存在 rsp 偏移 0x0, 0x4, 0x8, 0xC, 0x10, 0x14 的位置。

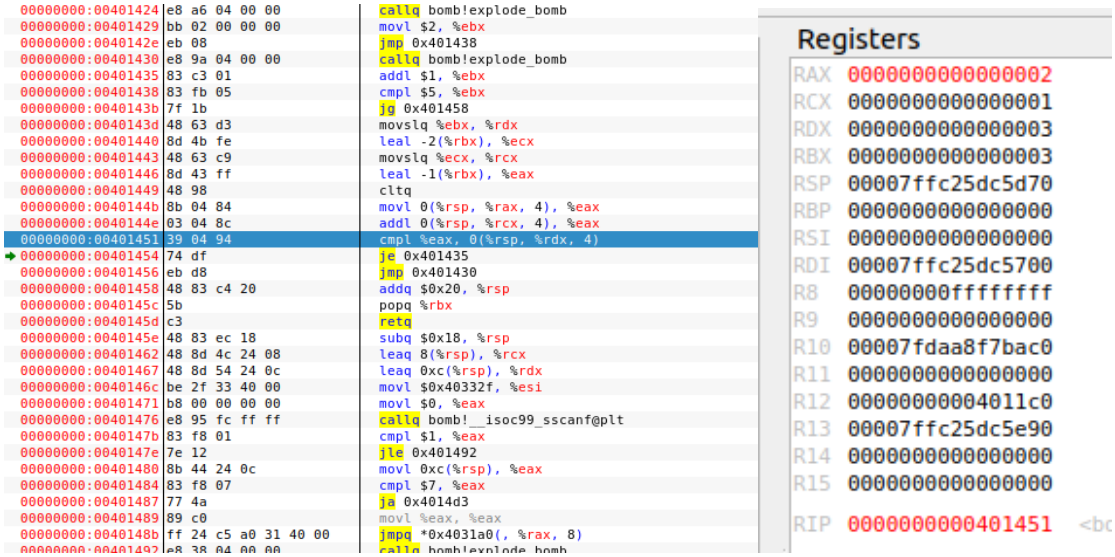
从上图可知，0x401424 地址对应爆炸函数，程序对比了输入的第 1 个数字和 1，如果第 1 个数字不为 0 则炸弹爆炸，因此输入的第一个数字应为 0。



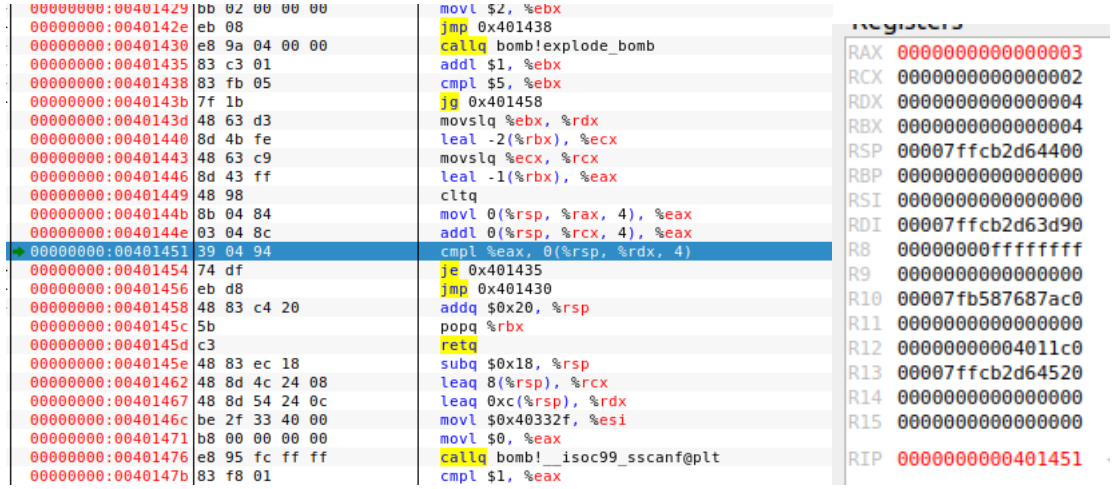
在比较第 2 个数字时，为了不让代码执行 0x401424 的炸弹爆炸函数，必须让第二个数等于 1 从而跳转到 0x401429。



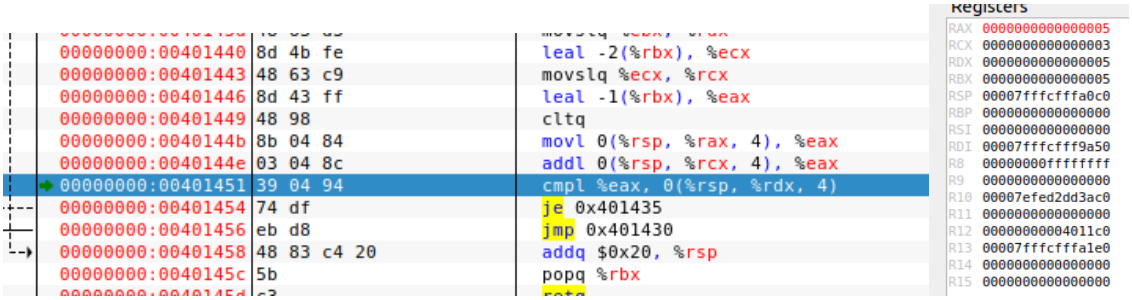
运行到第 3 次比较时，比较了%eax 寄存器中的值和内存 0(%rsp, %rdx, 4)中的值，通过 EDB 软件查看此时的寄存器内容可知此时%rdx 寄存器中的值为 2，%eax 寄存器中的值为 1，因此此时访问的是内存中 rsp 偏移为 2*4 的值，即输入的第 3 个数应当为 1。



在第 4 次比较时，同样比较了%eax 寄存器中的值和内存 0(%rsp, %rdx, 4)中的值，通过 EDB 软件查看此时的寄存器内容可知此时%rdx 寄存器中的值为 3，%eax 寄存器中的值为 2，因此输入的第 4 个数应该为 2。



第 5 次比较的代码与上面 2 次相同，只有寄存器的值不一样，此时%eax 寄存器的值为 3，%rdx 为 4，即输入的第 5 个数应该为 3。



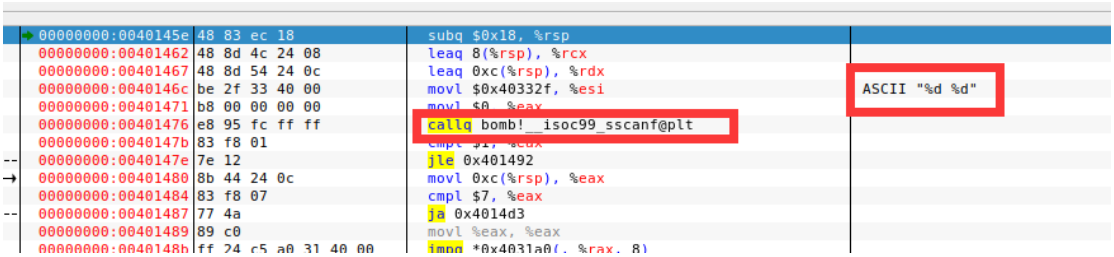
第 6 次比较时，%rdx 值为 5，%eax 值为 5，因此第 6 个数输入 5 即可。

由此可知输入的 6 个数分别为：0 1 1 2 3 5。

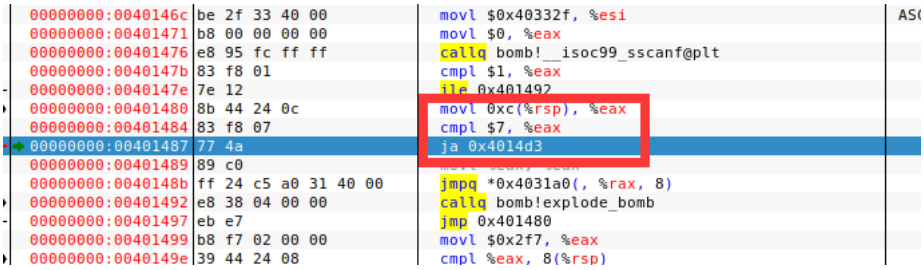
3.3 阶段 3 的破解与分析

密码如下：6 786

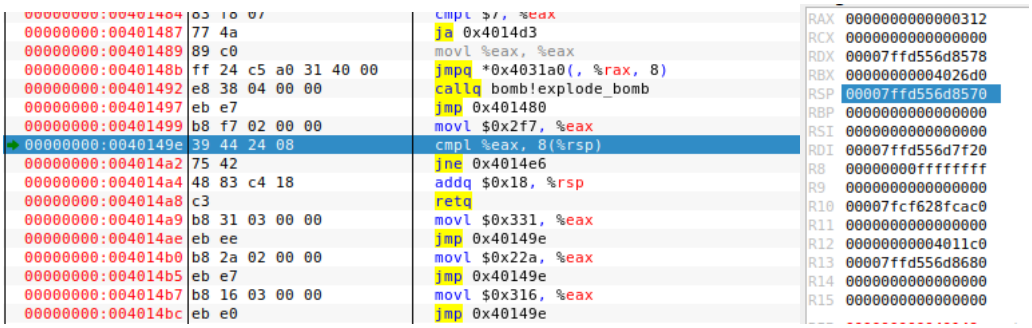
破解过程：



依然利用 EDB 动态调试第三关程序，由第 3 关刚开始的函数中“%d %d”字符串可以知道应该是输入了两个整数，结果放到 rsp 偏移量分别为 0xC 和 0x8 的位置(第一个数偏移为 0xC，第二个数偏移为 0x8)，偏移即十进制的 12 和 8，是两个 4 字节的整数。



读取两数之后将 rsp 偏移为 0xC(第一个数)放到%eax 寄存器中，然后将这个数与 7 比较，如果大于 7 就跳转到 0x4014d3(这个位置将执行爆炸函数，我们不应让这个函数执行)，因此必须让输入的第一个数小于或等于 7。



第二次比较时，比较了 rsp 偏移 8 处的数据(也就是输入的第二个数)和 eax 寄存器，通过 EDB 软件动态分析可知此时 eax 寄存器中的数为 0x312(十进制的 786)，如果两数不相等就执行爆炸函数，因此可知此处第二个数应该输入 786。

综上，第 3 关的密码可以输入 6 786。

3.4 阶段 4 的破解与分析

密码如下：10 37

破解过程：

```
00000000:0040151f 48 83 ec 18          subq $0x18, %rsp
00000000:00401523 48 8d 4c 24 08       leaq 8(%rsp), %rcx
00000000:00401528 48 8d 54 24 0c       leaq 0xc(%rsp), %rdx
00000000:0040152d be 2f 33 40 00       movl $0x40332f, %esi                ASCII "%d %d"
00000000:00401532 b8 00 00 00 00       movl $0, %eax
00000000:00401537 e8 d4 fb ff ff       callq bomb!__isoc99_sscanf@plt
00000000:0040153c 83 f8 02             cmpl $2, %eax
00000000:0040153f 75 0d                jne 0x40154e
```

通过第 4 关刚开始的代码分析可知这一关还是读取了两个整数，第一个整数放在 rsp 偏移 0xC 的位置，第二个整数放在 rsp 偏移 0x8 的位置。

```
00000000:0040152d be 2f 33 40 00       movl $0x40332f, %esi
00000000:00401532 b8 00 00 00 00       movl $0, %eax
00000000:00401537 e8 d4 fb ff ff       callq bomb!__isoc99_sscanf@plt
00000000:0040153c 83 f8 02             cmpl $2, %eax
00000000:0040153f 75 0d                jne 0x40154e
00000000:00401541 8b 44 24 0c          movl 0xc(%rsp), %eax
00000000:00401545 85 c0                testl %eax, %eax
00000000:00401547 78 05                js 0x40154e
00000000:00401549 83 f8 0e             cmpl $0xe, %eax
00000000:0040154c 7e 05                jle 0x401553
00000000:0040154e e8 7c 03 00 00       callq bomb!explode_bomb
```

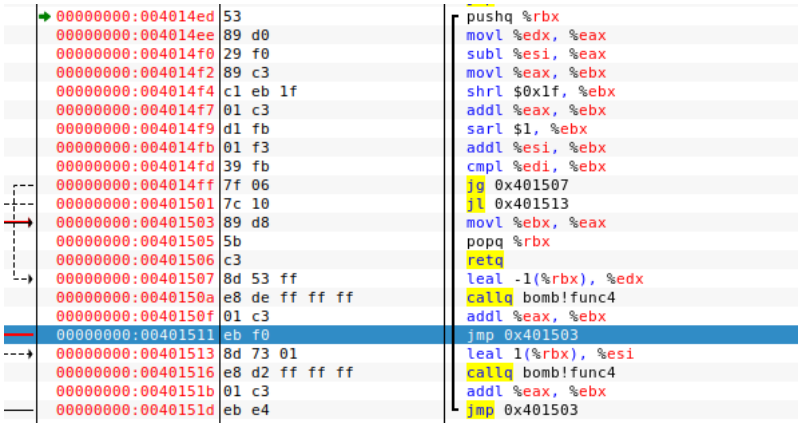
第一次比较时，将 rsp 偏移 0xC 的数放到了 rax 寄存器中，也就是输入的第一个数，通过比较，如果这个数是个负数就会跳转到 0x40154e，也就是爆炸函数，因此输入的第一个数必须是正数或 0。

```
00000000:00401541 8b 44 24 0c          movl 0xc(%rsp), %eax
00000000:00401545 85 c0                testl %eax, %eax
00000000:00401547 78 05                js 0x40154e
00000000:00401549 83 f8 0e             cmpl $0xe, %eax
00000000:0040154c 7e 05                jle 0x401553
00000000:0040154e e8 7c 03 00 00       callq bomb!explode_bomb
00000000:00401553 ba 0e 00 00 00       movl $0xe, %edx
00000000:00401558 be 00 00 00 00       movl $0, %esi
00000000:0040155d 8b 7c 24 0c          movl 0xc(%rsp), %edi
00000000:00401561 e8 87 ff ff ff       callq bomb!func4
```

第二次比较同样比较的是输入的第一个数，如果这个数小于或等于 0xe，就不会执行爆炸函数，因此要求输入的第一个数不大于 0xe，也就是十进制的 14。

```
00000000:0040153f 75 0d                jne 0x40154e
00000000:00401541 8b 44 24 0c          movl 0xc(%rsp), %eax
00000000:00401545 85 c0                testl %eax, %eax
00000000:00401547 78 05                js 0x40154e
00000000:00401549 83 f8 0e             cmpl $0xe, %eax
00000000:0040154c 7e 05                jle 0x401553
00000000:0040154e e8 7c 03 00 00       callq bomb!explode_bomb
00000000:00401553 ba 0e 00 00 00       movl $0xe, %edx
00000000:00401558 be 00 00 00 00       movl $0, %esi
00000000:0040155d 8b 7c 24 0c          movl 0xc(%rsp), %edi
00000000:00401561 e8 87 ff ff ff       callq bomb!func4
00000000:00401566 83 f8 25             cmpl $0x25, %eax
00000000:00401569 75 07                jne 0x401572
00000000:0040156b 83 7c 24 08 25       cmpl $0x25, 8(%rsp)
00000000:00401570 74 05                je 0x401577
00000000:00401572 e8 58 03 00 00       callq bomb!explode_bomb
00000000:00401577 48 83 c4 18          addq $0x18, %rsp
```

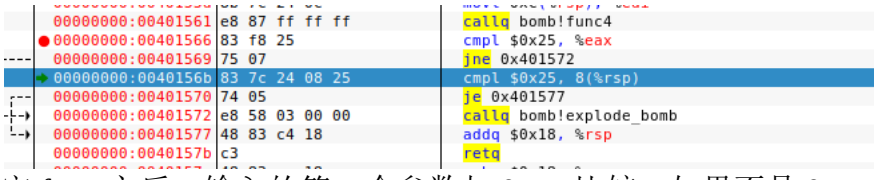
第三次比较时用到了一个函数 fun4，如果函数 fun4 的返回值不等于 0x25 则会导致爆炸函数，因此必须设法让 fun4 的返回值为 0x25，也就是十进制的 37。fun4 的参数为{%edi: 输入的第一个参数，%esi: 0，%edx: 0xe，也就是十进制的 14}。下面分析函数 fun4。



本人通过逆向工程技术，逐步分析语句，将函数 fun4 反编译出的 C 语言代码如下：

```
fun4(x, y, z)
//x in %edi, y in %esi, z in %edx
{
    int ret;
    ret = z - y;
    int n = (unsigned)ret >> 0x1f; //n in %ebx，0x1f = 31(十进制)，获取 n 的符号位
    n += ret;
    n >>= 1;
    n += y;
    if(n > x){
        //0x401507
        z = n - 1;
        n += fun4(x, y, z);
    }
    else if(n < x){
        //0x401513
        y = n + 1;
        n += fun4(x, y, z);
    }
    ret = n;
    return ret;
}
```

在第 4 关的函数中初次调用 fun4 是的传参为 fun4(屏幕输入的第一个参数，0, 14)。由 fun4 函数的定义可知，为了让 fun4 函数返回 37，只需要控制 fun4 的第一个参数即可(即在第 4 关最初输入的第一个参数)，而由上面的分析可知第一个参数只能取 0~14 之间的某个数，通过逐一尝试输入 0~14 发现，当输入的第一个参数为 10(十六进制的 0xa)时，可以让 fun4 函数返回 37(0x25)，因此第一个数必须输入 10。



执行完 fun4 之后，输入的第 2 个参数与 0x25 比较，如果不是 0x25 则执行爆

炸函数，因此输入的第 2 个参数必须是 0x25，也就是十进制的 37。

综上，输入的两个数应当为 10 37。

3.5 阶段 5 的破解与分析

密码如下：5 115

破解过程：

```
00000000:0040157c 48 83 ec 18           subq $0x18, %rsp
00000000:00401580 48 8d 4c 24 08        leaq 8(%rsp), %rcx
00000000:00401585 48 8d 54 24 0c        leaq 0xc(%rsp), %rdx
00000000:0040158a be 2f 33 40 00        movl $0x40332f, %esi          ASCII "%d %d"
00000000:0040158f b8 00 00 00 00        movl $0, %eax
00000000:00401594 e8 77 fb ff ff        callq bomb!__isoc99_sscanf@plt
00000000:00401599 83 f8 01              cmpl $1, %eax
00000000:0040159c 7e 32                 jle 0x4015d0
00000000:0040159e 8b 44 24 0c           movl 0xc(%rsp), %eax
00000000:004015a2 83 e0 0f              andl $0xf, %eax
```

结合前面几关的经验，从这段神秘代码中很容易发现依然是读取两个整数并检验其输入的值是否符合预期。

```
00000000:0040157c 48 83 ec 18           subq $0x18, %rsp
00000000:00401580 48 8d 4c 24 08        leaq 8(%rsp), %rcx
00000000:00401585 48 8d 54 24 0c        leaq 0xc(%rsp), %rdx
00000000:0040158a be 2f 33 40 00        movl $0x40332f, %esi
00000000:0040158f b8 00 00 00 00        movl $0, %eax
00000000:00401594 e8 77 fb ff ff        callq bomb!__isoc99_sscanf@plt
00000000:00401599 83 f8 01              cmpl $1, %eax
00000000:0040159c 7e 32                 jle 0x4015d0
00000000:0040159e 8b 44 24 0c           movl 0xc(%rsp), %eax
00000000:004015a2 83 e0 0f              andl $0xf, %eax
00000000:004015a5 89 44 24 0c           movl %eax, 0xc(%rsp)
00000000:004015a9 b9 00 00 00 00        movl $0, %ecx
00000000:004015ae ba 00 00 00 00        movl $0, %edx
00000000:004015b3 8b 44 24 0c           movl 0xc(%rsp), %eax
00000000:004015b7 83 f8 0f              cmpl $0xf, %eax
00000000:004015ba 74 1b                 je 0x4015d7
00000000:004015bc 83 c2 01              addl $1, %edx
00000000:004015bf 48 98                 cltq
00000000:004015c1 8b 04 85 e0 31 40 00  movl 0x4031e0(, %rax, 4), %eax
00000000:004015c8 89 44 24 0c           movl %eax, 0xc(%rsp)
00000000:004015cc 01 c1                 addl %eax, %ecx
00000000:004015ce eb e3                 jmp 0x4015b3
00000000:004015d0 e8 fa 02 00 00        callq bomb!explode_bomb
00000000:004015d5 eb c7                 jmp 0x40159e
00000000:004015d7 83 fa 0f              cmpl $0xf, %edx
00000000:004015da 75 06                 jne 0x4015e2
00000000:004015dc 39 4c 24 08           cmpl %ecx, 8(%rsp)
00000000:004015e0 74 05                 je 0x4015e7
00000000:004015e2 e8 e8 02 00 00        callq bomb!explode_bomb
00000000:004015e7 48 83 c4 18           addq $0x18, %rsp
00000000:004015eb c3                    retq
```

将第 5 关的这部分代码手动反编译可以得到如下 C 代码：

```
int a, b;
scanf("%d %d", &a, &b);    //a in %0xc(%rsp), b in 0x8(%rsp)
int ret = a & 0xf; //0b0000 1111 即取 a 的低 4 位，0～15
a = ret;    //ret in %eax
int n = 0; //n in %ecx
int m = 0; //m in %edx

do{
    ret = a;
    if(ret == 0xf)
        break;
    m += 1;
```

```
    ret = array[ret]; //array starts from 0x4031e0
    /*Array:0xa, 0x2, 0xe, 0x7, 0x8, 0xc, 0xf, 0xb, 0x0, 0x4, 0x1, 0xd, 0x3, 0x9, 0x6,
0x5
Array[0] = 0xa;
Array[1] = 0x2;
Array[2] = 0xe;
Array[3] = 0x7;
Array[4] = 0x8;
Array[5] = 0xc;
Array[6] = 0xf;
Array[7] = 0xb;
Array[8] = 0x0;
Array[9] = 0x4;
Array[0xa] = 0x1;
Array[0xb] = 0xd;
Array[0xc] = 0x3;
Array[0xd] = 0x9;
Array[0xe] = 0x6;
Array[0xf] = 0x5
*/
    a = ret;
    n += ret;
}while(1);

if(m != 0xf || b != n)
    bomb();
```

由此代码可知，输入的第一个数(也就是代码中的 a 变量)只有低 4 位有效，因此无论如何输入 a 的值，只有 0~15 这 16 个数是有效的。在 do-while 循环体中，只有当 ret 的值为 0xf(也就是十进制的 15)时才会 break，也就是说要让 a 的值最终被修改为 0xf。同时，要让 m 的值为 0xf，也就是说要执行 15 次循环体才行。Array 数组形成的链式结构如下：

0x0→0xa→0x1→0x2→0xe→0x6→0xf→0x5→0xc→0x3→0x7→0xb→0xd→0x9→0x4→0x8→0x0

因此让初始时的 a 为链表节点 0xf 之后的 0x5，即可让循环体执行 15 次。

同时要求第 2 个输入的数字等于最终的 n，通过分析可知最终 n = 0x0+0x1+0x2+0x3+…+0xf-0x5 = 115。之所以要减去 0x5，是因为第一次输入的 0x5 赋给 ret 之后执行了 ret = Array[ret]，此时的 ret 已经被修改过一次，因此不需要加 0x5。

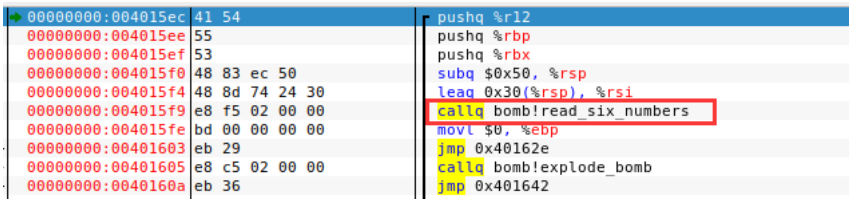
因此可以分析出输入的两个数分别是：5, 115。

3.6 阶段 6 的破解与分析

密码如下：6 5 1 4 3 2

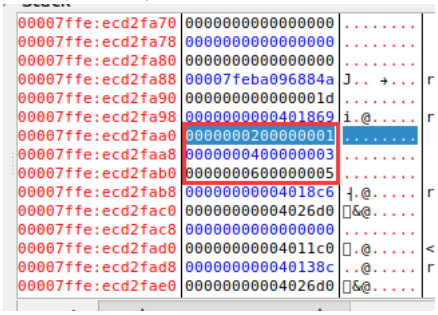
破解过程：

第 6 关的破解极其复杂，涉及到链表数据结构。



从第 6 关刚开始执行的过程来看，第 6 关是要输入 6 个数字并检验其正确性。

初次尝试输入六个数 1 2 3 4 5 6，观察堆栈内容，可以发现 6 个数分别放在 rsp 偏移为 0x30, 0x34, 0x38, 0x3C, 0x40, 0x44 的位置。



通过分析汇编代码发现，在第 6 关函数的栈帧中，不光有 6 个输入的整数，还存在 6 个 8 字节的指针数据，位于 rsp 偏移 0x0 开始的 6 个位置。

通过对第 6 关的代码反汇编，发现其中存在一个链表，对其代码进行手工反编译后得到如下 C 语言代码：

```
#include <stdio.h>
void bomb();
int main()
{
   int a[6];  //rsp+0x30, rsp+0x34, rsp+0x38, rsp+0x3C, rsp+0x40, rsp+0x44
   long *l[6]; //rsp+0x0, rsp+0x8, rsp+0x10, rsp+0x18, rsp+0x20, rsp+0x28
   scanf("%d %d %d %d %d %d", &a[0], &a[1], &a[2], &a[3], &a[4], &a[5]);
   long rax, rbx, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15;
   long rcx;
   long rdx;

   //0x4015fe
   //这段代码要求输入的 6 个数都小于等于 6，且两两不同
   for (rbp = 0; rbp <= 5; rbp++)
   {
      if (a[rbp] - 1 > 5)
         bomb();
      for (rbx = rbp + 1; rbx <= 5; rbx++)
      {
         if (a[rbx] == a[rbp])
            bomb();
```

```
    }
  }
  //0x40164b
  for (rsi = 0; rsi <= 5; rsi++)
  {
    rax = 1;  //rax 是一个计数器
    rdx = 0x4052d0; //这是一个全局的   链表的地址
    //其内容为：
    /**
     * 1：004052d0: 00 00 00 00 00 40 52 e0 00 00 00 01 00 00 01 a6
     * 2：0x4052e0: 00 00 00 00 00 40 52 f0 00 00 00 02 00 00 03 d5
     * 3：0x4052f0: 00 00 00 00 00 40 53 00 00 00 00 03 00 00 03 a2
     * 4：0x405300: 00 00 00 00 00 40 53 10 00 00 00 04 00 00 03 6c
     * 5：0x405310: 00 00 00 00 00 40 53 20 00 00 00 05 00 00 01 92
     * 6：0x405320: 00 00 00 00 00 00 00 00 00 00 00 06 00 00 01 80
     *
     */
    //可以推断这是一个链表
    /*
    对于一个 rsi, 下面的循环执行 a[rsi]次，之后把指针 l[rsi]赋为第 a[rsi]个节点(头节点为第 1 个而不是第 0 个节点)
    */
    while (a[rsi] > rax)
    {
      rdx = *((long *)(rdx + 8)); //使 rdx 指向下一个节点
      rax++;
    }
    l[rsi] = rdx;
  }

  //0x40167a
  rbx = l[0];
  rcx = rbx;
  //这部分代码是从 l[0]指针开始，按照 l[]指针数组把链表结点重新排序
  for (rax = 1; rax <= 5; rax++)
  {
    rdx = rax;
    rdx = l[rdx];
    *((long *)(rcx + 8)) = rdx;
    rcx = rdx;
  }
  //这部分代码要求前一个结点的值小于或等于后一个结点的值，因此排序后的结点应当为：
  //0x405320->0x405310->004052d0->0x405300->0x4052f0->0x4052e0
  //由此可以判断输入的六个数为：6 5 1 4 3 2
```

```
    *((long *)(rcx + 8)) = 0;
    for (rbp = 0, rbp <= 4; rbp++)
    {
        rbp = 0;
        rax = *((long *)(rcx + 8));
        rax = *((long *)(rax));
        if (*((long *)(rbx)) < rax)
            bomb();
    }
}
```

这个函数的栈帧中包含两个数组，其中一个是输入的 6 个整数的数组，另一个是包含 6 个指针的数组，这个指针数组分别用来指向链表的每一个结点。

其中，第一个循环要求输入的 6 个数两两不同且都不大于 6。

第 2 个循环修改了函数的指针数组，将指针数组的第 i 个元素赋为第 a[i]个结点的地址，其中 a[i]即为输入的第 i 个整数。

第 3 个循环修改了链表结构，按照函数分配的指针数组中的 6 个元素(6 个地址)的顺序，重新排列链表元素。

第 4 个循环依次检验链表元素，要求链表中前一个结点的数据必须不大于其后继节点的数据值。

通过 EDB 动态调试，可以发现程序中链表初始状态如：

*1：*004052d0: 00 00 00 00 00 40 52 e0 00 00 00 01 00 00 01 a6*

*2：*0x4052e0: 00 00 00 00 00 40 52 f0 00 00 00 02 00 00 03 d5*

*3：*0x4052f0: 00 00 00 00 00 40 53 00 00 00 00 03 00 00 03 a2*

*4：*0x405300: 00 00 00 00 00 40 53 10 00 00 00 04 00 00 03 6c*

*5：*0x405310: 00 00 00 00 00 40 53 20 00 00 00 05 00 00 01 92*

*6：*0x405320: 00 00 00 00 00 00 00 00 00 00 00 06 00 00 01 80*

因此，只需要输入 6 个正整数，使得链表重新排列后，能够按照从小到大的顺序排序即可。六个结点的数据从小到大排列依次是：6 5 1 2 3 4，每一个数字对应着一个结点的序号。

在输入这六个数之后，函数的指针数组 l[]会依次存储第 6 个结点、第 5 个结点、第 1 个结点、第 2 个结点、第 3 个结点、第 4 个结点的地址。经过第 3 个循环之后，结点 6 的后继节点变为第 5 个结点，结点 5 的后继节点变为第 1 个结点，……，结点 3 的后继节点变为结点 4，结点 4 的后继节点变为空。然后，再以结点 6 为头节点依次遍历链表，即可得到一个单调不减的序列，至此，第 6 关可以顺利通过。

综上所述，第 6 关需要输入的数据是：6 5 1 2 3 4

3.7 阶段 7 的破解与分析(隐藏阶段)

密码如下：在第 4 关时输入“10 37 DrEvil”，在隐藏关卡输入“47”

破解过程：

为了发现隐藏关卡的入口，直接利用 objdump -d 反汇编 bomp 可执行文件，发现其中存在字样“secret_phase”,推测这就是隐藏关卡的入口。

```
  401aaa:   48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  401aaf:   e8 39 fd ff ff          callq  4017ed <strings_not_equal>
  401ab4:   85 c0                   test   %eax,%eax
  401ab6:   75 de                   jne    401a96 <phase_defused+0x36>
  401ab8:   bf 58 32 40 00          mov    $0x403258,%edi
  401abd:   e8 9e f5 ff ff          callq  401060 <puts@plt>
  401ac2:   bf 80 32 40 00          mov    $0x403280,%edi
  401ac7:   e8 94 f5 ff ff          callq  401060 <puts@plt>
  401acc:   b8 00 00 00 00          mov    $0x0,%eax
  401ad1:   e8 3a fc ff ff          callq  401710 <secret_phase>
  401ad6:   eb be                   jmp    401a96 <phase_defused+0x36>

0000000000401ad8 <rio_readinitb>:
  401ad8:   89 37                   mov    %esi,(%rdi)
  401ada:   c7 47 04 00 00 00 00    movl   $0x0,0x4(%rdi)
  401ae1:   48 8d 47 10             lea    0x10(%rdi),%rax
  401ae5:   48 89 47 08             mov    %rax,0x8(%rdi)
  401ae9:   c3                      retq
```

只有在 phase_defused 过程才会调用 secret_phase。

```
0000000000401a60 <phase_defused>:
  401a60:   83 3d 05 3d 00 00 06    cmpl   $0x6,0x3d05(%rip)        # 40576c <num_input_strings>
  401a67:   74 01                   je     401a6a <phase_defused+0xa>
  401a69:   c3                      retq
  401a6a:   48 83 ec 68             sub    $0x68,%rsp
  401a6e:   4c 8d 44 24 10          lea    0x10(%rsp),%r8
  401a73:   48 8d 4c 24 08          lea    0x8(%rsp),%rcx
  401a78:   48 8d 54 24 0c          lea    0xc(%rsp),%rdx
  401a7d:   be 79 33 40 00          mov    $0x403379,%esi
  401a82:   bf 70 58 40 00          mov    $0x405870,%edi
  401a87:   b8 00 00 00 00          mov    $0x0,%eax
  401a8c:   e8 7f f6 ff ff          callq  401110 <__isoc99_sscanf@plt>
  401a91:   83 f8 03                cmp    $0x3,%eax
  401a94:   74 0f                   je     401aa5 <phase_defused+0x45>
  401a96:   bf b8 32 40 00          mov    $0x4032b8,%edi
  401a9b:   e8 c0 f5 ff ff          callq  401060 <puts@plt>
  401aa0:   48 83 c4 68             add    $0x68,%rsp
  401aa4:   c3                      retq
  401aa5:   be 82 33 40 00          mov    $0x403382,%esi
  401aaa:   48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  401aaf:   e8 39 fd ff ff          callq  4017ed <strings_not_equal>
  401ab4:   85 c0                   test   %eax,%eax
  401ab6:   75 de                   jne    401a96 <phase_defused+0x36>
  401ab8:   bf 58 32 40 00          mov    $0x403258,%edi
  401abd:   e8 9e f5 ff ff          callq  401060 <puts@plt>
  401ac2:   bf 80 32 40 00          mov    $0x403280,%edi
  401ac7:   e8 94 f5 ff ff          callq  401060 <puts@plt>
  401acc:   b8 00 00 00 00          mov    $0x0,%eax
  401ad1:   e8 3a fc ff ff          callq  401710 <secret_phase>
  401ad6:   eb be                   jmp    401a96 <phase_defused+0x36>
```

```
4012e4:   e8 77 fd ff ff        callq  401060 <puts@plt>
4012e9:   bf c8 30 40 00        mov    $0x4030c8,%edi
4012ee:   e8 6d fd ff ff        callq  401060 <puts@plt>
4012f3:   e8 3a 06 00 00        callq  401932 <read_line>
4012f8:   48 89 c7              mov    %rax,%rdi
4012fb:   e8 ec 00 00 00        callq  4013ec <phase_1>
401300:   e8 5b 07 00 00        callq  401a60 <phase_defused>
401305:   bf f8 30 40 00        mov    $0x4030f8,%edi
40130a:   e8 51 fd ff ff        callq  401060 <puts@plt>
40130f:   e8 1e 06 00 00        callq  401932 <read_line>
401314:   48 89 c7              mov    %rax,%rdi
401317:   e8 ee 00 00 00        callq  40140a <phase_2>
40131c:   e8 3f 07 00 00        callq  401a60 <phase_defused>
401321:   bf 3d 30 40 00        mov    $0x40303d,%edi
401326:   e8 35 fd ff ff        callq  401060 <puts@plt>
40132b:   e8 02 06 00 00        callq  401932 <read_line>
401330:   48 89 c7              mov    %rax,%rdi
401333:   e8 26 01 00 00        callq  40145e <phase_3>
401338:   e8 23 07 00 00        callq  401a60 <phase_defused>
40133d:   bf 5b 30 40 00        mov    $0x40305b,%edi
401342:   e8 19 fd ff ff        callq  401060 <puts@plt>
401347:   e8 e6 05 00 00        callq  401932 <read_line>
40134c:   48 89 c7              mov    %rax,%rdi
40134f:   e8 cb 01 00 00        callq  40151f <phase_4>
401354:   e8 07 07 00 00        callq  401a60 <phase_defused>
401359:   bf 28 31 40 00        mov    $0x403128,%edi
40135e:   e8 fd fc ff ff        callq  401060 <puts@plt>
401363:   e8 ca 05 00 00        callq  401932 <read_line>
401368:   48 89 c7              mov    %rax,%rdi
40136b:   e8 0c 02 00 00        callq  40157c <phase_5>
401370:   e8 eb 06 00 00        callq  401a60 <phase_defused>
401375:   bf 6a 30 40 00        mov    $0x40306a,%edi
40137a:   e8 e1 fc ff ff        callq  401060 <puts@plt>
40137f:   e8 ae 05 00 00        callq  401932 <read_line>
401384:   48 89 c7              mov    %rax,%rdi
401387:   e8 60 02 00 00        callq  4015ec <phase_6>
40138c:   e8 cf 06 00 00        callq  401a60 <phase_defused>
401391:   b8 00 00 00 00        mov    $0x0,%eax
401396:   5b                    pop    %rbx
```

在主函数当中，每通过一关，都会调用依次 phase_defused 函数。

```
00000000:00401a63 3d 00 00 06 74    cmpl $0x74060000, %eax
00000000:00401a68 01 c3             addl %eax, %ebx
00000000:00401a6a 48 83 ec 68       subq $0x68, %rsp
00000000:00401a6e 4c 8d 44 24 10    leaq 0x10(%rsp), %r8
00000000:00401a73 48 8d 4c 24 08    leaq 8(%rsp), %rcx
00000000:00401a78 48 8d 54 24 0c    leaq 0xc(%rsp), %rdx
00000000:00401a7d be 79 33 40 00    movl $0x403379, %esi        ASCII "%d %d %s"
00000000:00401a82 bf 70 58 40 00    movl $0x405870, %edi        ASCII "10 37"
00000000:00401a87 b8 00 00 00 00    movl $0, %eax
00000000:00401a8c e8 7f f6 ff ff    callq bomb!__isoc99_sscanf@plt
00000000:00401a91 83 f8 03          cmpl $3, %eax
00000000:00401a94 74 0f             je 0x401aa5
00000000:00401a96 bf b8 32 40 00    movl $0x4032b8, %edi        ASCII "Congratulations! You've defused the bomb!"
00000000:00401a9b e8 c0 f5 ff ff    callq bomb!puts@plt
00000000:00401aa0 48 83 c4 68       addq $0x68, %rsp
00000000:00401aa4 c3                retq
00000000:00401aa5 be 82 33 40 00    movl $0x403382, %esi        ASCII "DrEvil"
00000000:00401aaa 48 8d 7c 24 10    leaq 0x10(%rsp), %rdi
00000000:00401aaf e8 39 fd ff ff    callq bomb!strings_not_equal
00000000:00401ab4 85 c0             testl %eax, %eax
00000000:00401ab6 75 de             jne 0x401a96
00000000:00401ab8 bf 58 32 40 00    movl $0x403258, %edi        ASCII "Curses, you've found the secret phase!"
00000000:00401abd e8 9e f5 ff ff    callq bomb!puts@plt
00000000:00401ac2 bf 80 32 40 00    movl $0x403280, %edi        ASCII "But finding it and solving it are quite different..."
00000000:00401ac7 e8 94 f5 ff ff    callq bomb!puts@plt
00000000:00401acc b8 00 00 00 00    movl $0, %eax
```

在 EDB 动态调式中，发现 phase_defused 函数中以一种奇怪的方式调用了函数 scanf(“10 37”, “%d %d %s”);并且只有当返回值为 3 时(即 scanf 成功读入了 3 个内容)才能执行隐藏关卡。并且下面的代码要求%s 读取的字符串为”DrEvil”，联想到第 4 关输入的答案就是 10 37，推测这里要在第 4 关答案后面多输入一个字符串 DrEvil。在执行完第 4 关后没有立即进入隐藏关卡，而是在 6 关都通过之后才进入了隐藏关卡，推测是程序中对通过的关卡进行了计数，只有在 6 关都通过了之后才能进入隐藏关卡。在 phase_defused 函数中发现了一处比较印证了这一点。

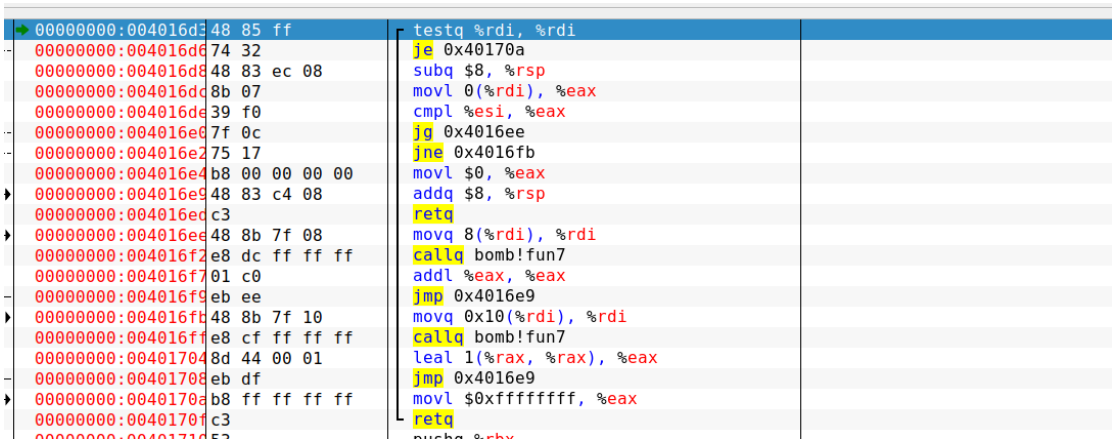
```
00000000:00401a54 48 89 b8 88 57 4…   movq %rdi, 0x405788(%rax)
00000000:00401a5b e8 6f fe ff ff      callq bomb!explode_bomb
00000000:00401a60 83 3d 05 3d 00 0…   cmpl $6, 0x3d05(%rip)
00000000:00401a67 74 01               je 0x401a6a
00000000:00401a69 c3                  retq
00000000:00401a6a 48 83 ec 68         subq $0x68, %rsp
00000000:00401a6e 4c 8d 44 24 10      leaq 0x10(%rsp), %r8
00000000:00401a73 48 8d 4c 24 08      leaq 8(%rsp), %rcx
00000000:00401a78 48 8d 54 24 0c      leaq 0xc(%rsp), %rdx
00000000:00401a7d be 79 33 40 00      movl $0x403379, %esi
```

在进入 secret 关卡之后，分析 secret 关卡的过程。

```
00000000:00401710 53                  pushq %rbx
00000000:00401711 e8 1c 02 00 00      callq bomb!read_line
00000000:00401716 48 89 c7            movq %rax, %rdi
00000000:00401719 e8 22 fa ff ff      callq bomb!atoi@plt
00000000:0040171e 89 c3               movl %eax, %ebx
00000000:00401720 8d 40 ff            leal -1(%rax), %eax
00000000:00401723 3d e8 03 00 00      cmpl $0x3e8, %eax
00000000:00401728 77 22               ja 0x40174c
00000000:0040172a 89 de               movl %ebx, %esi
00000000:0040172c bf f0 50 40 00      movl $0x4050f0, %edi
00000000:00401731 e8 9d ff ff ff      callq bomb!fun7
00000000:00401736 83 f8 05            cmpl $5, %eax
00000000:00401739 75 18               jne 0x401753
00000000:0040173b bf 78 31 40 00      movl $0x403178, %edi
00000000:00401740 e8 1b f9 ff ff      callq bomb!puts@plt
00000000:00401745 e8 16 03 00 00      callq bomb!phase_defused
00000000:0040174a 5b                  popq %rbx
00000000:0040174b c3                  retq
00000000:0040174c e8 7e 01 00 00      callq bomb!explode_bomb
00000000:00401751 eb d7               jmp 0x40172a
00000000:00401753 e8 77 01 00 00      callq bomb!explode_bomb
00000000:00401758 eb e1               jmp 0x40173b
```

可以发现，隐藏关卡要求输入一个字符串，然后调用 atoi 函数，把这个字符串转换成一个整数放在 rax 寄存器中，然后把这个值用 ebx 寄存器保存。接着又检测 rax 的值减去 1 后是否大于 0x3e8(十进制的 1000)，如果大于则炸弹爆炸，因此要求输入的数字小于等于 1001。

接着，隐藏关卡调用了函数 fun7，参数分别为 0x4050f0 和 ebx 寄存器的值，ebx 寄存器的值也就是我们通过终端输入的整数。要求 fun7 函数的返回值等于 5，则隐藏关卡通过。下面分析函数 fun7。



函数 fun7 首先检测了 rdi 寄存器，也就是第一个参数是否为 0，如果为 0 则返回 0xffffffff(十进制-1)，而我们希望 fun7 返回 0x5，也就不符合要求，因此不希望第一个参数为 1。Fun7 涉及到递归，我们把这段代码手工反编译，可以得到如下 C 语言代码：

```
int fun7(int *p, int x)
{
   if (!p)
      return -1;
   //sub $8, %rsp
   int ret = *p;
   if (ret > x)
   {
      ret = 2 * fun7(*(p + 2), x); //(char*)p+0x8
   }
   else if (ret < x)
   {
      ret = 2 * fun7(*(p + 4), x) + 1; //(char*)p+0x10
   }
   else
   {
     ret = 0;
   }
   return ret;
}
```

通过这段递归发现 p==0 或 x==*p 是终止条件，而 p==0 会引起返回-1，-2 再乘 2 得到的仍然是一个负数，不会返回+5，因此猜测应该让 x==*p 作为递归终止条件。初次调用函数时，p=0x4050f0。

分析内存内容：

地址 p	p+0 内容	p+0x8 内容	p+0x10 内容
0x4050f0	0x24	0x405110	0x405130
0x405110	0x8	0x405190	0x405150
0x405130	0x32	0x405170	0x4051b0
0x405190	0x6	0x4051f0	0x405250
0x405150	0x16	0x405270	0x405230

0x405170	0x2d	0x4051d0	0x405290
0x4051b0	0x6b	0x405210	0x4052b0
0x4051f0	0x01	0x0	0x0
0x405250	0x07	0x0	0x0
0x405270	0x14	0x0	0x0
0x405230	0x23	0x0	0x0
0x4051d0	0x28	0x0	0x0
0x405290	0x2f	0x0	0x0
0x405210	0x63	0x0	0x0
0x4052b0	0x03e9	0x0	0x0

通过内存可以看到这部分内存类似一棵二叉树。

在递归返回时，为了让最终的返回值是 5，可以通过这样的方法构造。

在最后一次调用函数时，返回 0；在倒数第二次调用函数时，走 2*fun7+1 分支，这样返回值会变成 1；在倒数第三次调用函数时，走 2*fun7 分支，这样会让返回值变成 2；在倒数第四次调用函数时，走 2*fun7+1 分支，这样会让返回值变成 5。

因此经过分析，可以得到如下的函数调用过程。

第一次走 2*fun7+1 分支：p=0x4050f0, x > 0x24

第二次走 2*fun7 分支：p=0x405130, x < 0x32

第三次走 2*fun7+1 分支：p=0x405170,x > 0x2d

第四次让函数返回 0：p=0x405290, x==0x2f

这样分析后，所需输入的数字应该是 0x2f，即为十进制的 47。

至此为止，所求出每一关的 solution 分别为：

I am just a renegade hockey mom.
0 1 1 2 3 5
6 786
10 37 DrEvil
5 115
6 5 1 4 3 2
47

第 4 章 总结

4.1 请总结本次实验的收获

- 学会了在 Ubuntu 下利用 EDB 动态调试可执行程序
- 学会了动态分析汇编代码
- 了解了结构体操作的汇编代码构成

4.2 请给出对本次实验内容的建议

- 本实验如果利用 IDA 等现代化反编译工具可以轻易完成，希望老师也可以在课堂上稍微介绍一下这类现代化工具。但是本实验如果真的用 IDA 来做，就失去了意义，无法真正地理解汇编代码。
- 炸弹的第 6 关是一个链表，在拆弹时一开始很难想到链表这一点，希望老师能给出一些提示。隐藏关卡是一个二叉树，也很难想到，如果给出一点提示则更加人性化。

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.