《计算机系统基础实践》任务书

实验二 机器级语言理解——二进制炸弹拆除

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

任务 1　二进制炸弹拆除

“二进制炸弹”是一个 Linux 可执行程序。每个同学按自己学号，到群文件中下载自己的压缩包，解压后可得到 bomb 执行程序和 bomb.c 总控源程序。

bomb 执行程序由六个阶段组成。每个阶段都需要输入特定的字符串。如果输入了正确的字符串，那么该阶段就“解除”。 否则，炸弹会通过打印“BOOM！！！”。每个同学的目标是解除尽可能多的阶段。

进度提示：

本实验使用两次课内上机，8 学时完成。找出尽可能多的密码字符串。

三、实验提示

每个同学的“解除”字符串也会不同。每个炸弹阶段会测试不同的方面：

第 1 阶段：字符串比较

第 2 阶段：循环

第 3 阶段：条件/开关

第 4 阶段：递归调用和堆栈规则

第 5 阶段：指针

第 6 阶段：链表/指针/结构体

还有一个“秘密阶段”，仅当学生将特定字符串附加到第四阶段时才会出现。

为了拆除炸弹，学生必须使用调试器，通常是 gdb 反汇编执行文件，并单步执行每个阶段的机器代码，使用课程中的知识来推断“解除”字符串。

还可以使用 objdump 工具，将 bomb 执行程序静态反汇编，通过研读生成的源程序，理解和掌握 bomb 的执行过程。objdump 反汇编得到源程序的命令为：

```
objdump  -d  bomb  >  bomb.s
```

四、实验记录和问题回答

首先，我们用 gdb 工具把程序进入调试环境下:

```
yuhang@yuhang-virtual-machine:~/桌面/bomb202315752$ gdb bomb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
```

图 4.1 gdb 环境

(1) 实验任务 1 的实验记录

phase_1:

我们先设置一些断点，让他停下来后，随便输入一些字符串“aaa”，看看表现如何：

```
(gdb) b phase_1
Breakpoint 1 at 0x14cd
(gdb) r
Starting program: /home/yuhang/桌面/bomb202315752/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
abc

Breakpoint 1, 0x565564cd in phase_1 ()
(gdb)
```

图 4.2 示例

我们输入 disas phase_1,进行反汇编，看到相关的代码情况：

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
=> 0x565564cd <+0>:     push   %ebx
   0x565564ce <+1>:     sub    $0x10,%esp
   0x565564d1 <+4>:     call   0x56556240 <__x86.get_pc_thunk.bx>
   0x565564d6 <+9>:     add    $0x4a8e,%ebx
   0x565564dc <+15>:    lea    -0x2e20(%ebx),%eax
   0x565564e2 <+21>:    push   %eax
   0x565564e3 <+22>:    pushl  0x1c(%esp)
   0x565564e7 <+26>:    call   0x56556b2f <strings_not_equal>
   0x565564ec <+31>:    add    $0x10,%esp
   0x565564ef <+34>:    test   %eax,%eax
   0x565564f1 <+36>:    jne    0x565564f8 <phase_1+43>
   0x565564f3 <+38>:    add    $0x8,%esp
   0x565564f6 <+41>:    pop    %ebx
   0x565564f7 <+42>:    ret
   0x565564f8 <+43>:    call   0x56556c47 <explode_bomb>
   0x565564fd <+48>:    jmp    0x565564f3 <phase_1+38>
End of assembler dump.
(gdb)
```

图 4.3 代码内容

Phase_1 的调用了<string_not_equal>函数，并且在前面调用%ebx，%esp,%eax，三个参数，我们猜测里面藏着我们的结果,于是我们逐个查看内容。我们发现我们向%eax 做了一次从

$ebx-0x2e20 的值传递的操作，所以我们输入这个值，看看是啥内容：

```
(gdb) x/s $eax
0x5655b3a0 <input_strings>:     "aaa"
(gdb) x/s $ebx-0x2e20
0x56558144:     "I am just a renegade hockey mom."
(gdb)
```

图 4.4 寄存器内容

我们发现，在%eax 里面是我们是我们输入的内容，而在$ebx-0x2e20 里面是另外一个字符串，我们猜测是结果，结果证明真的是结果：

```
(gdb) r
Starting program: /home/yuhang/桌面/bomb202315752/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am just a renegade hockey mom.
Phase 1 defused. How about the next one?
```

图 4.5 结果证明

图 4.3 代码内容我们分析如下：通过 add “$0x4a8e,%ebx”，程序将 %ebx 设置为一个偏移量，使它能够访问程序的静态存储区域。这一区域可能包含目标字符串（正确的字符串）。接下来，指令 “lea -0x2e20(%ebx),%eax” 计算了目标字符串的具体地址；lea 指令加载内存地址，而不是加载地址对应的值。“-0x2e20(%ebx)” 表的是从 %ebx 指向的地址减去 0x2e20 字节的地方。也就是说，这个计算出来的地址（存储在 %eax 中）是目标字符串在内存中的地址。因此，程序使用了一个相对偏移 -0x2e20，从 %ebx 指向的内存位置跳到了目标字符串所在的内存位置。因此，$ebx-0x2e20 是我们的地址

(2) 实验任务 2 的实验记录

phase_2:

我们先看看 phsase_2 的汇编代码情况，随机输入一串数字“1 2 3 4 5 6”后进入 Phase_2 的反汇编代码：

```
0x565564ff <+0>:     push   %edi
0x56556500 <+1>:     push   %esi
0x56556501 <+2>:     push   %ebx
0x56556502 <+3>:     sub    $0x28,%esp
0x56556505 <+6>:     call   0x56556240 <__x86.get_pc_thunk.bx>
0x5655650a <+11>:    add    $0x4a5a,%ebx
0x56556510 <+17>:    mov    %gs:0x14,%eax
0x56556516 <+23>:    mov    %eax,0x24(%esp)
0x5655651a <+27>:    xor    %eax,%eax
0x5655651c <+29>:    lea    0xc(%esp),%eax
0x56556520 <+33>:    push   %eax
0x56556521 <+34>:    pushl  0x3c(%esp)
0x56556525 <+38>:    call   0x56556c7c <read_six_numbers>
0x5655652a <+43>:    add    $0x10,%esp
0x5655652d <+46>:    cmpl   $0x0,0x4(%esp)
0x56556532 <+51>:    js     0x5655653f <phase_2+64>
0x56556534 <+53>:    mov    $0x1,%esi
0x56556539 <+58>:    lea    0x4(%esp),%edi
0x5655653d <+62>:    jmp    0x5655654e <phase_2+79>
0x5655653f <+64>:    call   0x56556c47 <explode_bomb>
0x56556544 <+69>:    jmp    0x56556534 <phase_2+53>
0x56556546 <+71>:    add    $0x1,%esi
```

图 4.6 代码内容

我们发现，这段代码首先把%edi,%esi 和%ebx 寄存器，同时给栈分配 40 字节的空间。后面，oxc（%esp）指向第一个数字存储的位置，后面实现 read_six_numbers 读取 6 个数字。

再往下面走，有一个语句“0x5655652d <+46>: cmpl $0x0,0x4(%esp)”说明一旦第一个数字小于 0，直接跑到“exploded_bomb”游戏失败并结束,所以，我们第一个数字不能小于 0

于是，我们进入“read_six_numbers”函数观察情况：

```
0x56556c7c <+0>:     push   %ebx
0x56556c7d <+1>:     sub    $0x8,%esp
0x56556c80 <+4>:     call   0x56556240 <__x86.get_pc_thunk.bx>
0x56556c85 <+9>:     add    $0x42df,%ebx
0x56556c8b <+15>:    mov    0x14(%esp),%eax
0x56556c8f <+19>:    lea    0x14(%eax),%edx
0x56556c92 <+22>:    push   %edx
0x56556c93 <+23>:    lea    0x10(%eax),%edx
0x56556c96 <+26>:    push   %edx
0x56556c97 <+27>:    lea    0xc(%eax),%edx
0x56556c9a <+30>:    push   %edx
0x56556c9b <+31>:    lea    0x8(%eax),%edx
0x56556c9e <+34>:    push   %edx
0x56556c9f <+35>:    lea    0x4(%eax),%edx
0x56556ca2 <+38>:    push   %edx
0x56556ca3 <+39>:    push   %eax
0x56556ca4 <+40>:    lea    -0x2c91(%ebx),%eax
0x56556caa <+46>:    push   %eax
0x56556cab <+47>:    pushl  0x2c(%esp)
0x56556caf <+51>:    call   0x56556140 <__isoc99_sscanf@plt>
0x56556cb4 <+56>:    add    $0x20,%esp
0x56556cb7 <+59>:    cmp    $0x5,%eax
```

图 4.7 函数内容

我们发现，在这个函数里面只是实现了把寄存器放在栈里面，同时给栈分配了六个空间以等待用户输入 6 个数值，利用 sscanf 函数输入

我们使用 s 命令，继续向下走：

```
0x56556549 <+74>:    cmp    $0x6,%esi
0x5655654c <+77>:    je     0x56556560 <phase_2+97>
0x5655654e <+79>:    mov    %esi,%eax
0x56556550 <+81>:    add    -0x4(%edi,%esi,4),%eax
0x56556554 <+85>:    cmp    %eax,(%edi,%esi,4)
0x56556557 <+88>:    je     0x56556546 <phase_2+71>
0x56556559 <+90>:    call   0x56556c47 <explode_bomb>
0x5655655e <+95>:    jmp    0x56556546 <phase_2+71>
0x56556560 <+97>:    mov    0x1c(%esp),%eax
0x56556564 <+101>:   sub    %gs:0x14,%eax
0x5655656b <+108>:   jne    0x56556574 <phase_2+117>
0x5655656d <+110>:   add    $0x20,%esp
0x56556570 <+113>:   pop    %ebx
0x56556571 <+114>:   pop    %esi
0x56556572 <+115>:   pop    %edi
0x56556573 <+116>:   ret
```

图 4.8 实现

我们发现“add -0x4(%edi, %esi, 4), %eax ; cmp %eax, (%edi, %esi, 4) ; ”的调用，前者实现的将数组中第 %esi－1 个元素的值加到 %eax 寄存器中，并将结果存入 %eax。并且和现在的数组相应位置进行对比，如果不符合，会直接爆炸

所以我们得到正确的答案应该是“输入 n0,n1,n2,n3,n4,n5”六个数字,其中要满足条件“n1=n0+1,n2=n1+2,n3=n2+3,n4=n3+4,n5=n4+5”,于是我们选择数列“0 1 3 6 10 15”成功通过

(3) 实验任务 3 的实验记录

phase_3:

我们利用 disas 命令先看一下整体代码布局，发现有一个标准输入函数，我们进入函数，发现他要求我们输入一个“%d%c%d”的字符，并且把两个 int 类型的数字储存在 esp 寄存器偏移 0x4 和偏移 0x8 的位置上，然后我们随机输入一个初始数字“6 8”，然后我们调出指令进行观察

首先，我们发现 eax 寄存器返回了输入的数值，要大于 2,所以我们输入两个数字的假想是错误的。于是我们更新一下输入为“6 8 9”，继续查看情况。其次，第一个数字也即是 esp 寄存器偏移 4 个字节的位置上的数要小于等于 7

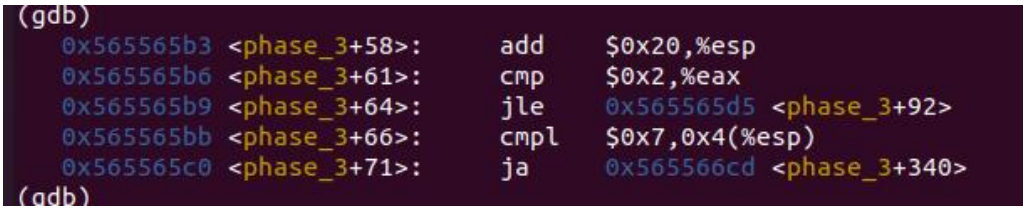

图 4.9 调试状态

在我们的样例中，避开了这一点，于是我们继续向下调试。我们发现程序会根据第一个数字的不同内容，计算得到不同的地址储存在 edx 寄存器里面，通访问地址找到目的地

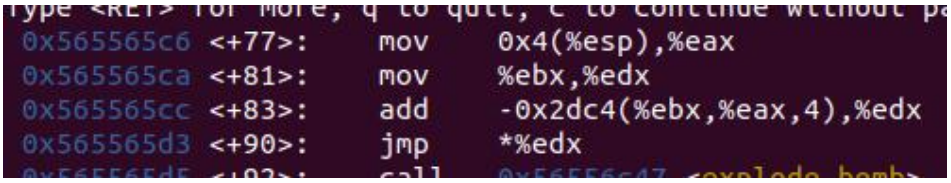

图 4.10 代码内容

通过单一的映射关系，我们可以对应第一个数字到下面的条件转移语句中，对应高级语言里面的 switch case 语句，第一个数字作为跳转依据，第二个数字作为他的一个映射结果。

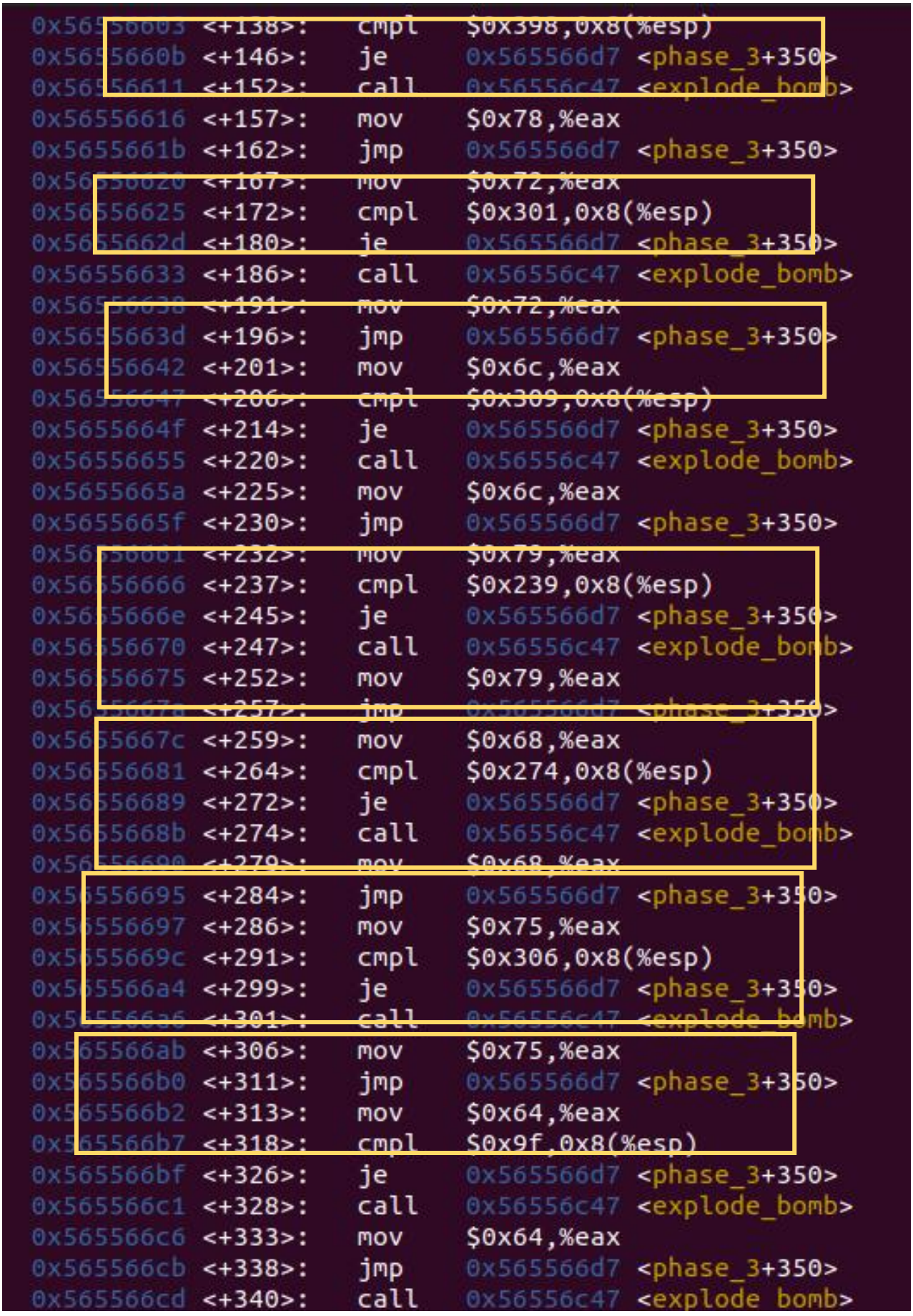
0x56556603 <+138>: cmpl $0x398,0x8(%esp)
0x5655560b <+146>: je 0x565566d7 <phase_3+350>
0x56556611 <+152>: call 0x56556c47 <explode_bomb>
0x56556616 <+157>: mov $0x78,%eax
0x5655561b <+162>: jmp 0x565566d7 <phase_3+350>
0x56556620 <+167>: mov $0x72,%eax
0x56556625 <+172>: cmpl $0x301,0x8(%esp)
0x5655562d <+180>: je 0x565566d7 <phase_3+350>
0x56556633 <+186>: call 0x56556c47 <explode_bomb>
0x56556638 <+191>: mov $0x72,%eax
0x5655563d <+196>: jmp 0x565566d7 <phase_3+350>
0x56556642 <+201>: mov $0x6c,%eax
0x56556647 <+206>: cmpl $0x309,0x8(%esp)
0x5655564f <+214>: je 0x565566d7 <phase_3+350>
0x56556655 <+220>: call 0x56556c47 <explode_bomb>
0x5655565a <+225>: mov $0x6c,%eax
0x5655565f <+230>: jmp 0x565566d7 <phase_3+350>
0x56556661 <+232>: mov $0x79,%eax
0x56556666 <+237>: cmpl $0x239,0x8(%esp)
0x5655566e <+245>: je 0x565566d7 <phase_3+350>
0x56556670 <+247>: call 0x56556c47 <explode_bomb>
0x56556675 <+252>: mov $0x79,%eax
0x5655567a <+257>: jmp 0x565566d7 <phase_3+350>
0x5655567c <+259>: mov $0x68,%eax
0x56556681 <+264>: cmpl $0x274,0x8(%esp)
0x56556689 <+272>: je 0x565566d7 <phase_3+350>
0x5655568b <+274>: call 0x56556c47 <explode_bomb>
0x56556690 <+279>: mov $0x68,%eax
0x56556695 <+284>: jmp 0x565566d7 <phase_3+350>
0x56556697 <+286>: mov $0x75,%eax
0x5655569c <+291>: cmpl $0x306,0x8(%esp)
0x565566a4 <+299>: je 0x565566d7 <phase_3+350>
0x565566a6 <+301>: call 0x56556c47 <explode_bomb>
0x565566ab <+306>: mov $0x75,%eax
0x565566b0 <+311>: jmp 0x565566d7 <phase_3+350>
0x565566b2 <+313>: mov $0x64,%eax
0x565566b7 <+318>: cmpl $0x9f,0x8(%esp)
0x565566bf <+326>: je 0x565566d7 <phase_3+350>
0x565566c1 <+328>: call 0x56556c47 <explode_bomb>
0x565566c6 <+333>: mov $0x64,%eax
0x565566cb <+338>: jmp 0x565566d7 <phase_3+350>
0x565566cd <+340>: call 0x56556c47 <explode_bomb>

图 4.11 代码内容

也同时，我们注意到这里会把%esp+0x8 和一些给定的数字进行比较，若果不相等会直接炸掉，若相等会进行下一步讨论。所以，esp+0x8 上面存的是什么数字呢我？我们打开查看，发现是我们的第三个数字

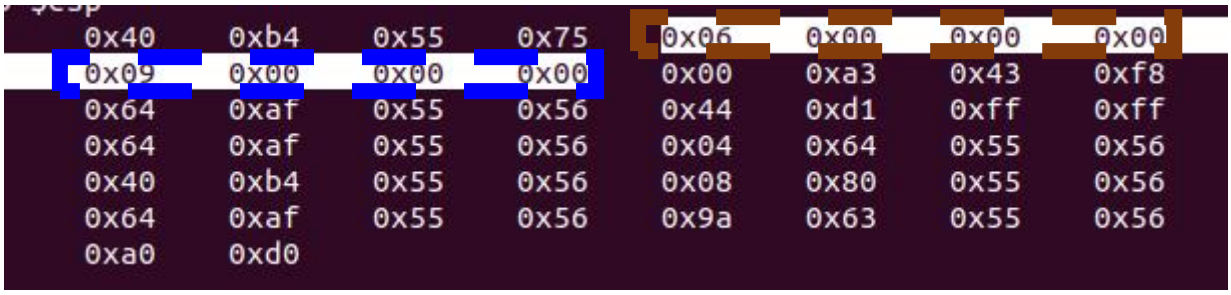

图 4.12 esp 寄存器内容

蓝色方框是 esp+0x8,褐色方框是 esp+0x4。于是，我们确定了第三个数是这里的比较数字，在下一步我们发现是把一个刚刚存在 eax 的立即数和 esp+0x3 的值进行比较，在本例中，这个立即数是 0x75，那么 eax 里面的数字是什么呢？我们进入到那个系统封装的输入语句看看，发现他会把第一个数（输入格式为十进制）放在 esp+0x4，第三个数放在 esp+0x8，而第二个数要求的是字符类型，以十六进制的方式放在 esp+0x3 的位置，也就是说，在本例里面，我们的第二个数字应该是 u（对应 ASCII 编码 0x75），所以，6 u 774 是一个答案.

图 4.13 输入函数内容(注：输入尝试为 3 6 5 774)

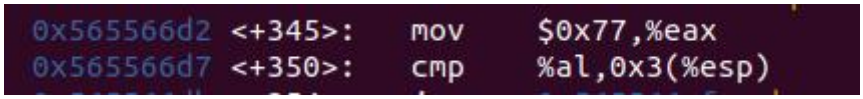

图 4.14 第二次比较

(4) 实验任务 4 的实验记录

phase_4:

我们利用 disas 命令先看一下整体代码布局，发现有一个标准输入函数，我们进入函数，发现他的输入格式是两个十进制的整数，同样的，输入返回的数量也会放在 eax 寄存器里面，

下面也有一个类似的检查返回数量的语句，我们首先输入 14 和 7 看看情况。

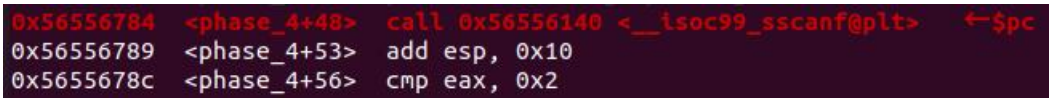

图 4.15 比较情况

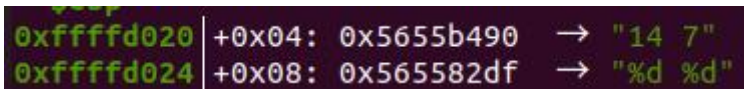

图 4.16 输入格式

图 4.16 eax 寄存器内容

当我们成功读入后检查 eax 寄存器的内容，发现其符合要比较的要求，所以我们确认了输入的格式，下面，我们又发现，有一个拿$esp+0x4 和 14 比较的语句，跳过结果是小于等于 14 很安全但是大于 14 会直接炸掉，所以我们合理猜测，$esp+0x4 是第一个数字或者第二个数字的储存地址，同样的，$esp+0x8 是第二个数字的储存地址，我们验证一下，发现真是这样子

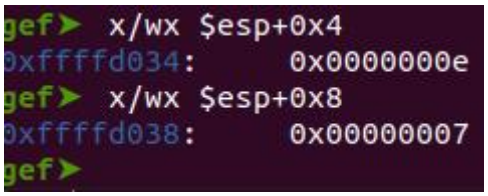

图 4.17 输入内容

然后我们继续往下看和这两个有关的数据，发现在下方有一句：

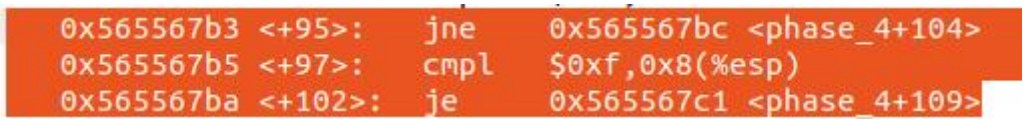

图 4.18 输入内容

我们发现这里会和 15 比较第二个数字,所以我们相信第二个数字是 15.下面我们更新数据后去看第一个数字是什么。我们发现上面有一个和 eax 和 15 比较的结果，如果不成功就会爆炸。我们认为这是在再上面的 func4 里面实现对 eax 的改变的，所以我们进入 func4 函数看看情况。我们获取 func4 函数内容后，对每一句话进行解释，发现这是一个类似二分查找的分支结构

1. `push ebx`：保存寄存器ebx的值到栈上，因为在函数中可能会修改它。
2. `sub esp,0x8`：减少栈指针esp的值，为局部变量分配8字节的空间。
3. `mov eax,DWORD PTR [esp+0x10]`：移动到eax寄存器。
4. `mov ecx,DWORD PTR [esp+0x18]`：移动到ecx寄存器。
5. `mov edx,ecx`：将ecx的值复制到edx寄存器。
6. `sub edx,DWORD PTR [esp+0x14]`：从edx中减去。
7. `mov ebx,edx`：将edx的值复制到ebx寄存器。
8. `shr ebx,0x1f`：将ebx右移31位，用于计算符号位。
9. `add ebx,edx`：将ebx和edx相加，用于计算绝对值。
10. `sar ebx,1`：将ebx算术右移1位，相当于除以2。
11. `add ebx,DWORD PTR [esp+0x14]`：将ebx和要查找的值相加。
12. `cmp ebx,eax`：比较ebx和eax的值。
13. `jg 0x56556727`：如果ebx大于eax，跳转到地址0x56556727。
14. `jl 0x5655673f`：如果ebx小于eax，跳转到地址0x5655673f。
15. `mov eax,ebx`：将ebx的值移动到eax寄存器。
16. `add esp,0x8`：恢复栈指针esp的值。
17. `pop ebx`：恢复ebx寄存器的值。

28. `sub esp,0x4`：为递归调用分配4字节的空间。
29. `push ecx`：将数组的大小压入栈中。
30. `lea edx,[ebx+0x1]`：计算新的上界索引。
31. `push edx`：将新的上界索引压入栈中。
32. `push eax`：将数组压入栈中。
33. `call 0x565566fb`：递归调用函数本身。
34. `add esp,0x10`：清理栈空间。
35. `add ebx,eax`：将递归调用的结果累加到ebx中。
36. `jmp 0x56556720`：跳转到代码的开始，继续执行。

19. `sub esp,0x4`：为递归调用分配4字节的空间。
20. `lea edx,[ebx-0x1]`：计算新的下界索引。
21. `push edx`：将新的下界索引压入栈中。
22. `push DWORD PTR [esp+0x1c]`：将数组的指针压入栈中。
23. `push eax`：将数组压入栈中。
24. `call 0x565566fb`：递归调用函数本身。
25. `add esp,0x10`：清理栈空间。
26. `add ebx,eax`：将递归调用的结果累加到ebx中。
27. `jmp 0x56556720`：跳转到代码的开始，继续执行。

图 4.19 函数内容和解释

而在更加前面，我们发现在第一次输入的时候把 0xe 和 0x0 两个立即数存入栈里面，函数里面又把 eax 和 ecx 接受了这两个数。下面的 ebx 减去 edx 操作后面还要加回来，我们相信类似于二分查找中“(r-l)/2+l”的操作，目的是获取中间位置的数字。所以，ebxx 储存的是中间数字的值。而 eax 的得到由 ebx 的累加得来，所以，经过计算，发现第一个个数字应该是 5，这样在我们每次累加之后的结果就会是 15。

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configu
47 commands loaded for GDB 9.2 using Python engine 3.8
[*] 7 commands could not be loaded, run `gef missing` to know why
Reading symbols from bomb...
gef➤  r an.txt
Starting program: /home/yuhang/桌面/bomb202315752/bomb an.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
```

图 4.20 phase_4 验证

验证成功，所以结果是 5 15。

(5) 实验任务 5 的实验记录

phase_5:

首先，我们使用 disas 命令看到 phase_4 函数的全部具体内容，我们发现有一个求解字符串长度的函数，而下面又把 eax 寄存器里面的内容和立即数 6 进行了比较，如果不为 6 会直接爆炸。所以我们有理由相信，输入会是一个字符串，而且这个字符串长度应该是 6，字符串的长度信息储存在 eax 寄存器里面。我们输入样例“abcdefg”调试断点到这一步，发现 eax

的值是 7，符合我们的猜想。黄色方框部分就是相关的内容

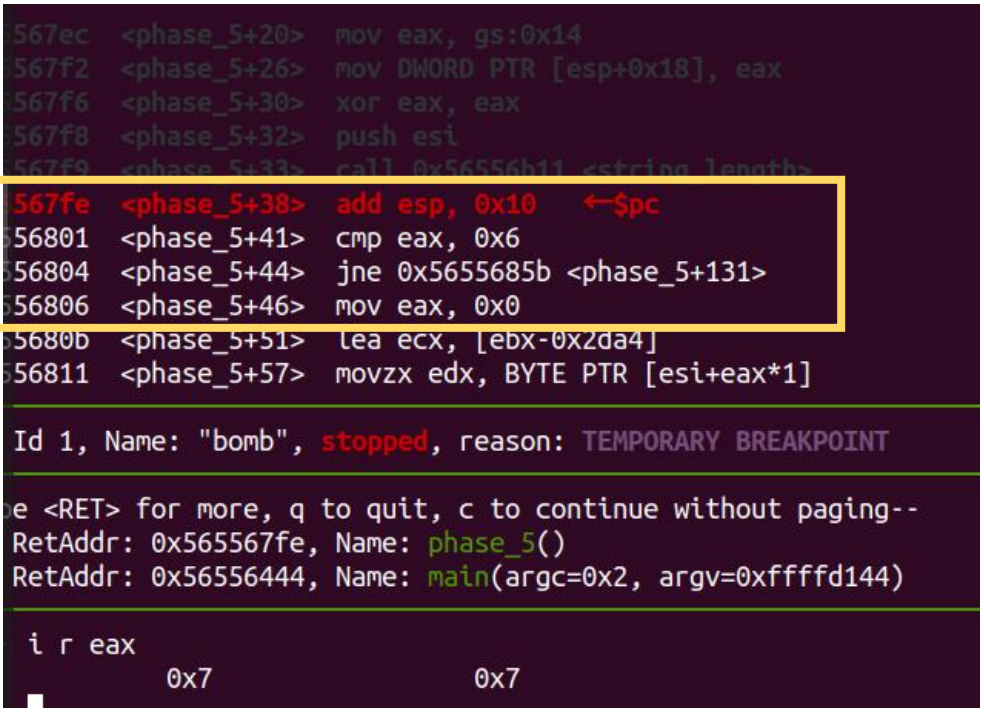

图 4.21 输入格式猜测依据

我们更换输入为“mfcdhp”进入调试。我们发现下面有一个循环结构，eax 表示计数，计数到 6 的时候自动跳出。那么在每一次循环里面发生了什么呢？

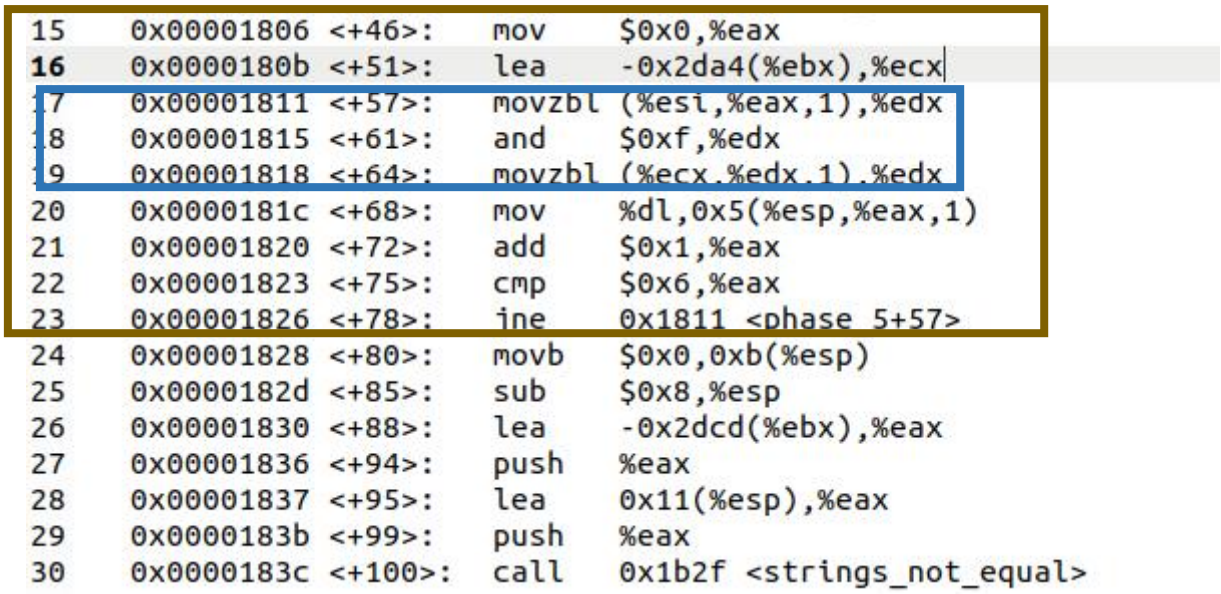

图 4.22 函数具体分析

我们看到上图，褐色方框之内的就是每一次循环的内容，我们首先看看 ecx 寄存器里面读入了什么东西，发现是一个指向字符串的地址。

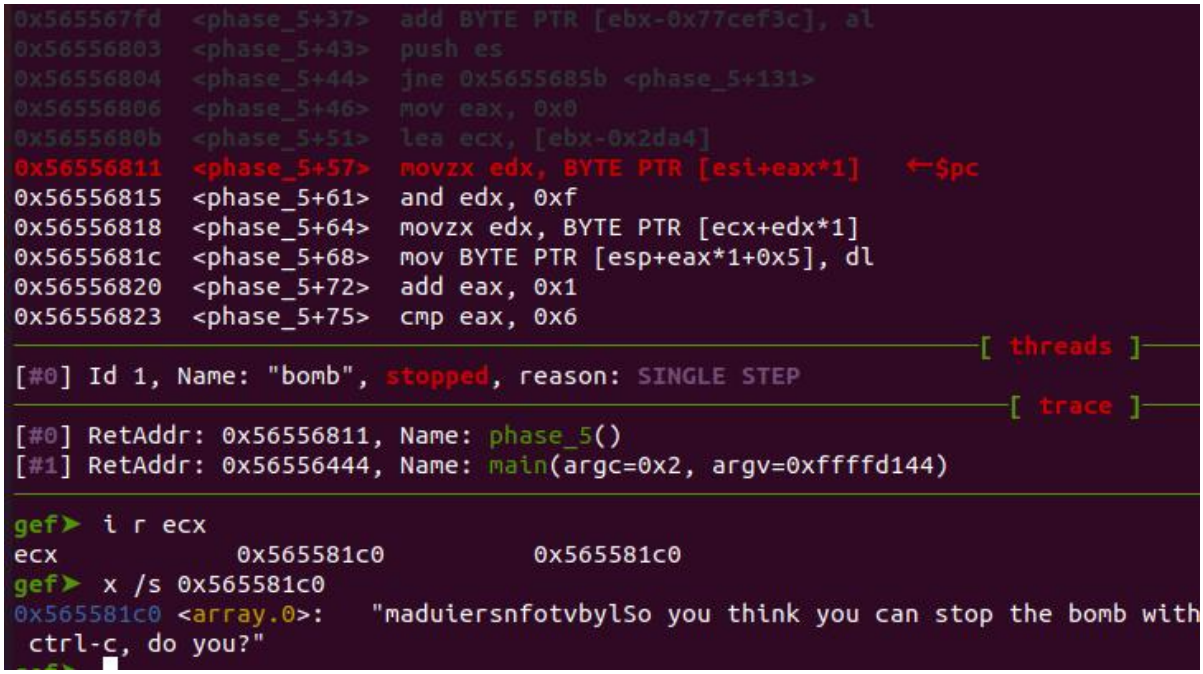

图 4.23 映射具体分析

我们已经知道，我们输入的字符串储存在 esi 寄存器里面，下面那一句话（图 4.19 蓝色方框所示），我们发现，他首先利用了循环次数作为偏移量，取出 esi 中的第 eax 号元素，这就是字符串数组的下标的操作。所以，这一步的目的是取出我们输入的字符串里面的每一位。然后，将这一个字符串的 ASCII 码进行与 15 的按位与运算，保留低四位，然后又以这个为数组下标，去寻找 ecx 字符串里面对应的字符，并赋值到 edx 寄存器里面。再在下方进行对比目标字符串。而目标字符串很好找，在比较是否相等的函数前面有两个 eax 入栈，一个是我们输入字符串映射的结果，一个是系统生成字符串，如下图所示。

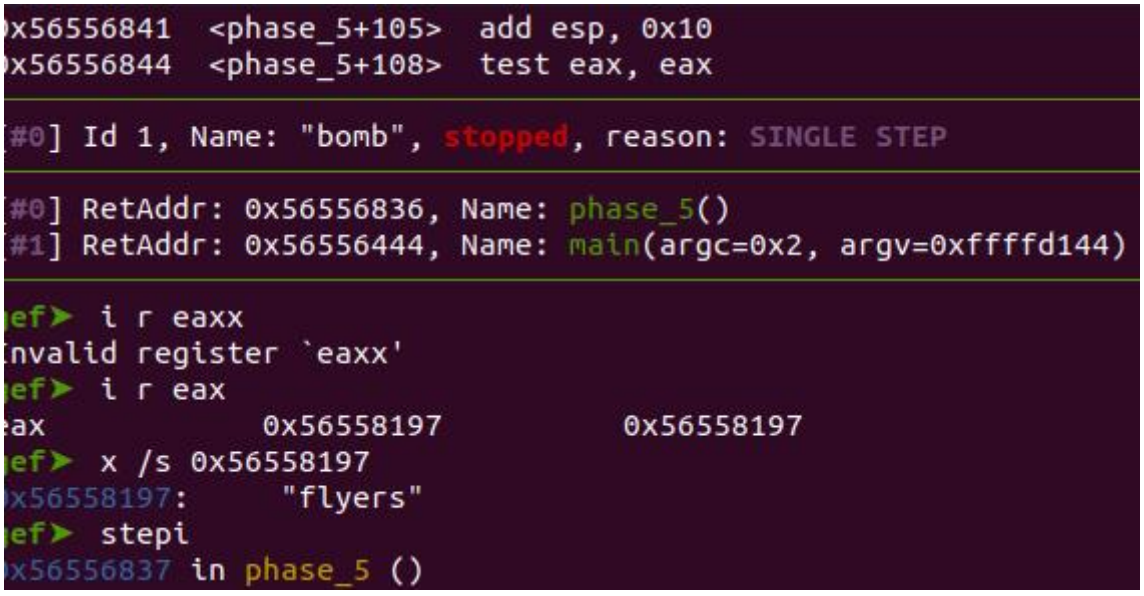

图 4.24 目标字符串具体分析

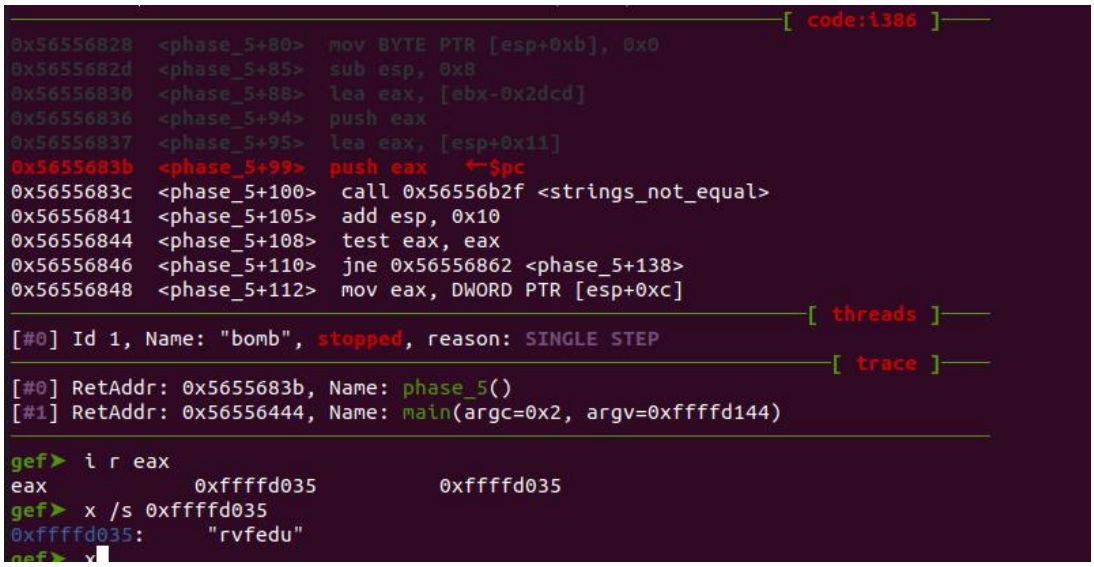

图 4.25 映射字符串初步结果

所以，下面我们要做的只是要找到几个字符串，让他们的地位进行索引，对应找到图 4.20 上的对应的字符即可。这个字符串可以有很多解，我们选取的是“yonefg”.

(6) 实验任务 6 的实验记录

phase_6:

首先我们还是使用 disas 来查看全局的情况，发现有一个“read_six_numbers”函数，我们有理由相信要求输入的是 6 个整数，我们进入到这个函数内部检查一下，果真如此，所以我们输入 1 2 3 4 5 6 看看情况.

我们一路查找遍历所有出现过的寄存器的内容，在这里，我们发现了一个类似 phase_5 里面输入的数字构成数组的检索结构，我们立刻打印原地址看看看情况，发现果然如此。ebp 寄存器里面存着我们输入的六个数字.

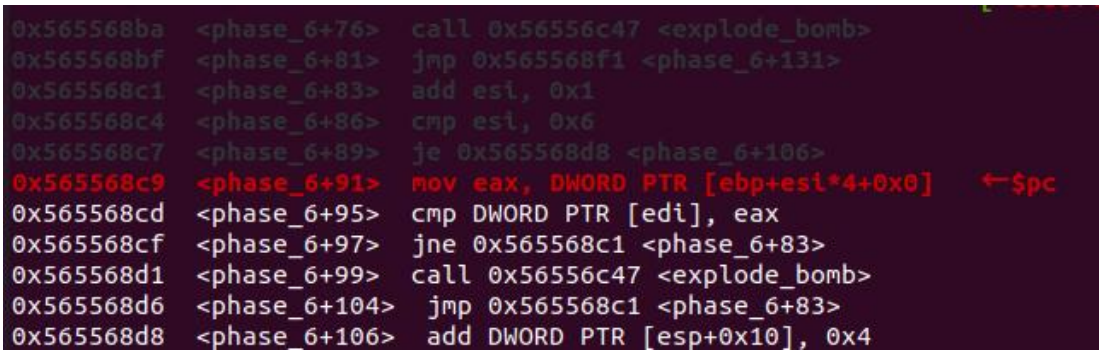

图 4.26 疑似索引结构

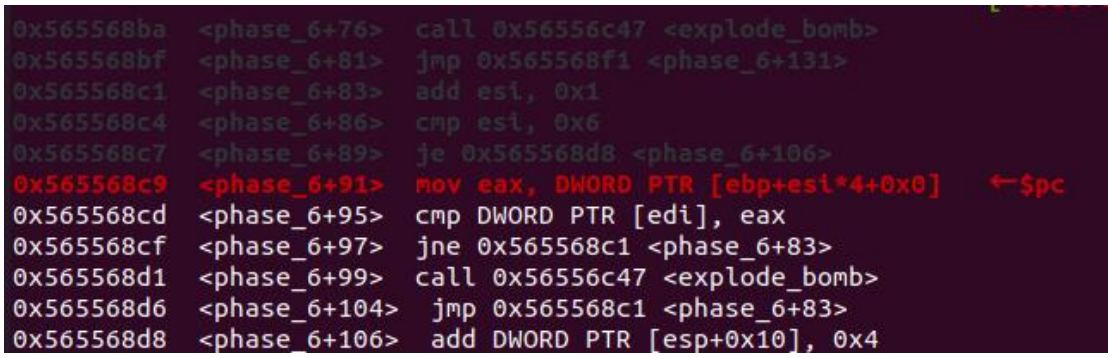

图 4.27 验证相关猜想

再下面，我们发现他要求我们比较 eax 也就是我们输入的数组的偏移量上的数字和 esi 存储地址上指向的那个数字进行对比大小，我们发现他的地址指向了 1 这个数字。这个 1 会不会就是我们的输入数字 1 呢？

我们发现，在上方，也就是在这个数组索引之前，出现过一次 esi 寄存器：

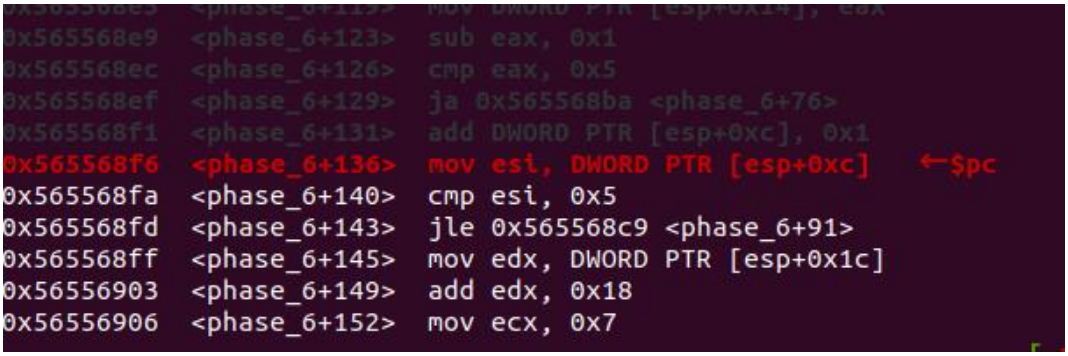

图 4.27 esi 寄存器首次出现

当然这里的 esi 寄存器接受了一个来自 esp 寄存器偏离了 0xc 位置的值，那这里又是多少呢？我们在执行完第 131 条语句之后仔细查看他的前后字节，发现：

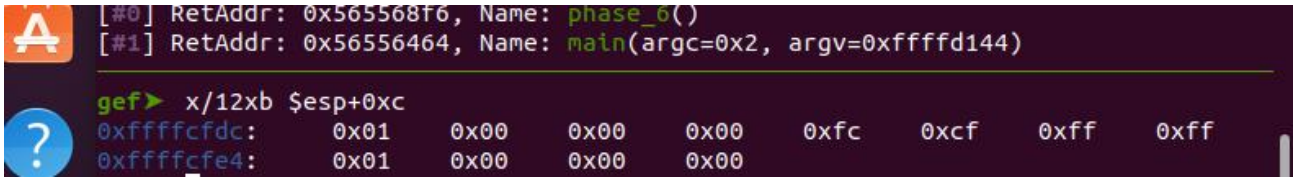

图 4.28 esi 具体情况

在这里，前四位字节一开始是 0，在加一操作之后变成 1，而中间四个字节，正好是图 4.27 里面 esi 所储存的地址，而第三个字，是我们的输入 1.

也就是说，这里是个结构体和链表，他的定义是一个“value”值，在函数中可变，一个“idx”值，也就是我们的索引，也就是我们输入的值，还有一个是“address”也就是下一个结构体的地址，具体定义如图所示：

```
typedef struct nodeStruct
{
    int value;
    int index;
    struct nodeStruct *next;
} listNode;
```

图 4.27 结构体内容定义

我们继续往下看这段代码:

```
0x00001892 <+36>:    mov    %eax,0x24(%esp)
0x00001896 <+40>:    push   %eax
0x00001897 <+41>:    pushl  0x8c(%esp)
0x0000189e <+48>:    call   0x1c7c <read_six_numbers>
0x000018a3 <+53>:    mov    %edi,0x28(%esp)
0x000018a7 <+57>:    add    $0x10,%esp
0x000018aa <+60>:    mov    %edi,0x10(%esp)
0x000018ae <+64>:    movl   $0x0,0xc(%esp)
0x000018b6 <+72>:    mov    %edi,%ebp
0x000018b8 <+74>:    jmp    0x18dd <phase_6+111>
0x000018ba <+76>:    call   0x1c47 <explode_bomb>
0x000018bf <+81>:    jmp    0x18f1 <phase_6+131>
0x000018c1 <+83>:    add    $0x1,%esi
0x000018c4 <+86>:    cmp    $0x6,%esi
0x000018c7 <+89>:    je     0x18d8 <phase_6+106>
0x000018c9 <+91>:    mov    0x0(%ebp,%esi,4),%eax
0x000018cd <+95>:    cmp    %eax,(%edi)
0x000018cf <+97>:    jne    0x18c1 <phase_6+83>
0x000018d1 <+99>:    call   0x1c47 <explode_bomb>
0x000018d6 <+104>:   jmp    0x18c1 <phase_6+83>
0x000018d8 <+106>:   addl   $0x4,0x10(%esp)
0x000018dd <+111>:   mov    0x10(%esp),%eax
0x000018e1 <+115>:   mov    %eax,%edi
0x000018e3 <+117>:   mov    (%eax),%eax
0x000018e5 <+119>:   mov    %eax,0x14(%esp)
0x000018e9 <+123>:   sub    $0x1,%eax
0x000018ec <+126>:   cmp    $0x5,%eax
0x000018ef <+129>:   ja     0x18ba <phase_6+76>
0x000018f1 <+131>:   addl   $0x1,0xc(%esp)
0x000018f6 <+136>:   mov    0xc(%esp),%esi
0x000018fa <+140>:   cmp    $0x5,%esi
0x000018fd <+143>:   jle    0x18c9 <phase_6+91>
```

图 4.28 第一个条件检验

这段代码表示出了两段循环，循环是嵌套的，最外层是次数循环，表示内层循环的次数，一共进行六次，次数储存在 esi 寄存器里面，当小于等于 5 的时候会把循环进行下去。那么内层循环实现了什么呢？

上面我们已经发现，我们输入的每个数字都储存在$esp+0x10 的位置，而在内层循环里面，

由+111 语句我们发现，我们输入的数值会先传给 eax 寄存器，eax 自己减少一个 1 之后和 5 进行比较，如果大于 5 会直接爆炸，所以我们输入的数字不可以大于 5.然后我们发现在前面，我们已经把当前值储存在 eax 寄存器的情况下，会之歌把后面的每一个数字用 edi 寄存器保存下来，然后进行与 eax 里面的数字进行比较，如果数字相同，也会爆炸。所以，这一步的要求实际上是要我们输入 6 个完全不同的数字，而且每个数字不可以大于 6.

我们继续看下一步的代码又干了什么。关注这一段代码：

```
0x000018ff <+145>:   mov    0x1c(%esp),%edx
0x00001903 <+149>:   add    $0x18,%edx
0x00001906 <+152>:   mov    $0x7,%ecx
0x0000190b <+157>:   mov    0x18(%esp),%eax
0x0000190f <+161>:   mov    %ecx,%esi
0x00001911 <+163>:   sub    (%eax),%esi
0x00001913 <+165>:   mov    %esi,(%eax)
0x00001915 <+167>:   add    $0x4,%eax
0x00001918 <+170>:   cmp    %eax,%edx
0x0000191a <+172>:   jne    0x190f <phase_6+161>
```

图 4.29 第二个条件检验

很明显，这里做的是把我们输入的数字的地址取出来，然后把立即数放在 0x7 里面的 ecx，去减对应的数字，也就是把对应的数字在 7 之内反转了一遍。

我们继续看第三个条件代码：

```
0x0000191c <+174>:   mov    $0x0,%esi
0x00001921 <+179>:   mov    %esi,%edi
0x00001923 <+181>:   mov    0x2c(%esp,%esi,4),%ecx
0x00001927 <+185>:   mov    $0x1,%eax
0x0000192c <+190>:   lea    0x168(%ebx),%edx
0x00001932 <+196>:   cmp    $0x1,%ecx
0x00001935 <+199>:   jle    0x1941 <phase_6+211>
0x00001937 <+201>:   mov    0x8(%edx),%edx
0x0000193a <+204>:   add    $0x1,%eax
0x0000193d <+207>:   cmp    %ecx,%eax
0x0000193f <+209>:   jne    0x1937 <phase_6+201>
0x00001941 <+211>:   mov    %edx,0x44(%esp,%edi,4)
0x00001945 <+215>:   add    $0x1,%esi
0x00001948 <+218>:   cmp    $0x6,%esi
0x0000194b <+221>:   jne    0x1921 <phase_6+179>
```

图 4.30 第三个条件检验

我们首先打印$esp+0x2c 的内容看看。我们发现刚刚是我们第二轮检验里面用 7 去每个相减的值。把那些值逐个传到 ecx 寄存器

```
gef➤  x/28xb $esp+0x2c
0xffffcffc:     0x06    0x00    0x00    0x00    0x05    0x00    0x00    0x00
0xffffd004:     0x04    0x00    0x00    0x00    0x03    0x00    0x00    0x00
0xffffd00c:     0x02    0x00    0x00    0x00    0x01    0x00    0x00    0x00
0xffffd014:     0x50    0x00    0x00    0x00
gef➤
```

图 4.31 $esp+0x2c 里面的内容

我们下面为了验证这个 edx 里面传进去的到底是什么，采用了对照实验法，用另外一个样例“1 3 2 4 6 5”打开 gdb 进行比较实验，前两个检验不影响我们对新样例的理解。我们发现，edx 传进去了一个结构体数组的首地址，但是这个结构体数组中的 idx 并不和我们输入的数值有关，是完全的从 1 到 6 的顺序排列（注：左边为“1 3 2 4 6 5”样例，右边为“1 2 3 4 5 6”样例，下同）

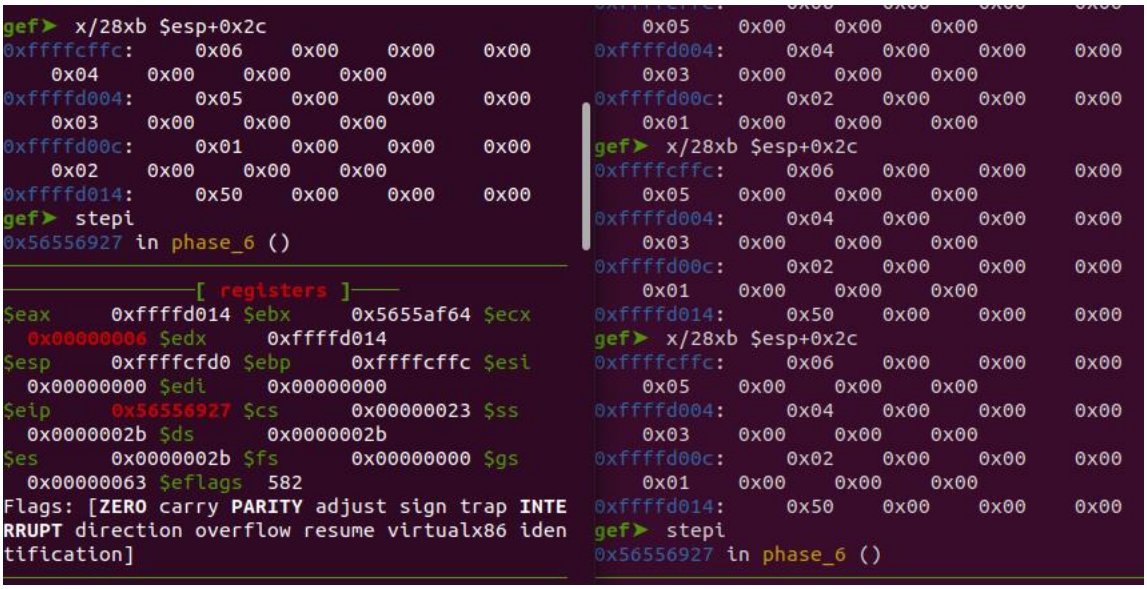

图 4.32 第二步检验后的情况

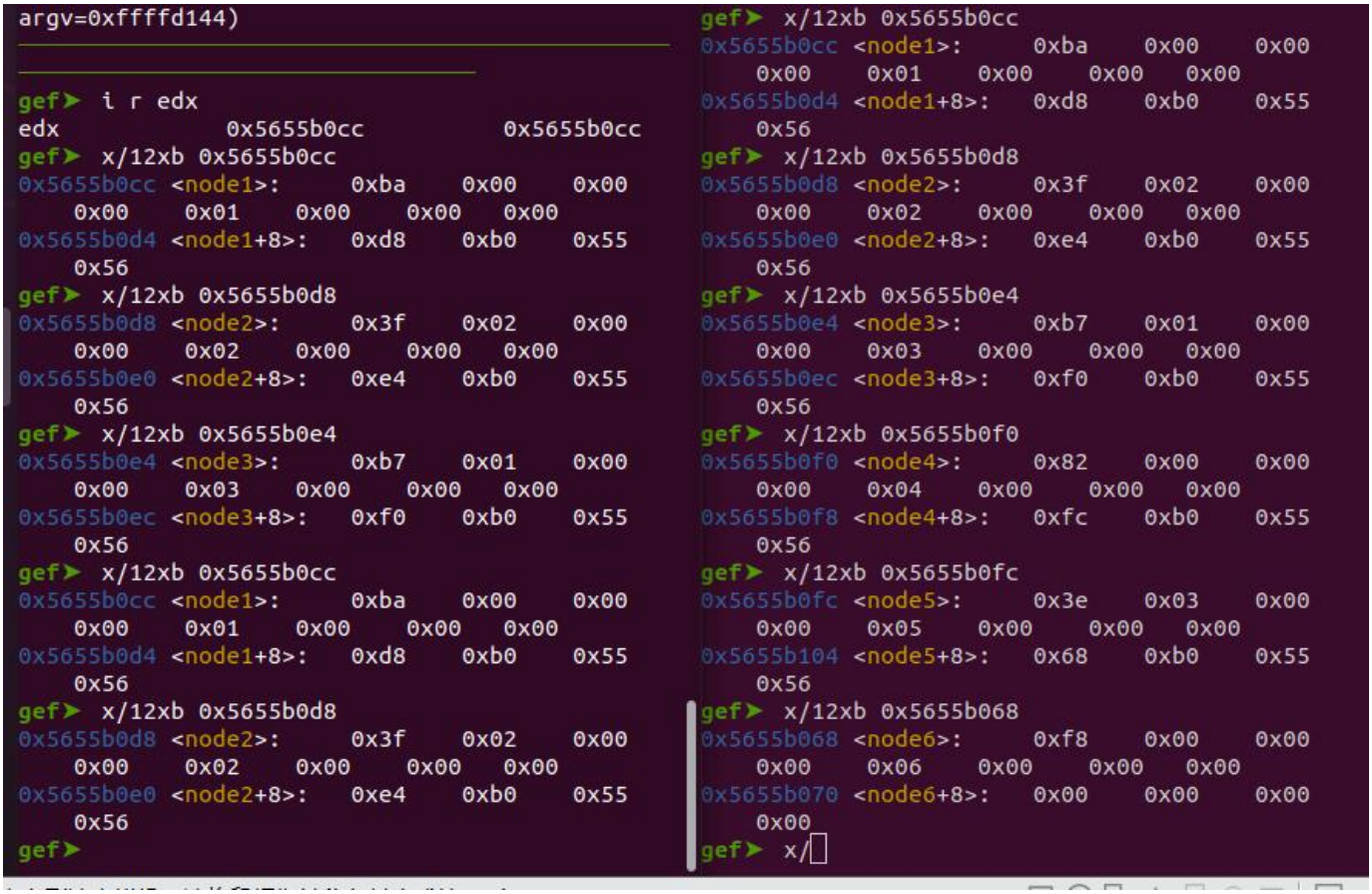

图 4.33 对比情况

我们继续看下面的代码有一个“0x00001937 <+201>: mov ，0x8(%edx),%edx”这句话里面我们已经知道了 edx 寄存器里是我们那个固定结构体数组的首地址的位置。所以，这一步是把下一个节点的地址。

下面，我们会把 eax 的值加 1，需要注意这里的 eax 的值来自立即数。然后和我们的输入的数字进行对比，看看是不是相等。如果不相等，会把那个固定的结构体数组往后移动，同时 eax 递增，再次比较，再次说明，ecx 是我们输入的值被 7 反转的值，eax 在每次从 1 开始递增。

在第一循环里面，我们发现 edi（来自 esi，esi 来自立即数 0）的值始终是 0

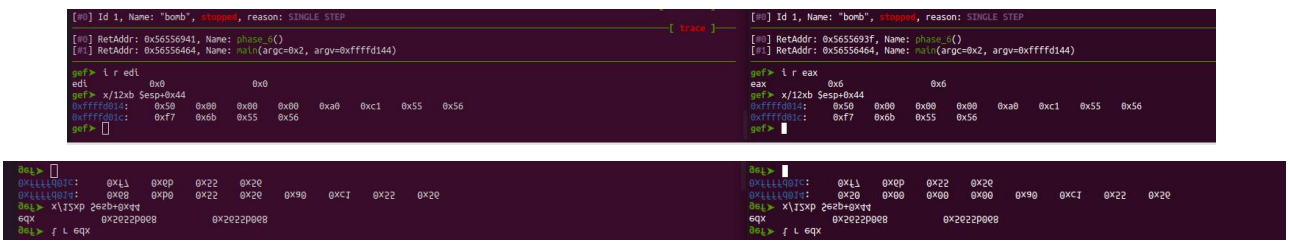

图 4.34 第一次循环后 esp+0x44 的位置情况

接下来，esi 会进行自增，所以 esi 是这一段循环的指示器，一共 6 次循环。同时给 edi 传值，edi 是数组下标。注意，每次进来 edx 都会初始化一次地址，他的偏移和+201 的来到次数有关，也就是和 eax 与 ecx 的值有关，而 ecx 每次来自我们输入的 7 的反转。到目前为止，我们搞清楚了，到底 174 到 221 是在做什么。完成第三阶段。

下面我们进行第四阶段的记录，首先的第一句话就是 esp+0x44 地址的操作，我们先看看这里是什么。

```
eax            0x5655b0fc          0x5655b0fc
gef➤  x/24xb $esp+0x44
0xffffd014:     0x68    0xb0    0x55    0x56    0xfc    0xb0    0x55    0x56
0xffffd01c:     0xf0    0xb0    0x55    0x56    0xe4    0xb0    0x55    0x56
0xffffd024:     0xd8    0xb0    0x55    0x56    0xcc    0xb0    0x55    0x56
gef➤
```

图 4.34 第四阶段开启的时候 esp+0x44 的位置情况

发现是固定数组的几位地址,而且是按照 idx 位置的倒叙排列。下面我们发现，esi,eax 和 edx 寄存器分别存了上图前三个地址,到 234 步为止，node6 在 esi，node5 在 eax.在 231 步，我们发现，他把原来 node6+8 位置上原本空地址改成了 node5 的地址。在 234 步，我们把 node4 的地址放在了 edx 寄存器.在 238，用 node4 的地址改变了 node5 原本指向 node6 的地址位，重定向到 node4，241 步，把 node3 传到 eax；245，重定向 node3 的地址作为 node4 的地址位置；248，node2 指向 edx；252，重定向把 node2 的地址作为 node3 的地址位置，后续同理，直到 262 结束，我们实现了固定数组的反转操作。269 把立即数传到 edi，跳到 284。284 把 node5 的地址传到 eax，我们发现，那是因为 esi 的地址是 node6,+8 必然是 node5.

接下来到了取值操作。289 实现了 node5 的取值到 eax 操作，是 0x33e。再下一步是要求 node5 的值要大于等于 node6 的值。再下面进行了一次循环，把固定结点的顺序按照倒序排列（如图 4.35 的蓝色方框部分）。

至此，我们确定，第一个四字节是 value，第二个四字节是 idx，第三个四字节是 next_address。

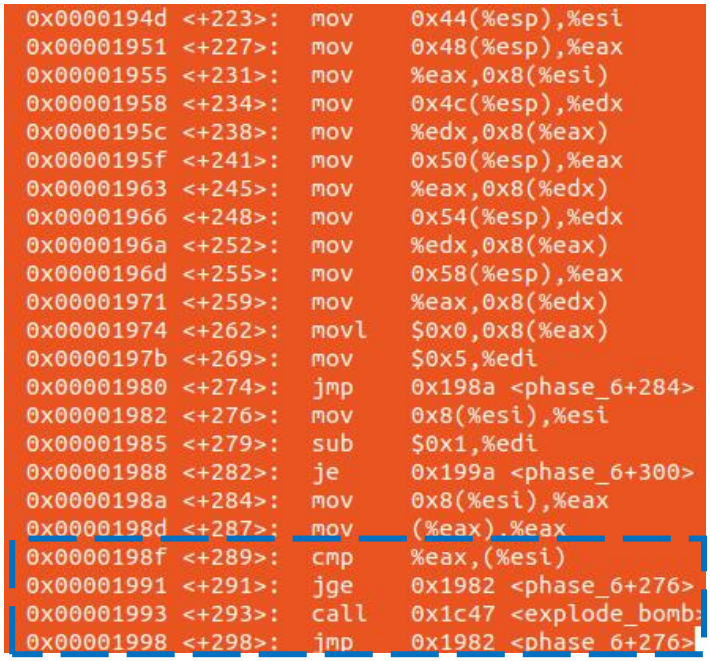

图 4.35 第四阶段代码

接下来，便是程序的结束。但是我们的问题是，我们的输入到底是什么？他和这个系统给出来的固定结构体有什么联系？这个时候，我们回头看第三阶段，我们一开始认为那个阶段什么也没干，实际上，他把我们输入的数字作为了 idx，对应了系统给出来的那个结构体数组的 value，而我们的 value 的值要求逆序输出，所以此时的输入应该是“5 2 3 6 1 4”（对应图 4.33），但是在第二阶段，系统用 7 减掉了我们的输入，然后在进入第三阶段，所以我们的输入应该是“2 5 4 1 6 3”，验证正确

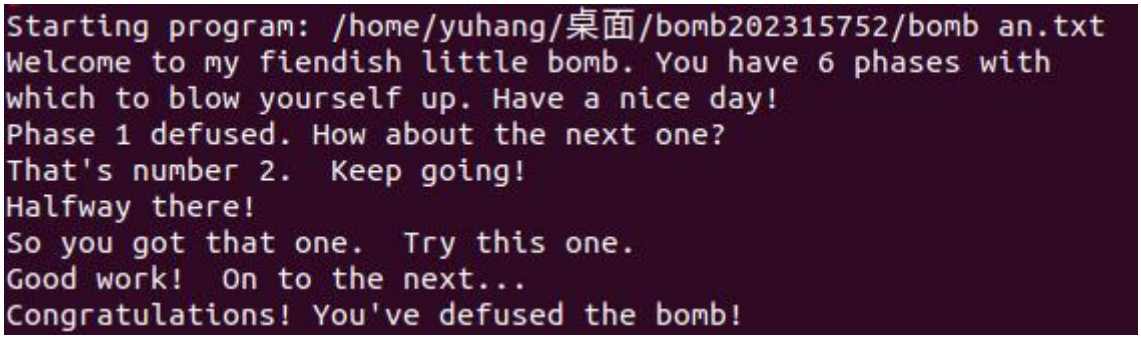

图 4.35 第四阶段和全程验证结果

(7) 实验任务 secret 的实验记录

phase_secret:

main 函数在进行完 phase_6 之后，我们发现他进入一个叫做 phase_defused 函数

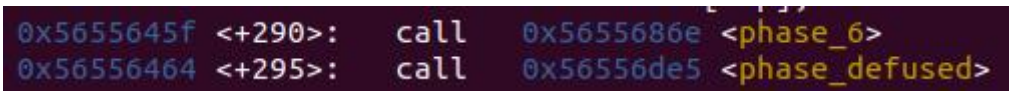

图 4.36 phase_defused 函数出现

在第 9 步，获得了 ebx 的地址是 0x5655af64,而在第 21 步，我们把 eax 储存的值转移，这个值在 esp 里面是这个，出现了 ebx 的地址：

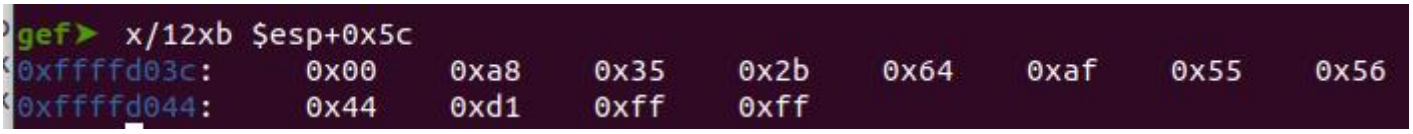

图 4.37

在第 9 步，获得了 ebx 的地址是 0x5655af64,而在第 21 步，我们把 eax 储存的值转移，这个值在 esp 里面是这个，出现了 ebx 的地址。27 步，我们发现出现了一个新函数：“num_input_strings”，这一步获取一个函数值为“1”，他会和 6 进行比较，但是我们仍然不知道这个式子要说的是什么，但是我们怀疑，这里是一个循环的指数器。

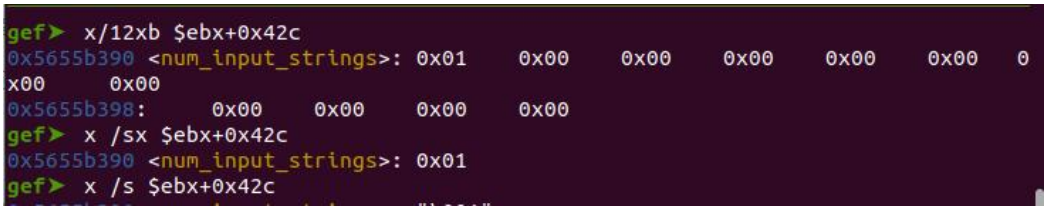

图 4.38 第 27 步内容

由于我们发现这个“phase_defused”在每一个 phase 里面都会执行一次，我们发现，直到 phase_6 的 phase_defused 函数，这个函数才会运行到这个函数的第 56 步，之前的所有步数在第 53 步全部停止。

第 61 和 65 步，eax 的值为 0xffffcfec，存入栈；66 和 70，eax 的值为 0xffffcfe8，也入栈；后面又把 0xffffcfe4,0x56558339,0x5655b490 入栈，而再下面，是一个输入函数，输入什么呢？我们进入函数，发现是要输入“一个十进制数字，一个十进制数，一个字符串”的格式。而那两个输入的十进制已经告诉我们是我们在第四关输入的两个数字 5 和 15

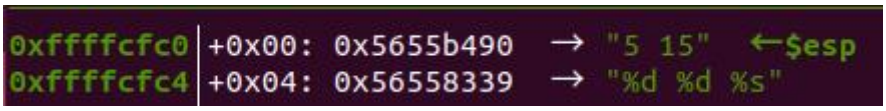

图 4.39 输入是什么格式

我们重新在 phase_4 的结果中加入了一个 test，重新运行看看情况，发现他存进来了，test 字符串存在了 0xffffcfec 里面。

后面，又把 0x56558342，然后和“test”比较，所以这里肯定有相关内容，打印，发现了目标字符：

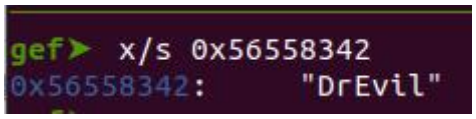

图 4.40 输入是什么格式

自此，我们找到了所有的目标语句，重运行，准备进入秘密关卡

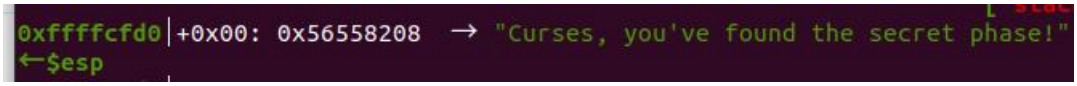

图 4.41 秘密关卡

首先，我们发现他会读入一行串“read_line”，我们输入 50 作为我们第一次的样例试试，他存在 eax 指向的 0x5655b580 地址里面,eax 入栈

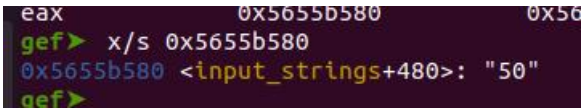

图 4.42 储存位置

再下面，我们发现调用了 strtol 函数，可见他把我们的输入变成了一个数字.我们发现，他以 16 进制的方式存到了 eax 寄存器里面，又放在了 esi 里面，为 0x32。下面来了一个比较，把 0x32 从地址去出来还减去 1，然后和 1000 对比，所以我们输入的数字要小于等于 1001。

我们接下来还是用 esi 里面的 0x32 进行操作。入栈，进入 fun7。

但是首先我们打印多几个看看 eax 里面到底是什么，到底是什么。

我们发现，eax 里面出现了一个 n1 的节点，我们猜测 n 是 node 的意思，我们发现，如果我们打印 16 个字节，他会一直在 n1,但是一旦我们打印 20 个字节，就会出现 n21，甚至 n21，我们怀疑这又是一个链表，甚至是二叉树。

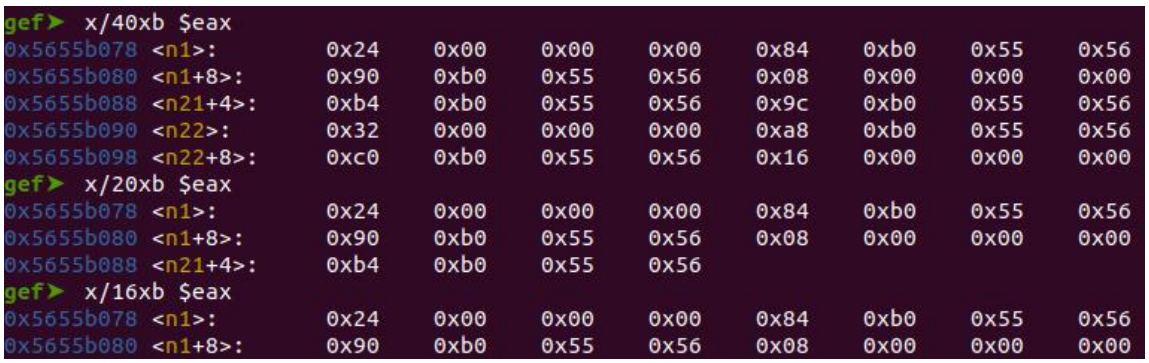

图 4.43 二叉树初现

我们强烈怀疑是二叉树，所以我们依照二叉树的思想，不断打印他的左右子树：

Node	val	left	right
N1	36	N21	N22
N21	8	N31	N32
N22	50	N33	N34
N31	6	N41	N42
N32	22	N43	N44
N33	45	N45	N46
N34	107	N47	N48
N41	1	NULL	NULL
N42	7	NULL	NULL
N43	20	NULL	NULL
N44	35	NULL	NULL
N45	40	NULL	NULL

N46	47	NULL	NULL
N47	99	NULL	NULL
N48	1001	NULL	NULL

图 4.43 二叉树表格

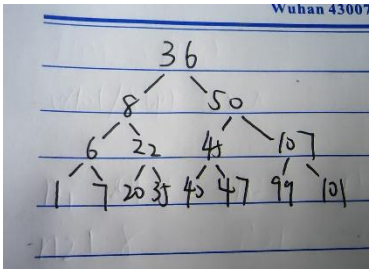

图 4.44 二叉树图示

进入 fun7,然后将 num 值和 node 的值 val 进行比较。若 node 的 val 大于 num,根据 44-52 行，将调用 fun7(node->left,num)，这种情况下返回值*2。若 node->val 等于 num，则返回 0。若 node->val 小于 num，则调用 fun7(node->right,num)，且在这种情况下返回值*2+1。

当 fun7 结束之后，将返回值与 1 进行比较。这里存在着一个类似二叉树查找的过程从最底层返回-1，相等返回 0，从左下角返回则返回值*2，从右下角返回则返回值*2+1。经过计算，这里笔者直接选择输入 107，最后返回 1。解决了 secret_phase。

```
gef➤  r an.txt
Starting program: /home/yuhang/桌面/bomb202315752/bomb an.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
[Inferior 1 (process 4800) exited normally]
```

图 4.45 所有代码解决

（2）拆除炸弹的过程中关键操作

拆炸弹的时候看着一堆的汇编代码，没有 c 语言的直观，如果没有一些敏锐性的话，会陷入混乱中，我们需要掌握一些关键的地方。

1. 注意 0x30,’0’的 ascii 码是 0x30, 我们输入的数字和字符串都会被当作字符串存储起来，后续处理提取成数字的时候如果出现了-0x30(%寄存器)这个就是应该将我们输入的字符串转换为数字，方便后续操作。
2. 进入一个新的关的时候，要先总体全局看一下大致的函数调用情况， push 了几个参数，每次我们都可以输出这几个参数，提前对接下来要运行的函数有个大概的了解。
3. 一定要边看汇编代码，边记录，由于局部变量都是通过类似-0x1c(%ebp)等方式去访问，这样如果多几个变量就会不直观，我们如果不记录下来这个地方存储了什么，接下来根本不知道对什么内容进行了操作，最好是 get 到一个点就记录一个点。

四、体会

在进行二进制炸弹拆除实验的过程中,我深刻领悟到了对计算机系统底层运作的理解和操作技巧的重要性。以下是我的一些心得体会：

首先，我重新从新的方面认识到了汇编代码的重要性。在面对没有源代码只有可执行文件的情况下，充分利用反汇编功能并理解汇编代码的细节，让我能够反向推断出源代码的思路和想法。这次实验让我更加明白了汇编代码在理解计算机系统中的关键作用。

其次，理解程序逻辑的关键性也是我在实验中的重要收获。仔细分析程序的逻辑结构，理解每一步操作的含义和作用，尤其是对于树形结构等复杂逻辑的理解，直接影响了我解决问题的效率和准确性。

另外，注重细节和耐心也是我在实验中不断强调的重点。汇编代码繁杂，很容易漏看一些关键节点，因此在研究代码时，我始终保持心平气和，一步步深入分析，以免遗漏重要信息。

灵活运用调试工具如 GDB 是解决问题的关键。通过 GDB，我能够快速定位问题所在，查看地址对应值，观察寄存器状态等，大大简化了解决问题的过程，提高了效率。

与同学交流学习也是我在实验中的重要策略。通过积极交流，我学到了新的知识和技巧，提升了团队合作能力，解决问题的效率也得到了提升。

最后,总结经验教训是巩固学习成果的关键一环。通过总结实验中遇到的问题和解决方法，我能够更好地应对类似的挑战，并不断提升自己的技能水平。

通过这次二进制炸弹拆除实验，我不仅深入理解了汇编代码的工作原理，反汇编的重要性和调试工具的使用方法，还提升了自己的问题解决能力和团队合作意识。这些经验对我今后的学习和工作都将产生积极的影响。