# 拆炸弹实验报告


【实验目的】

理解汇编语言，学会使用调试器。

【实验原理】

二进制炸弹是作为一个目标代码文件提供给学生们的程序，运行时，它提示用户输入6个不同的字符串。如果其中任何一个不正确，炸弹就会“爆炸”：打印出一条错误信息。学生通过反汇编和逆向工程来确定是哪六个字符串，从而解除他们各自炸弹的雷管。

【实验过程】

一、使用putty登录、修改密码

1、打开putty，输入用户名和密码（csapp），使用命令passwd username修改密码。

2、使用命令ls后看到有一个文件bomb49.tar，输入 tar xvf bomb49.tar 解压后，再用命令ls看到目录下新出现了bomb、bomb.c和README三个文件。一开始在看到.c文件后很开心，以为这样就可以看到c语言代码了，那拆炸弹的任务就变得简单多了。输入cat bomb.c后，把c代码看了一遍，发现原来这里只有主函数，每一关的具体代码都没有。看来只有从汇编代码入手了。

3、输入反汇编命令objdump -d bomb后，出现了大量汇编代码，在putty小窗口里看起来很麻烦。

二、用linux终端作准备

1、Ubuntu装好了之后，在终端输入ssh username@10.92.13.8连接到服务器，并开始新一轮的尝试。

2、输入objdump -d bomb > 1.txt将汇编代码输出到服务器上一个自动生成的叫1.txt的文件中。

3、中断连接，退回自己的系统桌面，使用命令scp username@10.92.8:1.txt 1.txt 将在桌面复制生成一个也叫1.txt的文件。这时候就可以很方便的查看汇编代码了。

三、开始拆炸弹

1、首先是找到main函数，发现它调用了从phase1到phase6这六个函数。这应该就是每一关需要看懂的函数了。

2、于是找到phase1,代码如下：

```
08048dd9 <phase_1>:
 8048dd9:    55                      push   %ebp
 8048dda:    89 e5                   mov    %esp,%ebp
 8048ddc:    83 ec 08                sub    $0x8,%esp
 8048ddf:    c7 44 24 04 e4 98 04    movl   $0x80498e4,0x4(%esp) ②
```

```
8048de6:    08
8048de7:    8b 45 08                mov    0x8(%ebp),%eax
8048dea:    89 04 24                mov    %eax,(%esp)
8048ded:    e8 89 01 00 00          call   8048f7b <strings_not_equal>
8048df2:    85 c0                   test   %eax,%eax ①
8048df4:    74 05                   je     8048dfb <phase_1+0x22>
8048df6:    e8 c0 04 00 00          call   80492bb <explode_bomb>
8048dfb:    c9                      leave
8048dfc:    8d 74 26 00             lea    0x0(%esi),%esi
8048e00:    c3                      ret
```

可以看到在%eax!=0的时候就会调用<explode_bomb>，所以在调用<strings_not_equal> 函数之后的返回值%eax必须为0。继续往前，发现代码movl $0x80498e4,0x4(%esp)有立即数，是将此处地址的值拿来用，输入gdb bomb进入调试状态，用x/s 0x80498e4查看内容，终端显示出字符串"Why make trillions when we could make... billions?"。

下面一步 mov 0x8(%ebp),%eax就是把我们输入的参数放进%eax中，然后放进(%esp) ，再调用函数<strings_not_equal>。很容易猜测我要输入的就是0x80498e4中的字符。

于是开始第一关的尝试。非常重要的一步是在<explode_bomb>之前设置断点，找到该函数的入口地址是0x80492bb,即:break *0x80492bb。然后输入命令run，进入程序，在输入提示的下一行输入"Why make trillions when we could make... billions?"，终端显示“Phase 1 defused. How about the next one?”也即第一关顺利通过。

3、第二关

汇编代码略。

首先要注意到的是<read_six_numbers>这个函数，根据名字的提示可以猜测这一关要我输入六个数字。紧接着这个函数三条指令的是cmpl $0x1,-0x20(%ebp)，je 8048f42 <phase_2+0x49>和call 80492bb <explode_bomb>，也就是如果输入的第一个数不等于1则炸弹爆炸。因此第一个数为1。接下来是一个循环，两条重要指令是imul -0x4(%esi,%edx,4),%eax 和cmp %eax,(%esi,%edx,4)，其中%eax 和%edx在每次循环中加1，因此第二到第六个数分别为1*2=2, 2*3=6, 4*4=24, 24*5=120, 120*6=720。经检验，结果正确！

4、第三关

汇编代码略。

注意到movl $0x8049abb,0x4(%esp)，输入指令x/s 0x8049abb，得到0x8049abb: "%d %d"，显示出应该输入两个数字。而cmp $0x1,%eax表明输入参数必须多于1个。再往下到达cmpl $0x7,-0x4(%ebp)，即输入的第一个参数值必须小于等于7。然后看到jmp *0x8049920(,%eax,4)，这是典型的switch跳转语句，即跳转到以地址*0x8049920为基址的跳转表中。输入p/x *0x8049920，得到地址0x8048ea2，在代码中找到该处指令，得到第一个输入为0时对应的第二个输入为0x211,转换成十进制为529。经调试后结果正确。当然此题不止一个答案，当输入为1时，通过p/x *0x8049924,得到地址0x8048e97，找到第二个输入为447。其他答案不一一写出。

5、第四关

phase4汇编代码略。

同样的由movl $0x8049abe,0x4(%esp)我们知道这一关是要输入一个数字。由cmpl $0x0,-0x4(%ebp) 知道输入的参数必须大于0。注意到这里调用了函数<func4>，找到代码如下：

```
08048bd0 <func4>:
 8048bd0:    55                      push   %ebp
 8048bd1:    89 e5                   mov    %esp,%ebp
 8048bd3:    83 ec 18                sub    $0x18,%esp
 8048bd6:    89 5d f8                mov    %ebx,-0x8(%ebp)
 8048bd9:    89 75 fc                mov    %esi,-0x4(%ebp)
 8048bdc:    8b 75 08                mov    0x8(%ebp),%esi
 8048bdf:    b8 01 00 00 00          mov    $0x1,%eax
 8048be4:    83 fe 01                cmp    $0x1,%esi
 8048be7:    7e 1a                   jle    8048c03 <func4+0x33>
 8048be9:    8d 46 ff                lea    -0x1(%esi),%eax
 8048bec:    89 04 24                mov    %eax,(%esp)
 8048bef:    e8 dc ff ff ff          call   8048bd0 <func4>
 8048bf4:    89 c3                   mov    %eax,%ebx
 8048bf6:    8d 46 fe                lea    -0x2(%esi),%eax
 8048bf9:    89 04 24                mov    %eax,(%esp)
 8048bfc:    e8 cf ff ff ff          call   8048bd0 <func4>
 8048c01:    01 d8                   add    %ebx,%eax
 8048c03:    8b 5d f8                mov    -0x8(%ebp),%ebx
 8048c06:    8b 75 fc                mov    -0x4(%ebp),%esi
 8048c09:    89 ec                   mov    %ebp,%esp
 8048c0b:    5d                      pop    %ebp
 8048c0c:    c3                      ret
```

由cmp $0x1,%esi 及下面的代码知如果所传递的参数小于等于1则结束<func4> ，跳转到主结构，否则减1并继续调用<func4> 并在调用的<func4> 又一次调用其本身，也就是进入双层递归模式。在满足跳出条件后，将每一次的值加到%eax ，且有如下关系：

func4(0)=1;func4(1)=1;func4(2)=func4(0)+func4(1)...

这其实就是斐波那契数列。然后回到<phase_4>，发现紧接着的代码是cmp $0xe9,%eax，说明<func4> 返回值应该是0xe9,转为十进制是233,也就是斐波那契数列的第12个。使用gdb调试结果正确。

6、第五关

汇编代码略。

由以下两句代码call 8048f60 <string_length>和cmp $0x6,%eax可知我们需要输入一个长度为6的字符串。看到mov $0x8049940,%ecx，在终端输入x/s 0x8049940，得到“isrveawhobpnutfgERROR: dup(0) error”。往下看到movsbl (%esi,%edx,1),%eax 和and $0xf,%eax 就是把输入的字符串中的每个字符依次拿出来，保留低四位，高四位任意。

接下来的两句movzbl (%ecx,%eax,1),%eax 和mov %al,(%ebx,%edx,1)将%ecx中根据%eax的值位移后的字符的低四位放到目的地。接下来是循环体中%edx的值再加1，直到全部6个字符串比较完毕。注意到movl $0x8049917,0x4(%esp)的立即数，使用x/s 0x8049917，得到字符串"giants"。之后将调用函数<strings_not_equal>。

到此思路已经基本清晰，也就是要求我们输入含有6个字符的字符串，这六个字符的ASCII码的低四位所指示的0x8049940中的字符串中的位置所对应的字符分别是"giants"六个字符。首先是第一个字符g，它在0x8049940中的位置是f，所以输入的第一个字符的ASCII码的低四位就是f。其余的5个字符也是根据这种关系得到它们ASCII码的低四位。最后得到一种答案为“opekma”。经检验，结果正确。

7、第六关

代码如下：

```
08048d20 <phase_6>:
 8048d20:   55                      push   %ebp
 8048d21:   89 e5                   mov    %esp,%ebp
 8048d23:   53                      push   %ebx
 8048d24:   83 ec 14                sub    $0x14,%esp
 8048d27:   c7 44 24 08 0a 00 00    movl   $0xa,0x8(%esp)
 8048d2e:   00
 8048d2f:   c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
 8048d36:   00
 8048d37:   8b 45 08                mov    0x8(%ebp),%eax
 8048d3a:   89 04 24                mov    %eax,(%esp)
 8048d3d:   e8 0a fb ff ff          call   804884c <strtol@plt>
 8048d42:   bb 2c a6 04 08          mov    $0x804a62c,%ebx
 8048d47:   89 03                   mov    %eax,(%ebx)
 8048d49:   89 1c 24                mov    %ebx,(%esp)
 8048d4c:   e8 bc fe ff ff          call   8048c0d <fun6>
 8048d51:   8b 40 08                mov    0x8(%eax),%eax
 8048d54:   8b 40 08                mov    0x8(%eax),%eax
 8048d57:   8b 40 08                mov    0x8(%eax),%eax
 8048d5a:   8b 40 08                mov    0x8(%eax),%eax
 8048d5d:   8b 40 08                mov    0x8(%eax),%eax
 8048d60:   8b 00                   mov    (%eax),%eax
 8048d62:   3b 03                   cmp    (%ebx),%eax
 8048d64:   74 05                   je     8048d6b <phase_6+0x4b>
 8048d66:   e8 50 05 00 00          call   80492bb <explode_bomb>
 8048d6b:   83 c4 14                add    $0x14,%esp
 8048d6e:   5b                      pop    %ebx
 8048d6f:   5d                      pop    %ebp
 8048d70:   c3                      ret
```

还是先观察<explode_bomb>前面的跳转条件，这里是cmp (%ebx),%eax 。在调用完<fun6> 之后是一系列链表操作，而(%ebx)的值没有改变，还是我们所输入的参数。对于函数<strtol@plt>，作用是将我们输入的数字当作字符串处理，然后转为十进制。简单地说就是输入什么十进制数参数就是那个十进制数本身。

然后研究 <fun6> 的输入参数。送入参数是这句代码：mov %ebx,(%esp)，往前查看，有mov $0x804a62c,%ebx ，使用x/s 0x804a62c发现该地址里存的是空字符串，也就是<fun6> 没有输入参数！那么它的返回值就应该是固定的。

在je 8048d6b <phase_6+0x4b> 前设置一个断点，使用gdb调试，在进入第六关时随便输入一个数字（输入100）,程序执行到断点。用info reg观察此时寄存器内容。发现%eax=500（十进制）,而我的输入是放在 (%ebx)里面的，也就是输入应该就是500。

经验证，结论是正确的。

但是真的就是这么简单吗？答案是否定的。因为在调试secret_phase时，我再次查看了0x804a62c地址处的值，却发现该处放着数字“500”，也就是我在进入第六关时的输入。说明之前的空字符串只是在我没有输入的时候为空，也就说明<fun6>并不是没有参数进入的。<fun6>的参数正是我们的输入。那么，为什么之前可以歪打正着得到答案呢？我又试了两次，分别在第六关输入不同的答案（100和200），然后使用info reg查看寄存器状态，却发现%eax的值仍然为500。这是否意味着我们的输入并不影响<fun6>的返回值？

为了更清楚第六关的变化，认真研究了<fun6>，经过反复推算、设置断点查看地址值，得到许多段由三个连续单元组成的序列。三个连续单元依次是一个十六进制数、一个索引值、一个地址。由此推测这是一段列表，且此列表是往低地址延续的。使用x/50w 0x804a5bc，得到如下代码：

```
0x804a5bc： 0x00000000 0x000003b5 0x00000009 0x00000000
0x804a5cc： 0x000001b5 0x00000008 0x0804a5c0 0x00000377
0x804a5dc： 0x00000007 0x0804a5cc 0x0000032e 0x00000006
0x804a5ec： 0x0804a5d8 0x0000015f 0x00000005 0x0804a5e4
0x804a5fc： 0x000001f4 0x00000004 0x0804a5f0 0x000001b0
0x804a60c: 0x00000003 0x0804a5fc 0x00000306 0x00000002
0x804a61c: 0x0804a608 0x00000280 0x00000001 0x0804a614
0x804a62c: 0x00000000 0x00000000 0x0804a620 0x000003e9
```

（注：删除了<node>等符号。）

由此我们可以得到链表顺序如下（地址只写出最后三位）：

62c->620->614->608->5fc->5f0->5e4->5d8->5cc->5c0->end

对应的值如下（这里使用十六进制，省略了前面的0x）：

0->280->306->1b0->1f4->15f->32e->377->1b5->3b5

其中第一个值0用我们的输入代替即可。

然后在<fun6>后面的0x8048d51设置一个断点，看看运行后链表发生了什么变化。在进入第六关后输入100，进入断点调试。使用x/50w 0x804a5fc，得到如下代码：

```
0x804a5bc： 0x00000000 0x000003b5 0x00000009 0x0804a5d8
0x804a5cc： 0x000001b5 0x00000008 0x0804a608 0x00000377
0x804a5dc： 0x00000007 0x0804a5e4 0x0000032e 0x00000006
0x804a5ec： 0x0804a614 0x0000015f 0x00000005 0x0804a62c
0x804a5fc： 0x000001f4 0x00000004 0x0804a5cc 0x000001b0
0x804a60c： 0x00000003 0x0804a5f0 0x00000306 0x00000002
0x804a61c： 0x0804a620 0x00000280 0x00000001 0x0804a5fc
0x804a62c： 0x00000064 0x00000000 0x00000000 0x000003e9
```

发现链表顺序改变如下：

5c0->5d8->5e4->614->620->5fc->5cc->608->5f0->62c->end

对应的值如下：

3b5->377->32e->306->280->1f4->1b5->1b0->15f->64

很容易发现经过<fun6>后列表进行了从大到小的排序。又在单步调试后发现返回值为0x804a5c0，也就是最大值的地址。而后经过六次mov 0x8(%eax),%eax的传递，得到第六大的数，与输入进行比较。

到此为止，<fun6>的全部疑惑已经解决。之所以前面输入0、100和200都会有答案500，是因为这三个数并没有影响500在链表中的正向排序。也就是说，第六关的答案应该是500到640（就是第五大的数）中的任意一个，这样就可以替代500成为第六大的数，并返回与它自己比较。重新运行，输入501、600和640，同样可以通过第六关。因此结论正确！

8、寻找秘密关卡

之前就听有的同学说其实不只有六关，网上查找到的资料也有提示由秘密关卡的。

仔细看了与唯一提到进入秘密关卡的<phase_defused>函数，从上往下看首先发现cmpl $0x6,0x804a86c，估计是看你是否已经通过了前面6关，没有的话就直接跳到结束部分。若已经通过6关，应该会往下执行。又有一句出现了立即数的movl $0x8049a7c,0x4(%esp) ，用x/s 0x8049a7c，得到 "%d %s"。下面是movl $0x804a970,(%esp)，同样地，得到该地址处为数字12。然后调用函数<sscanf@plt>并在这次要求返回值为2。猜测sscanf可能是C语言的内部函数，查到其定义为：int sscanf(const char *str, const char *format,...)，一个使用实例为：sscanf(“s 1”, “%s %d”, str, &a)，函数返回2（因为接收了2个参数。因此这里要求0x804a970处存放的是一个数字和一个字符串。

在cmp $0x2,%eax处设置断点，运行后发现%eax=1。而地址 0x804a970仍然为数字12,这个数字和第四关的输入相同，而且第四关也调用了<sscanf@plt>这个函数。

回到第四关的分析，我写了这么一句话“由movl $0x8049abe,0x4(%esp) 我们知道这一关是要输入一个数字”，而调用这个函数的时候要求返回值为1,所以当时就觉得这句话有些多余，因为只有一个输入的话返回值必然只能等于1，即使输入多个数字因为它只能接受一个参数返回值仍然会等于1。

为了验证猜想，再次使用gdb调试，在输入第四关答案时故意在12后面输入了字符串“hello”，结果第四关仍能通过！说明猜想正确！然后我继续调试，在输入第六关答案之后程序再次遇到断点，这时我查看地址0x804a970，发现除了数字12,还有字符串“hello”！说明秘密关卡和我的第四关是相联系的！这就解决了cmp $0x2,%eax 的问题。继续往下走是：movl $0x8049a82,0x4(%esp)，该地址处存储着字符串 "austinpowers"。

接下来调用了函数<strings_not_equal>。函数的另一个参数正是我在第四关的“额外输入”。也就是说我在第四关输入的字符串就应该是"austinpowers"!如此则可满足调用该函数之后返回值为0的问题。

继续往下看，又有一句含有立即数的代码：movl $0x8049b6c,(%esp)，查看其内容是"Curses, you've found the secret phase!"也就是说此时我已经进入了秘密关卡！即进入秘密关卡的钥匙就藏在第四关，就是那个“额外输入”的字符串。

9、秘密关卡

代码如下：

```
08048cba <secret_phase>:
 8048cba:    55                      push   %ebp
 8048cbb:    89 e5                   mov    %esp,%ebp
 8048cbd:    53                      push   %ebx
 8048cbe:    83 ec 14                sub    $0x14,%esp
 8048cc1:    e8 0a 07 00 00          call   80493d0 <read_line>
```

```
 8048cc6:    c7 44 24 08 0a 00 00   movl   $0xa,0x8(%esp)
 8048ccd:    00
 8048cce:    c7 44 24 04 00 00 00   movl   $0x0,0x4(%esp)
 8048cd5:    00
 8048cd6:    89 04 24               mov    %eax,(%esp)
 8048cd9:    e8 6e fb ff ff         call   804884c <strtol@plt>
 8048cde:    89 c3                  mov    %eax,%ebx
 8048ce0:    8d 40 ff               lea    -0x1(%eax),%eax
 8048ce3:    3d e8 03 00 00         cmp    $0x3e8,%eax
 8048ce8:    76 05                  jbe    8048cef <secret_phase+0x35>
 8048cea:    e8 cc 05 00 00         call   80492bb <explode_bomb>
 8048cef:    89 5c 24 04            mov    %ebx,0x4(%esp)
 8048cf3:    c7 04 24 e0 a6 04 08   movl   $0x804a6e0,(%esp)
 8048cfa:    e8 6a ff ff ff         call   8048c69 <fun7>
 8048cff:    83 f8 07               cmp    $0x7,%eax
 8048d02:    74 05                  je     8048d09 <secret_phase+0x4f>
 8048d04:    e8 b2 05 00 00         call   80492bb <explode_bomb>
 8048d09:    c7 04 24 bc 98 04 08   movl   $0x80498bc,(%esp)
 8048d10:    e8 57 fc ff ff         call   804896c <puts@plt>
 8048d15:    e8 11 05 00 00         call   804922b <phase_defused>
 8048d1a:    83 c4 14               add    $0x14,%esp
 8048d1d:    5b                     pop    %ebx
 8048d1e:    5d                     pop    %ebp
 8048d1f:    c3                     ret
```

首先一句call 80493d0 <read_line>，表明程序先读入一行，随后返回值%eax作为函数<strtol@plt>的参数之一，另外两个参数分别是0xa和0x0,也就是我们需要输入一个十进制数。由lea -0x1(%eax),%eax 和cmp $0x3e8,%eax 这两句知输入的十进制数要小于等于1001。随后将所输入的数作为<fun7> 的参数之一。另外一个参数来自 0x804a6e0，查看为0x24。

下面是<fun7>函数的代码：

```
08048c69 <fun7>:
 8048c69:    55                     push   %ebp
 8048c6a:    89 e5                  mov    %esp,%ebp
 8048c6c:    53                     push   %ebx
 8048c6d:    83 ec 14               sub    $0x14,%esp
 8048c70:    8b 5d 08               mov    0x8(%ebp),%ebx//第一个参数A
 8048c73:    8b 4d 0c               mov    0xc(%ebp),%ecx//第二个参数B，也就是输入
 8048c76:    b8 ff ff ff ff         mov    $0xffffffff,%eax
 8048c7b:    85 db                  test   %ebx,%ebx//递归终止，返回%ebx =0
 8048c7d:    74 35                  je     8048cb4 <fun7+0x4b>
 8048c7f: 8b 13                     mov    (%ebx),%edx
 8048c81:    39 ca                  cmp    %ecx,%edx
 8048c83:    7e 13                  jle    8048c98 <fun7+0x2f>
```

```
  //若*A>B，将(A+4)作为地址进入递归
 8048c85:    89 4c 24 04          mov    %ecx,0x4(%esp)
 8048c89:    8b 43 04             mov    0x4(%ebx),%eax
 8048c8c:89 04 24                  mov    %eax,(%esp)
 8048c8f: e8 d5 ff ff ff           call   8048c69 <fun7>
 8048c94:    01 c0                add    %eax,%eax//在此处将递归返回值加倍
 8048c96:    eb 1c                jmp    8048cb4 <fun7+0x4b>
 8048c98:    b8 00 00 00 00       mov    $0x0,%eax
 8048c9d:    39 ca                cmp    %ecx,%edx
 8048c9f: 74 13                    je     8048cb4 <fun7+0x4b>
 8048ca1:    89 4c 24 04          mov    %ecx,0x4(%esp)
   //若*A<B将(A+8)作为地址进入递归
 8048ca5:    8b 43 08             mov    0x8(%ebx),%eax
 8048ca8:    89 04 24             mov    %eax,(%esp)
 8048cab:e8 b9 ff ff ff            call   8048c69 <fun7>
   //在此处将递归返回值加倍后再加1
 8048cb0:    8d 44 00 01           lea    0x1(%eax,%eax,1),%eax
 8048cb4:    83 c4 14             add    $0x14,%esp
 8048cb7:    5b                   pop    %ebx
 8048cb8:    5d                   pop    %ebp
 8048cb9:    c3                   ret
```

在调用完<fun7>之后，紧跟着cmp $0x7,%eax，也就是返回值必须为7。分析函数发现它也是一个递归函数，注释如上。递归最深处的返回值肯定是0,即最外层返回值为7，则可得出如下反递归过程：

A*2+1=7-->A=3　　即有*A<B

A*2+1=3-->A=1　　同样有*A<B

A*2+1=1-->A=0　　也是*A<B

也就是说在这三次递归中都是执行了“若*A<B将(A+8)作为地址进入递归”系列代码。则使用p/x *(0x804a6e0+8)得到了地址0x804a6c8，这里存储的数字是0x32。接下来是p/x *(0x804a6c8+8)得到地址0x804a698，这里存储的数字是0x6b。最后一次，p/x *(0x804a698+8)，得到地址0x804a638，这里存储的数字就是我们所要输入的0x3e9,也就是十进制的1001。这个数字刚好满足秘密关卡开始出的“进入条件”。

至此，七个关卡全部通过。

【实验结论】

这次实验的答案如下：（2、4、5不止一个答案）

```
Why make trillions when we could make... billions?
1 2 6 24 120 720
0 529
12 austinpowers
opekma
500
1001
```

通过这次实验，对于Linux系统的一些操作命令有了一些了解和掌握，学习了如何使用gdb这个强大的工具进行调试，以及加深了对于汇编语言的熟悉。