《计算机系统》

BombLab 实验报告

目录

1 实验项目一

1.1 项目名称

BombLab.

1.2 实验目的

根据给定的文件包，通过调试、逆向工程、运行字符串、反编译等方式，阅读源代码，找到六个问题和一个隐藏问题的答案，完成拆炸弹任务。

1.3 实验资源

给定文件包，ubuntu12，gdb，gedit，vim。

2 实验任务

```
17    /* Hmm...  Six phases must be more secure than one phase! */
18    input = read_line();             /* Get input                  */
19    phase_1(input);                  /* Run the phase              */
20    phase_defused();                 /* Drat!  They figured it out!
21                                     * Let me know how they did it. */
22    printf("Phase 1 defused. How about the next one?\n");
23
24    /* The second phase is harder.  No one will ever figure out
25     * how to defuse this... */
26    input = read_line();
27    phase_2(input);
28    phase_defused();
29    printf("That's number 2.  Keep going!\n");
```

主函数逻辑大致相同，读取输入字符串，调用 phase_n 函数，如果通过测试进行下一个，否则炸弹爆炸。

2.1 phase_1 字符串比较

```
292  8048a8a:    c7 04 24 10 a1 04 08    movl   $0x804a110,(%esp)
293  8048a91:    e8 6a fd ff ff          call   8048800 <puts@plt>
294  8048a96:    e8 62 06 00 00          call   80490fd <read_line>
295  8048a9b:    89 04 24                mov    %eax,(%esp)
296  8048a9e:    e8 ad 00 00 00          call   8048b50 <phase_1>
297  8048aa3:    e8 b3 07 00 00          call   804925b <phase_defused>
```

观察主函数的反汇编代码，可以发现在调用函数 phase_1 前将寄存器%eax 的值传到了%esp 存的地址的位置。现在还不知道这个位置存的是什么。292-294 行用于打印字符串和输入。

```
341 08048b50 <phase_1>:
342  8048b50:    83 ec 1c                sub    $0x1c,%esp
343  8048b53:    c7 44 24 04 8c a1 04    movl   $0x804a18c,0x4(%esp)
344  8048b5a:    08
345  8048b5b:    8b 44 24 20             mov    0x20(%esp),%eax
346  8048b5f:    89 04 24                mov    %eax,(%esp)
347  8048b62:    e8 5d 04 00 00          call   8048fc4 <strings_not_equal>
348  8048b67:    85 c0                   test   %eax,%eax
349  8048b69:    74 05                   je     8048b70 <phase_1+0x20>
350  8048b6b:    e8 66 05 00 00          call   80490d6 <explode_bomb>
351  8048b70:    83 c4 1c                add    $0x1c,%esp
352  8048b73:    c3                      ret
```

观察 phase_1 的反汇编代码部分，可以发现函数开辟了一个 0x1c 的新空间，而后在%esp+4 的位置存储了一个立即数（地址）。此时加上 call 时保存地址的四个字节，上述存在旧%esp 中的内容地址刚好与新%esp 相差 0x20，将该内容再次存入新%esp 指向的地址中，然后调用函数进行比较，退出函数后检查%eax 的值是否为 0，如果为 0，成功，不 bomb，收回空间并退出。

此时根据过程我们已经可以推测出，新%esp 和新%esp+4 的位置分别存储了两个地址，

分别指向输入字符串和内存中存储的字符串，而新%esp+4 位置的地址是用立即数直接给出的，该位置为所存储字符串。

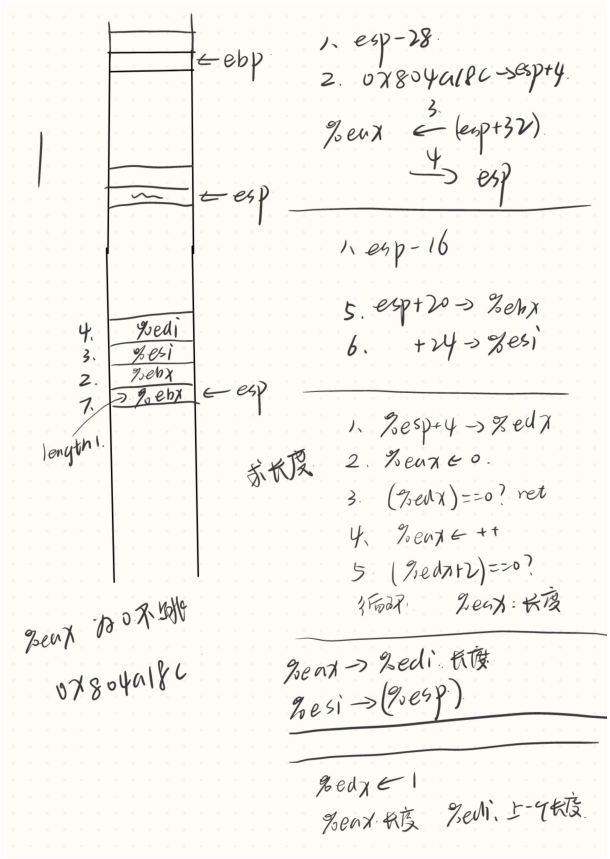
```
(gdb) b main
Breakpoint 1 at 0x80489e4: file bomb.c, line 37.
(gdb) r
Starting program: /home/guo/bomb18_202208040204/bomb

Breakpoint 1, main (argc=1, argv=0xbffff3e4) at bomb.c:37
37      {
(gdb) x/s 0x804a18c
0x804a18c:       "I can see Russia from my house!"
(gdb) q
```

```
guo@ubuntu:~/bomb18_202208040204$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
```

使用 gdb 查看内存中对应位置的值，得到该字符串，测试并通过。

```
5 08048fc4 <strings_not_equal>:
4  8048fc4:       83 ec 10                sub    $0x10,%esp
3  8048fc7:       89 5c 24 04             mov    %ebx,0x4(%esp)
2  8048fcb:       89 74 24 08             mov    %esi,0x8(%esp)
1  8048fcf:       89 7c 24 0c             mov    %edi,0xc(%esp)
0  8048fd3:       8b 5c 24 14             mov    0x14(%esp),%ebx
1  8048fd7:       8b 74 24 18             mov    0x18(%esp),%esi
2  8048fdb:       89 1c 24                mov    %ebx,(%esp)
3  8048fde:       e8 c8 ff ff ff          call   8048fab <string_length>
4  8048fe3:       89 c7                   mov    %eax,%edi
5  8048fe5:       89 34 24                mov    %esi,(%esp)
6  8048fe8:       e8 be ff ff ff          call   8048fab <string_length>
7  8048fed:       ba 01 00 00 00          mov    $0x1,%edx
8  8048ff2:       39 c7                   cmp    %eax,%edi
```

我们还可以看一下比较函数来验证猜测，比较函数中，先开辟一个 0x10 的空间，然后保存三个被调用者保存寄存器的值，记住这里有 call 使用的 4 个字节，因此上文中新%esp 和新%esp+4 的位置分别对应偏移量 0x14 和 0x18，将这两个位置存储至%ebx 和%edi，下文中调用其他函数比如长度函数进行二者的比较。与猜测对应。



简易栈如图所示。

2.2 phase_2 循环/数组

```
1  8048aa8:       c7 04 24 3c a1 04 08    movl   $0x804a13c,(%esp)
2  8048aaf:       e8 4c fd ff ff          call   8048800 <puts@plt>
3  8048ab4:       e8 44 06 00 00          call   80490fd <read_line>
4  8048ab9:       89 04 24                mov    %eax,(%esp)
5  8048abc:       e8 b3 00 00 00          call   8048b74 <phase_2>
6  8048ac1:       e8 95 07 00 00          call   804925b <phase_defused>
```

与 phase_1 相同，把输入字符串地址存储在%esp 的位置。

```
12 08048b74 <phase_2>:
11  8048b74:       56                      push   %esi
10  8048b75:       53                      push   %ebx
 9  8048b76:       83 ec 34                sub    $0x34,%esp
```

保存被调用者保存寄存器的值，开辟 0x34 的新空间。

```
 8  8048b79:       8d 44 24 18             lea    0x18(%esp),%eax
 7  8048b7d:       89 44 24 04             mov    %eax,0x4(%esp)
```

将%esp+0x18 的位置存到%esp+4。

```
 6  8048b81:       8b 44 24 40             mov    0x40(%esp),%eax
 5  8048b85:       89 04 24                mov    %eax,(%esp)
```

将%esp+0x40 的位置存到%esp，由上述可知，call 使用 4 字节，push 了两个寄存器的值，

开辟了 0x34 的新空间，一共偏移量为 0x40，该位置存的值为输入字符串地址。

此时调用了 read_six_numbers 函数。

```
0804920b <read_six_numbers>:
 804920b:       83 ec 2c                sub    $0x2c,%esp
 804920e:       8b 44 24 34             mov    0x34(%esp),%eax
 8049212:       8d 50 14                lea    0x14(%eax),%edx
 8049215:       89 54 24 1c             mov    %edx,0x1c(%esp)
 8049219:       8d 50 10                lea    0x10(%eax),%edx
 804921c:       89 54 24 18             mov    %edx,0x18(%esp)
 8049220:       8d 50 0c                lea    0xc(%eax),%edx
 8049223:       89 54 24 14             mov    %edx,0x14(%esp)
 8049227:       8d 50 08                lea    0x8(%eax),%edx
 804922a:       89 54 24 10             mov    %edx,0x10(%esp)
 804922e:       8d 50 04                lea    0x4(%eax),%edx
 8049231:       89 54 24 0c             mov    %edx,0xc(%esp)
 8049235:       89 44 24 08             mov    %eax,0x8(%esp)
 8049239:       c7 44 24 04 63 a3 04    movl   $0x804a363,0x4(%esp)
 8049240:       08
 8049241:       8b 44 24 30             mov    0x30(%esp),%eax
 8049245:       89 04 24                mov    %eax,(%esp)
 8049248:       e8 23 f6 ff ff          call   8048870 <__isoc99_sscanf@plt>
 804924d:       83 f8 05                cmp    $0x5,%eax
 8049250:       7f 05                   jg     8049257 <read_six_numbers+0x4c>
 8049252:       e8 7f fe ff ff          call   80490d6 <explode_bomb>
 8049257:       83 c4 2c                add    $0x2c,%esp
 804925a:       c3                      ret
```

```
guo@ubuntu:~/bomb18_202208040204$ gdb bomb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/guo/bomb18_202208040204/bomb...done.
(gdb) b main
Breakpoint 1 at 0x80489e4: file bomb.c, line 37.
(gdb) x/s 0x804a363
0x804a363:       "%d %d %d %d %d %d"
```

发现 12 行有一个地址 0x804a363 出现，被存储在当前%esp+0x4 的地方，我们可以用 gdb 看一下这个地址的内容，是 6 个整数。

```
 8048b88:       e8 7e 06 00 00          call   804920b <read_six_numbers>
 8048b8d:       83 7c 24 18 01          cmpl   $0x1,0x18(%esp)
 8048b92:       74 05                   je     8048b99 <phase_2+0x25>
 8048b94:       e8 3d 05 00 00          call   80490d6 <explode_bomb>
 8048b99:       8d 5c 24 1c             lea    0x1c(%esp),%ebx
 8048b9d:       8d 74 24 30             lea    0x30(%esp),%esi
 8048ba1:       8b 43 fc                mov    -0x4(%ebx),%eax
 8048ba4:       01 c0                   add    %eax,%eax
 8048ba6:       39 03                   cmp    %eax,(%ebx)
 8048ba8:       74 05                   je     8048baf <phase_2+0x3b>
 8048baa:       e8 27 05 00 00          call   80490d6 <explode_bomb>
 8048baf:       83 c3 04                add    $0x4,%ebx
 8048bb2:       39 f3                   cmp    %esi,%ebx
 8048bb4:       75 eb                   jne    8048ba1 <phase_2+0x2d>
 8048bb6:       83 c4 34                add    $0x34,%esp
 8048bb9:       5b                      pop    %ebx
 8048bba:       5e                      pop    %esi
 8048bbb:       c3                      ret
```

返回后比较%esp+0x18 是否为 1，如果为 1，跳过 bomb，接着把%esp+0x1c 的值传给%ebx，把%esp+0x30 的值传给%esi（均为地址）。

接下来是循环部分，将%esp+0x1c-4 也就是%esp+0x18 的值传给%eax，加倍，比较

与%esp+0x1c 地址中的值是否相同，如果相同跳过爆炸，对%ebx 中地址+4，比较与%esi 中值是否相同，如果不同，跳转至循环开始处，再次进行循环。

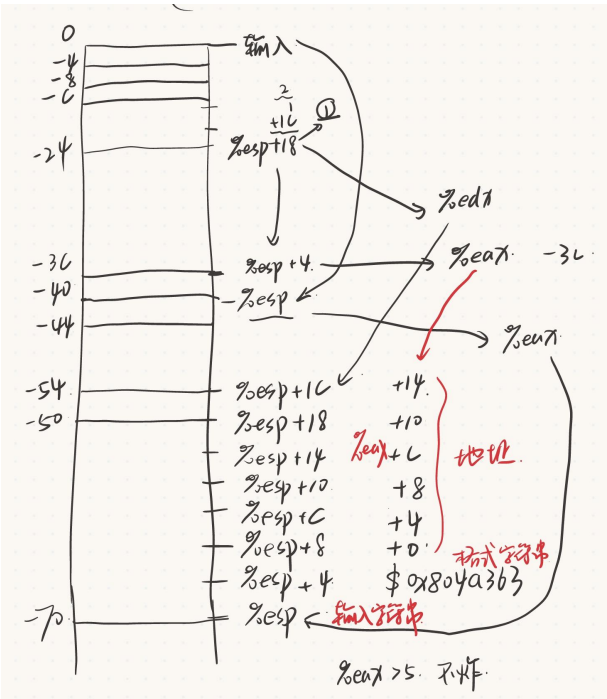
我们可以再看一下第二次循环，%ebx 的值为%esp+0x20，将%esp+0x20-4 也就是%esp+0x1c 的值传给%eax，加倍，比较与%esp+0x20 地址中的值是否相同，如果相同跳过爆炸，对%ebx 中地址+4，比较与%esi 中值是否相同，跳转，再次进入循环。

%esp+0x1c 与%esp+0x30 相差 0x14，即跳转 5 次，总共了执行 6 次该循环。由上述过程我们可以知道，如果%esp+0x18 到%esp+0x2c 存的值分别为 1,2,4,8,16,32 即为正确答案。

```
guo@ubuntu:~/bomb18_202208040204$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

测试通过。

但我还是很好奇为什么值会被存在%esp+0x18 到%esp+0x2c 中，我觉得问题应该是出现在函数 sscanf 上，这个函数并未给出汇编代码，上网查询资料可以得知：sscanf 会尝试根据格式字符串从输入字符串中读取六个整数，并将它们存储到之前计算出的地址中，返回值存放在%eax 中，表示的是成功解析和匹配的字符个数。问题解决，叙述与我之前画的简易栈内容完全匹配。格式字符串即为 0x804a363 我们所看到的“%d %d %d %d %d %d”，输入字符串是栈顶存储的地址，计算地址分别分布在更高地址处，简易栈如下：



2.3 phase_3 跳转表

每个函数开始前都将输入字符串放置在栈顶，以后将不再赘述。

```
08048bbc <phase_3>:
 8048bbc:       83 ec 2c                sub    $0x2c,%esp
 8048bbf:       8d 44 24 1c             lea    0x1c(%esp),%eax
 8048bc3:       89 44 24 0c             mov    %eax,0xc(%esp)
 8048bc7:       8d 44 24 18             lea    0x18(%esp),%eax
 8048bcb:       89 44 24 08             mov    %eax,0x8(%esp)
 8048bcf:       c7 44 24 04 6f a3 04    movl   $0x804a36f,0x4(%esp)
 8048bd6:       08
 8048bd7:       8b 44 24 30             mov    0x30(%esp),%eax
 8048bdb:       89 04 24                mov    %eax,(%esp)
 8048bde:       e8 8d fc ff ff          call   8048870 <__isoc99_sscanf@plt>
```

以上内容均与上一题 sscanf 部分基本相同，不再赘述。

```
guo@ubuntu:~/bomb18_202208040204$ gdb bomb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/guo/bomb18_202208040204/bomb...done.
(gdb) b main
Breakpoint 1 at 0x80489e4: file bomb.c, line 37.
(gdb) x/s 0x804a36f
0x804a36f:      "%d %d"
```

可以看一下格式字符串，对象为两个整数。

```
 8048be3:       83 f8 01                cmp    $0x1,%eax
 8048be6:       7f 05                   jg     8048bed <phase_3+0x31>
 8048be8:       e8 e9 04 00 00          call   80490d6 <explode_bomb>
 8048bed:       83 7c 24 18 07          cmpl   $0x7,0x18(%esp)
 8048bf2:       77 3c                   ja     8048c30 <phase_3+0x74>
 8048bf4:       8b 44 24 18             mov    0x18(%esp),%eax
 8048bf8:       ff 24 85 dc a1 04 08    jmp    *0x804a1dc(,%eax,4)
 8048bff:       b8 59 02 00 00          mov    $0x259,%eax
 8048c04:       eb 3b                   jmp    8048c41 <phase_3+0x85>
 8048c06:       b8 63 00 00 00          mov    $0x63,%eax
 8048c0b:       eb 34                   jmp    8048c41 <phase_3+0x85>
 8048c0d:       b8 97 00 00 00          mov    $0x97,%eax
 8048c12:       eb 2d                   jmp    8048c41 <phase_3+0x85>
 8048c14:       b8 b3 01 00 00          mov    $0x1b3,%eax
 8048c19:       eb 26                   jmp    8048c41 <phase_3+0x85>
 8048c1b:       b8 79 03 00 00          mov    $0x379,%eax
 8048c20:       eb 1f                   jmp    8048c41 <phase_3+0x85>
 8048c22:       b8 f1 01 00 00          mov    $0x1f1,%eax
 8048c27:       eb 18                   jmp    8048c41 <phase_3+0x85>
 8048c29:       b8 7a 00 00 00          mov    $0x7a,%eax
 8048c2e:       eb 11                   jmp    8048c41 <phase_3+0x85>
 8048c30:       e8 a1 04 00 00          call   80490d6 <explode_bomb>
 8048c35:       b8 00 00 00 00          mov    $0x0,%eax
 8048c3a:       eb 05                   jmp    8048c41 <phase_3+0x85>
 8048c3c:       b8 7b 03 00 00          mov    $0x37b,%eax
 8048c41:       3b 44 24 1c             cmp    0x1c(%esp),%eax
 8048c45:       74 05                   je     8048c4c <phase_3+0x90>
 8048c47:       e8 8a 04 00 00          call   80490d6 <explode_bomb>
 8048c4c:       83 c4 2c                add    $0x2c,%esp
 8048c4f:       c3                      ret
```

首先比较%eax 的返回值，如果大于 1，跳过 bomb，然后比较我们输入值与 7 的大小，如果小于等于 7 则不跳转 bomb。

接下来是一个立即跳转，我们可以看到下面的分支，所有的分支都汇聚到了 0x8048c41 这一步，它对%esp+0x1c 与%eax 比较，如果相同则通过。反向追溯，可以发现将不同的立即数放入%eax，也就是说输入的值可以不同，很容易想到跳转表，且第 19 行也是跳转表的表示方式，此题应有多解。跳转表首地址为 0x804a1dc，我们可以根据 gdb 查看跳转表中的地址，比如：

```
(gdb) x/4bx 0x804a1dc
0x804a1dc:      0xff    0x8b    0x04    0x08
```

当 x=0 时，即输入数字为 0 时，跳转地址为 0x08048bff，即上图第 18 行，将 0x259（转化为 10 进制为 601）立即数传送至%eax，后跳转至 0x8048c41 进行比较和判断，结束本题。

```
guo@ubuntu:~/bomb18_202208040204$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I can see Russia from my house!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 601
Halfway there!
```

测试通过。

```
That's number 2.  Keep going!
1 891
Halfway there!
```

```
That's number 2.  Keep going!
2 99
Halfway there!
```

```
That's number 2.  Keep going!
3 151
Halfway there!
```

```
That's number 2.  Keep going!
4 435
Halfway there!
```

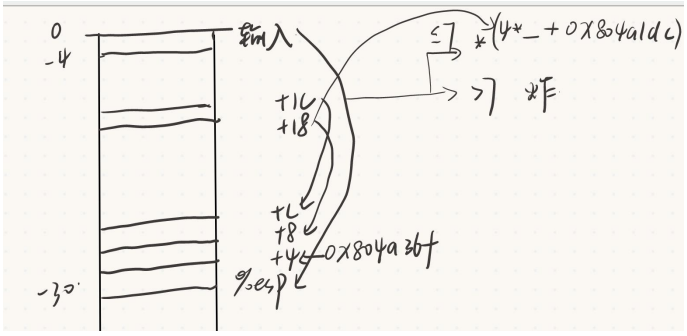
```
That's number 2.  Keep going!
5 889
Halfway there!
```

```
That's number 2.  Keep going!
6 497
Halfway there!
```

```
That's number 2.  Keep going!
7 122
Halfway there!
```

其余几个答案分别为：1 0x37b，2 0x63，3 0x97，4 0x1b3，5 0x379，6 0x1f1，7 0x7a。

转换为十进制为：1 891，2 99，3 151，4 435，5 889，6 497，7 122。



简易栈如图所示。

2.4 phase_4 递归函数

```
08048cad <phase_4>:
 8048cad:       83 ec 2c                sub    $0x2c,%esp
 8048cb0:       8d 44 24 18             lea    0x18(%esp),%eax
 8048cb4:       89 44 24 0c             mov    %eax,0xc(%esp)
 8048cb8:       8d 44 24 1c             lea    0x1c(%esp),%eax
 8048cbc:       89 44 24 08             mov    %eax,0x8(%esp)
 8048cc0:       c7 44 24 04 6f a3 04    movl   $0x804a36f,0x4(%esp)
 8048cc7:       08
 8048cc8:       8b 44 24 30             mov    0x30(%esp),%eax
 8048ccc:       89 04 24                mov    %eax,(%esp)
 8048ccf:       e8 9c fb ff ff          call   8048870 <__isoc99_sscanf@plt>
 8048cd4:       83 f8 02                cmp    $0x2,%eax
 8048cd7:       75 0e                   jne    8048ce7 <phase_4+0x3a>
```

以上部分与 phase_2 部分 sscanf 部分基本相同，不再赘述。但要注意的是，传参数的时候两个数的地址对调了，将%esp+0x18 放在了%esp+0xc 的位置，将%esp+0x1c 放在了%esp+0x8 的位置，与我们习惯的输入顺序相反。返回值（输入个数）若不为 2 则 bomb。

```
(gdb) x/s 0x804a36f
0x804a36f:      "%d %d"
```

查看栈中格式字符串部分，对象为两个整数。

```
 8048cd9:       8b 44 24 18             mov    0x18(%esp),%eax
 8048cdd:       83 f8 01                cmp    $0x1,%eax
 8048ce0:       7e 05                   jle    8048ce7 <phase_4+0x3a>
 8048ce2:       83 f8 04                cmp    $0x4,%eax
 8048ce5:       7e 05                   jle    8048cec <phase_4+0x3f>
 8048ce7:       e8 ea 03 00 00          call   80490d6 <explode_bomb>
 8048cec:       8b 44 24 18             mov    0x18(%esp),%eax
 8048cf0:       89 44 24 04             mov    %eax,0x4(%esp)
 8048cf4:       c7 04 24 09 00 00 00    movl   $0x9,(%esp)
 8048cfb:       e8 50 ff ff ff          call   8048c50 <func4>
 8048d00:       3b 44 24 1c             cmp    0x1c(%esp),%eax
 8048d04:       74 05                   je     8048d0b <phase_4+0x5e>
 8048d06:       e8 cb 03 00 00          call   80490d6 <explode_bomb>
 8048d0b:       83 c4 2c                add    $0x2c,%esp
 8048d0e:       c3                      ret
```

查看剩余部分，首先对输入的第二个数进行比较，如果小于等于 1 或大于 4 则 bomb，输入为 2,3,4 通过，然后将该数字放在%esp+4 的位置，将立即数 9 放在%esp 处，调用 func4 函数，在函数返回后，将输入的第一个数与返回值%eax 进行比较，如果相同则通过测试。接下来我们看一下 func4 的内容。

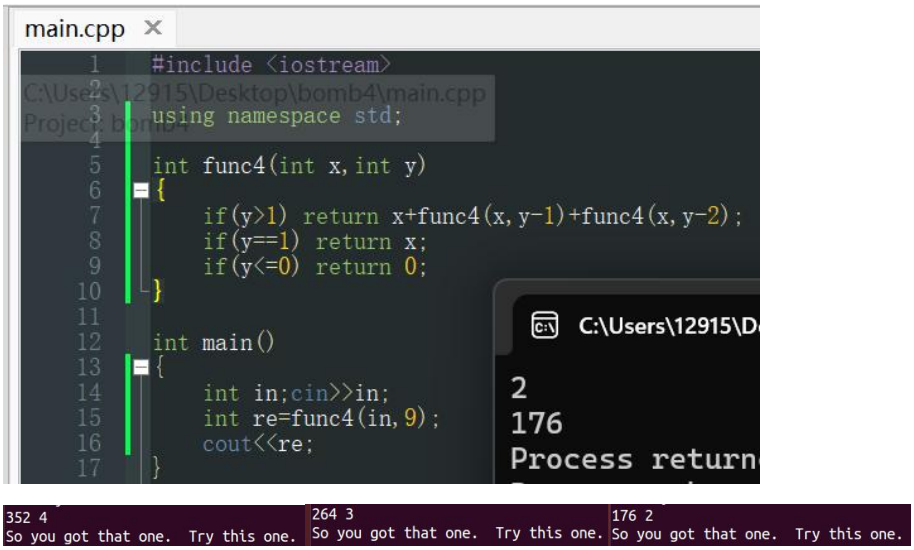
```
08048c50 <func4>:
 8048c50:       83 ec 1c                sub    $0x1c,%esp
 8048c53:       89 5c 24 10             mov    %ebx,0x10(%esp)
 8048c57:       89 74 24 14             mov    %esi,0x14(%esp)
 8048c5b:       89 7c 24 18             mov    %edi,0x18(%esp)
 8048c5f:       8b 74 24 20             mov    0x20(%esp),%esi
 8048c63:       8b 5c 24 24             mov    0x24(%esp),%ebx
```

首先做堆栈的准备，开辟一个新的 0x1c 的空间，将%ebx,%esi,%edi 寄存器保存，将立即数 9 放在%esi，将我们输入的数放在%ebx。

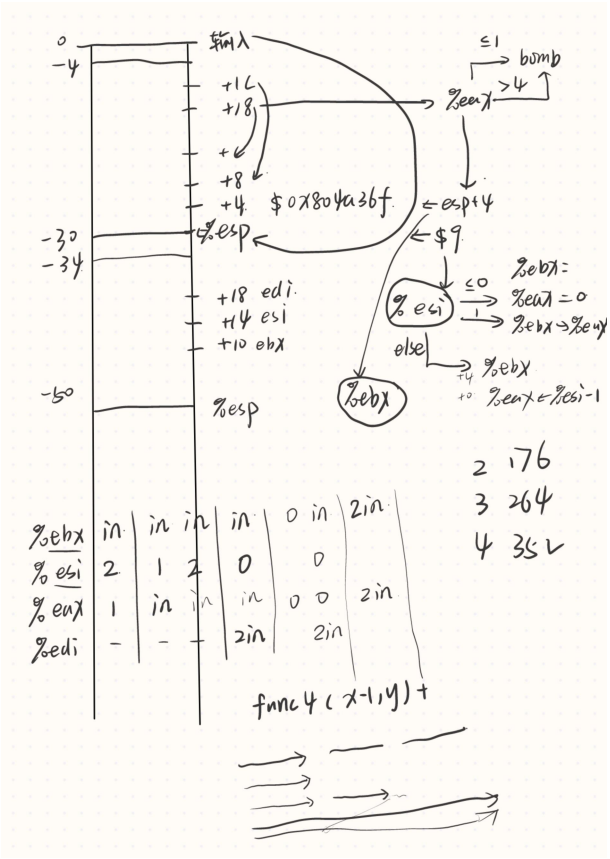
```
4  8048c67:    85 f6                test   %esi,%esi
3  8048c69:    7e 2b                jle    8048c96 <func4+0x46>
2  8048c6b:    83 fe 01             cmp    $0x1,%esi
1  8048c6e:    74 2b                je     8048c9b <func4+0x4b>
0  8048c70:    89 5c 24 04          mov    %ebx,0x4(%esp)
1  8048c74:    8d 46 ff             lea    -0x1(%esi),%eax
2  8048c77:    89 04 24             mov    %eax,(%esp)
3  8048c7a:    e8 d1 ff ff ff       call   8048c50 <func4>
4  8048c7f:    8d 3c 18             lea    (%eax,%ebx,1),%edi
5  8048c82:    89 5c 24 04          mov    %ebx,0x4(%esp)
6  8048c86:    83 ee 02             sub    $0x2,%esi
7  8048c89:    89 34 24             mov    %esi,(%esp)
8  8048c8c:    e8 bf ff ff ff       call   8048c50 <func4>
9  8048c91:    8d 1c 07             lea    (%edi,%eax,1),%ebx
0  8048c94:    eb 05                jmp    8048c9b <func4+0x4b>
1  8048c96:    bb 00 00 00 00       mov    $0x0,%ebx
2  8048c9b:    89 d8                mov    %ebx,%eax
3  8048c9d:    8b 5c 24 10          mov    0x10(%esp),%ebx
4  8048ca1:    8b 74 24 14          mov    0x14(%esp),%esi
5  8048ca5:    8b 7c 24 18          mov    0x18(%esp),%edi
6  8048ca9:    83 c4 1c             add    $0x1c,%esp
7  8048cac:    c3                   ret
```

然后是特殊值处理，如果%esi 的值小于等于 0，跳转到 8048c96，将 0 传给%eax 返回；如果%esi 的值为 1，跳转到 8048c9b，将%ebx 传给%eax 返回，即将我们输入的数字返回。

进入递归部分，将%ebx 和%esi-1 作为新的参数传给下一层递归函数，返回值与输入数字相加存储在%edi 中；再将%ebx 和%esi-2 作为新的参数传给下一层递归函数，返回值与%edi 相加存储在%ebx 中。将%ebx 传给%eax 返回。如下是递归代码的 c++版本以及执行结果。



验证并通过。



简易栈如图所示。

以及一个惨痛的教训就是太复杂的递归不要想手搓栈和寄存器，可能是书上习题的递归比较简单，都能凭脑子画出来，这个递归太多步骤了，我在这里卡了好久，上网搜了别人的才知道可以用程序模拟，真的是先入为主了。

2.5 phase_5 指针

```
08048d0f <phase_5>:
 8048d0f:       53                      push   %ebx
 8048d10:       83 ec 28                sub    $0x28,%esp
 8048d13:       8b 5c 24 30             mov    0x30(%esp),%ebx
 8048d17:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8048d1d:       89 44 24 1c             mov    %eax,0x1c(%esp)
 8048d21:       31 c0                   xor    %eax,%eax
 8048d23:       89 1c 24                mov    %ebx,(%esp)
 8048d26:       e8 80 02 00 00          call   8048fab <string_length>
 8048d2b:       83 f8 06                cmp    $0x6,%eax
 8048d2e:       74 05                   je     8048d35 <phase_5+0x26>
 8048d30:       e8 a1 03 00 00          call   80490d6 <explode_bomb>
 8048d35:       b8 00 00 00 00          mov    $0x0,%eax
```

首先将旧%ebx 入栈，然后开辟新的空间，将输入地址传送到%ebx 中。接下来从%gs:0x14 处取内容放置在%esp+0x1c 处，后续我们可以知道这是为了做一个栈保护（哨兵变量）。而后将输入地址放在栈顶，作为地址传送给函数 string_length，接下来将返回内容即字符串长

度与 6 比较，如果不是 6 就 bomb，也就是说，我们的输入为字符串且长度为 6。

```
494  8048d35:    b8 00 00 00 00          mov    $0x0,%eax
495  8048d3a:    0f be 14 03             movsbl (%ebx,%eax,1),%edx
496  8048d3e:    83 e2 0f                and    $0xf,%edx
497  8048d41:    0f b6 92 fc a1 04 08    movzbl 0x804a1fc(%edx),%edx
498  8048d48:    88 54 04 15             mov    %dl,0x15(%esp,%eax,1)
499  8048d4c:    83 c0 01                add    $0x1,%eax
500  8048d4f:    83 f8 06                cmp    $0x6,%eax
501  8048d52:    75 e6                   jne    8048d3a <phase_5+0x2b>
```

接下来是一个循环，首次执行时将%eax 置为 0，而后将%ebx 的值（输入地址）加%eax 并传给%edx，取其后四位，并加上一个立即数 0x804a1fc，再次取低八位，保存在栈内%esp+0x15+%eax 地址处，随后进行%eax 自加和与 6 的比较，如果不是 6，跳转到循环开头处。很容易看出，%eax 中的内容由 0 循环至 5，在为 5 时加一并退出循环，在此过程内，我们将【（输入地址+%eax 内数字的偏移量）后四位+地址 0x804a1fc】的内容传递到了栈上%esp+0x15 值%esp+0x1a 处。

```
502  8048d54:    c6 44 24 1b 00          movb   $0x0,0x1b(%esp)
503  8048d59:    c7 44 24 04 d2 a1 04    movl   $0x804a1d2,0x4(%esp)
504  8048d60:    08
505  8048d61:    8d 44 24 15             lea    0x15(%esp),%eax
506  8048d65:    89 04 24                mov    %eax,(%esp)
507  8048d68:    e8 57 02 00 00          call   8048fc4 <strings_not_equal>
508  8048d6d:    85 c0                   test   %eax,%eax
509  8048d6f:    74 05                   je     8048d76 <phase_5+0x67>
510  8048d71:    e8 60 03 00 00          call   80490d6 <explode_bomb>
511  8048d76:    8b 44 24 1c             mov    0x1c(%esp),%eax
512  8048d7a:    65 33 05 14 00 00 00    xor    %gs:0x14,%eax
513  8048d81:    74 05                   je     8048d88 <phase_5+0x79>
514  8048d83:    e8 48 fa ff ff          call   80487d0 <__stack_chk_fail@plt>
515  8048d88:    83 c4 28                add    $0x28,%esp
516  8048d8b:    5b                      pop    %ebx
517  8048d8c:    c3                      ret
```

在结束循环后，在栈中%esp+0x1b 处放 0 以指示字符串结束，将地址 0x804a1d2 放到%esp+4 处，将%esp+0x15（即刚才传递的字符串首地址）放在%esp 处，调用字符串比较函数进行比较，如果相同，则跳过 bomb，本题基本结束，进行栈保护检验工作（金丝雀检测）和空间释放。

让我们分别看一下地址 0x804a1d2 和地址 0x804a1fc 的内容。

```
(gdb) x/s 0x804a1d2
0x804a1d2:      "bruins"
```

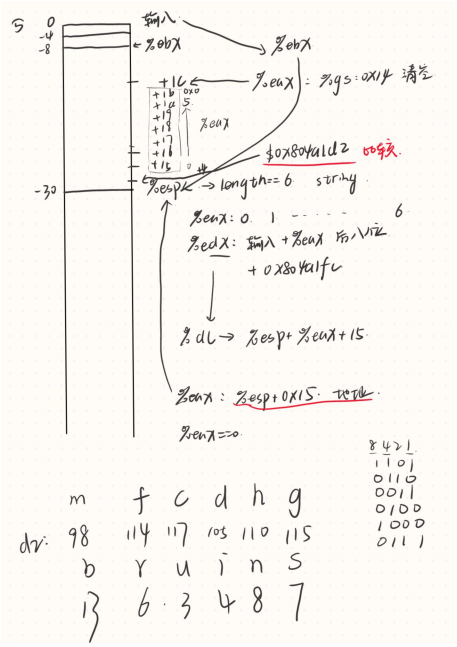
```
(gdb) x/s 0x804a1fc
0x804a1fc <array.2954>:  "maduiersnfotvbylSo you think you can stop the bomb with
ctrl-c, do you?"
```

经过以上分析，我们可以知道这个【（输入地址+%eax 内数字的偏移量）后四位+地址 0x804a1fc】的内容代表的含义：取输入字符串中相对%eax 偏移的那个字符，取 ASCII 码后四位，加上地址 0x804a1fc，并取对应的字符。而在上图中我们可以看到，0x804a1d2 的内容是一个单词，0x804a1fc 中是一段需要我们自行计算偏移量的字符串，我们找到相对应的

偏移并根据 ASCII 码表找到对应字符即可找到结果。当然，结果并不唯一。

```
mfcdhg
Good work!  On to the next...
```

验证并通过，这是其中一个答案。

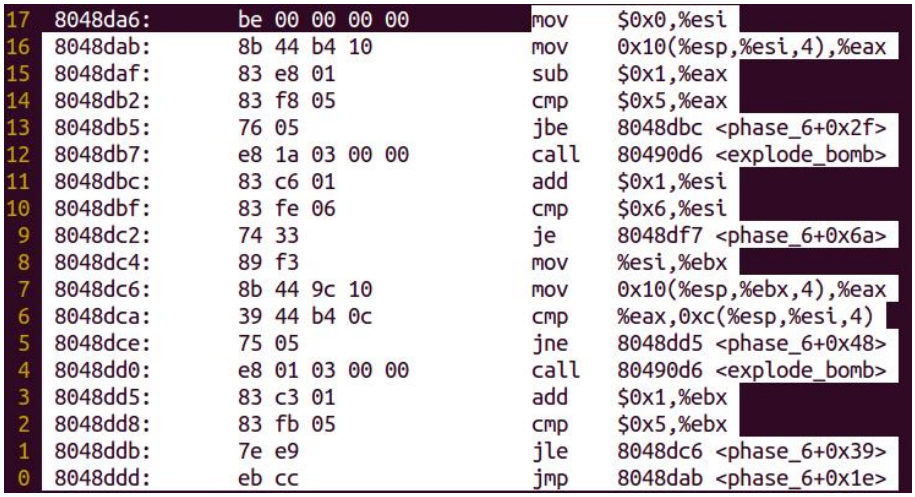


简易栈如图所示。

2.6 phase_6 链表

```
08048d8d <phase_6>:
 8048d8d:       56                      push   %esi
 8048d8e:       53                      push   %ebx
 8048d8f:       83 ec 44                sub    $0x44,%esp
 8048d92:       8d 44 24 10             lea    0x10(%esp),%eax
 8048d96:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d9a:       8b 44 24 50             mov    0x50(%esp),%eax
 8048d9e:       89 04 24                mov    %eax,(%esp)
 8048da1:       e8 65 04 00 00          call   804920b <read_six_numbers>
 8048da6:       be 00 00 00 00          mov    $0x0,%esi
```

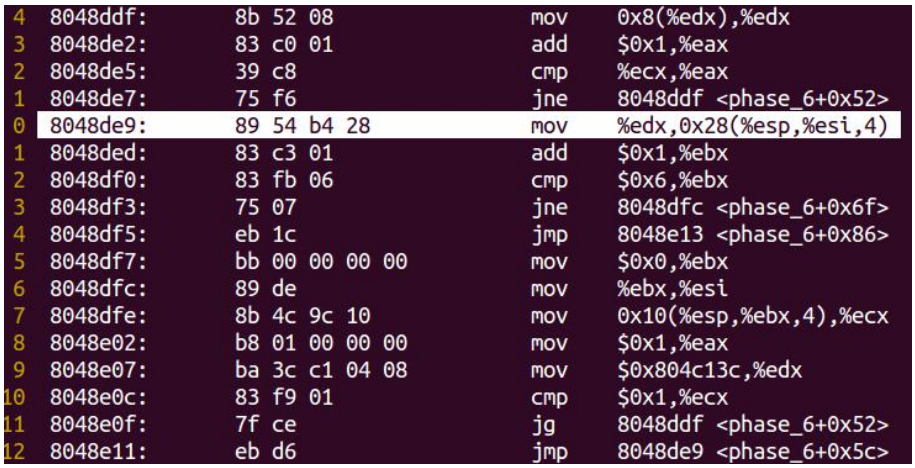
首先还是栈的准备，将%esi 入栈，开辟一个 0x44 的新空间，将%esp+0x10 位置的**地址**放在%esp+4 处，将输入地址放在%esp 处，准备 read_six_numbers 的调用，这段代码与第二题的相同，不另做分析，最终结果为将输入的六个整数分别放置在了%esp+0x10 至%esp+0x24 处。我们接下来一部分代码的循环将围绕着这个首地址%esp+0x10。



接下来的这段代码是一个循环，由两个跳转拼接而成。首先对%esi 置初值 0，将%esp+4%esi+0x10 位置的值传给%eax，这里%esi 的作用看起来很像数组的下标。然后将%eax 减一并与 5 相比较，如果小于等于就跳过 bomb，可以看出输入的数字一定小于等于 6。然后对下标%esi 的值自加，与 6 相比，如果不相等则不跳出目前的部分。

提前将%ebx 与%esi 的值置为相等，接下来是两个重要的地址，%esp+4%ebx+0x10 与%esp+4%esi+0xc 作比较，也就是将当前指向的位置与前一位置作比较。如果比较结果不相等，跳过 bomb，对%ebx 自加并与 5 作比较，小于等于则跳转到两个重要地址比较部分（不断将当前指向位置后移），否则将跳转到函数的起始部分（将被比较的位置后移）。

这里的逻辑其实相当于一个双重 for 循环，当所有数都互不相等且所有数都小于 6 时满足条件，不 bomb。



接下来仍然为循环。需要注意的是，上一部分我们跳出循环的目的地址为 0x8048df7，所以我们初始从这里开始，首先将%ebx 置为 0，%esi 置为%ebx，然后%edx 存放地址 0x804c13c，【将%eax 置为 1，将%esp+4%ebx+0x10 放在%ecx 内并与 1 相比较，如果大于

1 跳转到这段最开始部分，%edx 存放与当前地址偏移量为 8 的地址的内容，若%eax 与%ecx 不相等，重复上一步骤直至相等；否则不进行%edx 的偏移（输入为 1，偏移量为 0）】

接下来将%edx 的内容存放在栈内%esp+4%esi+0x28 的位置（输入的 6 个数的更高位），将%ebx 自加并与 6 比较，如果相等跳出循环，否则跳过%ebx 置零的部分，并循环进行如上所有操作。

操作为：从地址 0x804c13c 开始，根据我们输入的数字，查找存储单元内的数据（是地址）存到栈中。

```
8048e13:    8b 5c 24 28           mov    0x28(%esp),%ebx
8048e17:    8b 44 24 2c           mov    0x2c(%esp),%eax
8048e1b:    89 43 08              mov    %eax,0x8(%ebx)
8048e1e:    8b 54 24 30           mov    0x30(%esp),%edx
8048e22:    89 50 08              mov    %edx,0x8(%eax)
8048e25:    8b 44 24 34           mov    0x34(%esp),%eax
8048e29:    89 42 08              mov    %eax,0x8(%edx)
8048e2c:    8b 54 24 38           mov    0x38(%esp),%edx
8048e30:    89 50 08              mov    %edx,0x8(%eax)
8048e33:    8b 44 24 3c           mov    0x3c(%esp),%eax
8048e37:    89 42 08              mov    %eax,0x8(%edx)
8048e3a:    c7 40 08 00 00 00 00  movl   $0x0,0x8(%eax)
```

从栈中向外取地址，首个地址放到%ebx 中，第二个放在首地址+8 的位置，第三个放在第二个+8 的位置，以此类推，根据我们上部分内容取出的不同地址覆盖初始值，重新串联各部分。

```
8048e41:    be 05 00 00 00        mov    $0x5,%esi
8048e46:    8b 43 08              mov    0x8(%ebx),%eax
8048e49:    8b 10                 mov    (%eax),%edx
8048e4b:    39 13                 cmp    %edx,(%ebx)
8048e4d:    7e 05                 jle    8048e54 <phase_6+0xc7>
8048e4f:    e8 82 02 00 00        call   80490d6 <explode_bomb>
8048e54:    8b 5b 08              mov    0x8(%ebx),%ebx
8048e57:    83 ee 01              sub    $0x1,%esi
8048e5a:    75 ea                 jne    8048e46 <phase_6+0xb9>
8048e5c:    83 c4 44              add    $0x44,%esp
8048e5f:    5b                    pop    %ebx
8048e60:    5e                    pop    %esi
8048e61:    c3                    ret
```

这段代码为比较。在此前的过程中，%ebx 中存的内容为首个地址。每次取前一个节点的值【（%ebx）】和后一个进行比较【%edx/（%eax）】，前一个节点的值必须小于等于后一个节点，否则 bomb。至此本题目结束，收回栈空间。

如上内容可知，该问题的主要内容是一个链表，每一个节点的 0-3 为内容，8-11 为下一个节点的地址，我们需要通过输入的数字将不同的下一节点地址放在栈中，然后根据顺序重新存到节点中，使得前一节点的值一定小于等于后一节点。

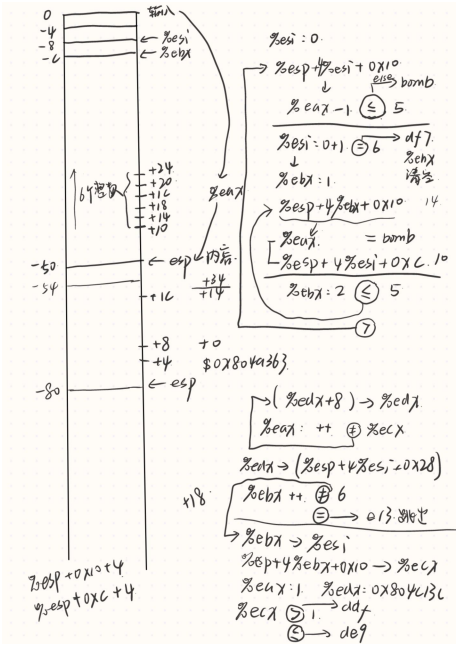
我们可以用 gdb 看一下内存中存储这个链表区域的内容。

```
(gdb) x/20xw 0x804c13c
0x804c13c <node1>:      0x00000382      0x00000001      0x0804c148      0x000003c7
0x804c14c <node2+4>:    0x00000002      0x0804c154      0x000003d3      0x00000003
0x804c15c <node3+8>:    0x0804c160      0x00000306      0x00000004      0x0804c16c
0x804c16c <node5>:      0x00000307      0x00000005      0x0804c178      0x000002e8
0x804c17c <node6+4>:    0x00000006      0x00000000      0x00000012      0x00000000
```

可以看到每三个块为一个节点，前六个的节点内容分别为0x382,0x3c7,0x3d3,0x306,0x307,0x2e8，所以我们输入的数字应该为 6 4 5 1 2 3。

```
6 4 5 1 2 3
Congratulations! You've defused the bomb!
```

测试通过。



简易栈如图所示。

2.7 hidden question

```
17      /* Hmm...  Six phases must be more secure than one phase! */
18      input = read_line();             /* Get input                  */
19      phase_1(input);                  /* Run the phase              */
20      phase_defused();                 /* Drat!  They figured it out!
21                                        * Let me know how they did it. */
22      printf("Phase 1 defused. How about the next one?\n");
23
24      /* The second phase is harder.  No one will ever figure out
25       * how to defuse this... */
26      input = read_line();
27      phase_2(input);
28      phase_defused();
29      printf("That's number 2.  Keep going!\n");
```

让我们再次观察主函数，发现每一个函数后面都有一个 phase_defused 未用到，且注释中给出了提示，应该就是隐藏关卡的进入点。让我们看一下反汇编的代码。

```
19 0804925b <phase_defused>:
18  804925b:       81 ec 8c 00 00 00       sub    $0x8c,%esp
17  8049261:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
16  8049267:       89 44 24 7c             mov    %eax,0x7c(%esp)
15  804926b:       31 c0                   xor    %eax,%eax
14  804926d:       83 3d cc c3 04 08 06    cmpl   $0x6,0x804c3cc
13  8049274:       75 72                   jne    80492e8 <phase_defused+0x8d>
```

这里首先开辟了一个栈空间，做了一个栈保护，然后将 0x804c3cc 的内容与 6 比较，猜测是我们通过的关卡数量，如果没通过六关就不会开启这个隐藏任务。

```
9  8049276:       8d 44 24 2c             lea    0x2c(%esp),%eax
8  804927a:       89 44 24 10             mov    %eax,0x10(%esp)
7  804927e:       8d 44 24 28             lea    0x28(%esp),%eax
6  8049282:       89 44 24 0c             mov    %eax,0xc(%esp)
5  8049286:       8d 44 24 24             lea    0x24(%esp),%eax
4  804928a:       89 44 24 08             mov    %eax,0x8(%esp)
3  804928e:       c7 44 24 04 75 a3 04    movl   $0x804a375,0x4(%esp)
2  8049295:       08
1  8049296:       c7 04 24 d0 c4 04 08    movl   $0x804c4d0,(%esp)
0  804929d:       e8 ce f5 ff ff          call   8048870 <__isoc99_sscanf@plt>
1  80492a2:       83 f8 03                cmp    $0x3,%eax
2  80492a5:       75 35                   jne    80492dc <phase_defused+0x81>
```

这里仍然使用了 sscanf 函数，将 0x804c4d0 中的内容放到栈中，这是它的输入地址，格式为 0x804a375 中存储的格式，我们可以看一下它的内容。

```
(gdb) x/s 0x804a375
0x804a375:      "%d %d %s"
```

是两个整数和一个字符串。

```
(gdb) n
Halfway there!
94          input = read_line();
(gdb) 176 2
Undefined command: "176".  Try "help".
(gdb) n
176 2
95          phase_4(input);
(gdb) info reg
eax            0x804c4d0        134530256
ecx            0x6      6
edx            0x4      4
ebx            0xbffff3e4       -1073744924
esp            0xbffff330       0xbffff330
ebp            0xbffff348       0xbffff348
```

可以看到执行到第四个函数的时候，地址相同，故得知该隐藏关卡是由第四关进入的。

```
10  80492a2:    83 f8 03              cmp    $0x3,%eax
 9  80492a5:    75 35                 jne    80492dc <phase_defused+0x81>
 8  80492a7:    c7 44 24 04 7e a3 04  movl   $0x804a37e,0x4(%esp)
 7  80492ae:    08
 6  80492af:    8d 44 24 2c           lea    0x2c(%esp),%eax
 5  80492b3:    89 04 24              mov    %eax,(%esp)
 4  80492b6:    e8 09 fd ff ff        call   8048fc4 <strings_not_equal>
 3  80492bb:    85 c0                 test   %eax,%eax
 2  80492bd:    75 1d                 jne    80492dc <phase_defused+0x81>
 1  80492bf:    c7 04 24 44 a2 04 08  movl   $0x804a244,(%esp)
 0  80492c6:    e8 35 f5 ff ff        call   8048800 <puts@plt>
 1  80492cb:    c7 04 24 6c a2 04 08  movl   $0x804a26c,(%esp)
 2  80492d2:    e8 29 f5 ff ff        call   8048800 <puts@plt>
 3  80492d7:    e8 d7 fb ff ff        call   8048eb3 <secret_phase>
 4  80492dc:    c7 04 24 a4 a2 04 08  movl   $0x804a2a4,(%esp)
 5  80492e3:    e8 18 f5 ff ff        call   8048800 <puts@plt>
 6  80492e8:    8b 44 24 7c           mov    0x7c(%esp),%eax
 7  80492ec:    65 33 05 14 00 00 00  xor    %gs:0x14,%eax
 8  80492f3:    74 05                 je     80492fa <phase_defused+0x9f>
 9  80492f5:    e8 d6 f4 ff ff        call   80487d0 <__stack_chk_fail@plt>
10  80492fa:    81 c4 8c 00 00 00     add    $0x8c,%esp
11  8049300:    c3                    ret
```

接着观察该函数，将 0x804a37e 地址的内容与输入内容的第三个内容相比较，相同时才能触发关卡，查看该地址内容。

```
(gdb) x/s 0x804a37e
0x804a37e:      "DrEvil"
```

在第四关答案后输入 DrEvil 即可开启隐藏关卡。接着我们观察隐藏关卡的汇编代码部分。

```
 2 08048eb3 <secret_phase>:
 3  8048eb3:    53                    push   %ebx
 4  8048eb4:    83 ec 18              sub    $0x18,%esp
 5  8048eb7:    e8 41 02 00 00        call   80490fd <read_line>
 6  8048ebc:    c7 44 24 08 0a 00 00  movl   $0xa,0x8(%esp)
 7  8048ec3:    00
 8  8048ec4:    c7 44 24 04 00 00 00  movl   $0x0,0x4(%esp)
 9  8048ecb:    00
10  8048ecc:    89 04 24              mov    %eax,(%esp)
11  8048ecf:    e8 0c fa ff ff        call   80488e0 <strtol@plt>
12  8048ed4:    89 c3                 mov    %eax,%ebx
13  8048ed6:    8d 40 ff              lea    -0x1(%eax),%eax
14  8048ed9:    3d e8 03 00 00        cmp    $0x3e8,%eax
15  8048ede:    76 05                 jbe    8048ee5 <secret_phase+0x32>
16  8048ee0:    e8 f1 01 00 00        call   80490d6 <explode_bomb>
17  8048ee5:    89 5c 24 04           mov    %ebx,0x4(%esp)
18  8048ee9:    c7 04 24 88 c0 04 08  movl   $0x804c088,(%esp)
19  8048ef0:    e8 6d ff ff ff        call   8048e62 <fun7>
20  8048ef5:    83 f8 06              cmp    $0x6,%eax
21  8048ef8:    74 05                 je     8048eff <secret_phase+0x4c>
22  8048efa:    e8 d7 01 00 00        call   80490d6 <explode_bomb>
23  8048eff:    c7 04 24 ac a1 04 08  movl   $0x804a1ac,(%esp)
24  8048f06:    e8 f5 f8 ff ff        call   8048800 <puts@plt>
25  8048f0b:    e8 4b 03 00 00        call   804925b <phase_defused>
26  8048f10:    83 c4 18              add    $0x18,%esp
27  8048f13:    5b                    pop    %ebx
28  8048f14:    c3                    ret
```

首先进行一些堆栈准备，然后将 0xa 压入%esp+8，0x0 压入%esp+4，%eax（猜测为输入）压入%esp，调用一个 strtol 函数，查询得知该函数是一个 c 语言库函数，strtol(char *ch1,char *ch2,int base)，作用是将字符串*ch1 当作一个 base 进制的整数并返回，这里的 base=0xa，为十进制，调用 strtol，说明把输入的字符串当做一个 10 进制的整数来使用。

调用该函数结束后，将返回值-1，与 0x3e8 作比较，如果小于等于跳过 bomb 函数，准备两个参数并调用 fun7。返回值为 6 的时候跳过 bomb，结束该函数。接下来我们进入 fun7 函数观察。

```
08048e62 <fun7>:
 8048e62:       53                      push   %ebx
 8048e63:       83 ec 18                sub    $0x18,%esp
 8048e66:       8b 54 24 20             mov    0x20(%esp),%edx
 8048e6a:       8b 4c 24 24             mov    0x24(%esp),%ecx
 8048e6e:       85 d2                   test   %edx,%edx
 8048e70:       74 37                   je     8048ea9 <fun7+0x47>
 8048e72:       8b 1a                   mov    (%edx),%ebx
 8048e74:       39 cb                   cmp    %ecx,%ebx
 8048e76:       7e 13                   jle    8048e8b <fun7+0x29>
 8048e78:       89 4c 24 04             mov    %ecx,0x4(%esp)
 8048e7c:       8b 42 04                mov    0x4(%edx),%eax
 8048e7f:       89 04 24                mov    %eax,(%esp)
 8048e82:       e8 db ff ff ff          call   8048e62 <fun7>
 8048e87:       01 c0                   add    %eax,%eax
 8048e89:       eb 23                   jmp    8048eae <fun7+0x4c>
 8048e8b:       b8 00 00 00 00          mov    $0x0,%eax
 8048e90:       39 cb                   cmp    %ecx,%ebx
 8048e92:       74 1a                   je     8048eae <fun7+0x4c>
 8048e94:       89 4c 24 04             mov    %ecx,0x4(%esp)
 8048e98:       8b 42 08                mov    0x8(%edx),%eax
 8048e9b:       89 04 24                mov    %eax,(%esp)
 8048e9e:       e8 bf ff ff ff          call   8048e62 <fun7>
 8048ea3:       8d 44 00 01             lea    0x1(%eax,%eax,1),%eax
 8048ea7:       eb 05                   jmp    8048eae <fun7+0x4c>
 8048ea9:       b8 ff ff ff ff          mov    $0xffffffff,%eax
 8048eae:       83 c4 18                add    $0x18,%esp
 8048eb1:       5b                      pop    %ebx
 8048eb2:       c3                      ret
```

这是一个递归函数，通过画出寄存器和过程可以知道，当我们输入的值和传入地址中存储的值相同时返回 0；当输入的值大时，递归调用%ecx（不变）和(%edx)+4 为参的 fun7，并在调用函数结束后返回 2%eax；当输入的值小时，递归调用%ecx（不变）和(%edx)+8 为参的 fun7，并在调用函数结束后返回 2%eax+1。

我们已经知道我们的返回值为 9，也就是说，进入函数之后第一层为 2***3**=6，调用进入第二层为 2***1**+1，调用进入第三层为 2***0**+1，第四层为 **0**。

```
(gdb) x/40x 0x804c088
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
```

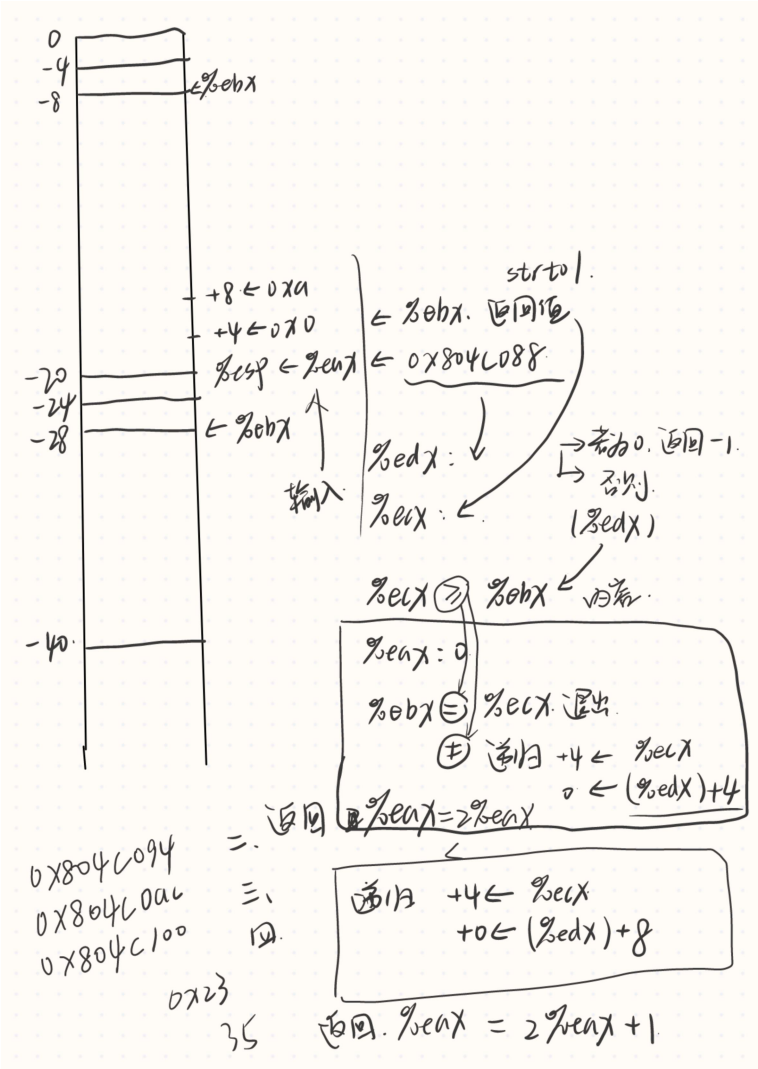
让我们观察一下内存中的值。

第二层的地址是+4 的位置，0x804c094，跳转到该地址处，第三层的位置是当下+8,0x804c0ac，跳转，第四层的位置为 0x804c100，因为相等，返回值为 0，所以这个地址

处的值就是我们输入的值，为 0x23，十进制为 35。验证一下我们的思路。

```
But finding it and solving it are quite different...
35
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

这个隐藏函数的作用类似一个二叉搜索树，%ebx 里存的是当下节点的值，用于和%ecx 即输入的值（要查找的值）比较，当小于的时候查当下地址+4，大的时候查当下地址+8，始终使用指针进行地址间的跳转和数据访问。



简易栈如图所示。

3 总结

3.1 实验中出现的问题

有一些程序读起来非常痛苦，感觉脑子都缩小了，比如那个很多层的递归，参考了别人的思路才知道可以写对应的 C 代码让程序给你算，可以说是我先入为主了。

有一些函数有隐含的一些操作，比如使用了很多次的 sscanf 函数，把输入读取到栈中，它能提供给我们很多信息。

与我而言比较有效的方法是一步一步地画出栈和寄存器的值，根据他的过程来推测他的操作，不过也会容易被程序绕晕，写完这个 bomb 实验我读汇编的能力比之前好很多，尤其是经过递归的洗礼。

3.2 心得体会

做得最艰难和痛苦的一次实验，主要是做了很久，每一道题都研究很久，感觉总也做不完，而且这一周还有 OS 的验收、期中考、组会等等很多事情，压力非常大。不过终于是做完了，感觉有很大的收获，对汇编和栈的理解更加深刻了，也窥视到了用汇编写程序的程序员们日常的一角，非常佩服。