# Bomblab Report

bomb4

## Start

分析`bomb.c`可知，bomb有6个phase，每个phase可以被特定的input来defuse。因此我们的任务是分析汇编代码来找出这些input。

用`objdump -d bomb > bomb.s`把bomb反汇编到文件，便于查看汇编代码。可以看到函数`explode_bomb`，从名称推测应该是使炸弹爆炸。

`gdb bomb`加载可执行文件bomb到gdb。在gdb中设置断点`b explode_bomb`防止爆炸，然后`r`运行，`layout asm`打开窗口查看当前运行到的汇编代码。

## Techniques

- x86的寄存器名称比较难记，有时还直接用内存引用作为operand，因此我理解指令的主要方式是把它们的operands换成好记点的名称在纸上写一遍
- 指令中突兀的内存地址往往是突破点

## Phase 1: String

`string_not_equal`是比较rdi和rsi两个参数所存地址的字符串是否相等。rdi已经放了输入字符串地址，`mov $0x402730,%esi`这句显然就是目标字符串地址。因此在gdb中用`x/s 0x402730`查看目标字符串，发现是"Verbosity leads to unclear, inarticulate things."。

## Phase 2: Sequence

由于涉及到栈的使用，函数首尾分别插入`mov %fs:0x28,%rax`和`xor %fs:0x28,%rax`，使用金丝雀值进行栈破坏检测，防止缓冲区溢出攻击。

`read_six_numbers`读入六个数字，按地址从低到高依次放到栈上（该函数中调用sscanf前有一句`mov $0x402a61,%esi`，gdb中`x/s 0x402a61`查看发现是"%d %d %d %d %d %d"，确定函数作用）。

接下来两句`cmpl`检查头两个数字是0和1。后面是一个循环，遍历六个数，检查相邻两项之和是不是后一个项，因此这六个数是个斐波那契数列，答案是"0 1 1 2 3 5"。

## Phase 3: Jump

gdb中`x/s 0x40278e`查看scanf的pattern string是"%d %c %d"，说明要输入两个数字和一个字符。它们会分别被存入rsp+16，rsp+15，rsp+20。

本阶段关键在这句`jmpq *0x4027a0(,%rax,8)`，根据输入的数字来决定跳转位置（0x4027a0地址是一个跳转表，`*<addr>`表示跳往`<addr>`所存的地址）。gdb中`x/12x 0x4027a0`查看跳转表，发现第一个地址对应的就是当前指令，因此简单起见第一个输入设为0，相当于不进行跳转。之后有一句`mov $0x7a,%eax`，然后`cmpl $0x34,0x14(%rsp)`检查第二个输入数字是否为52。

跳转到函数末尾后，`cmp 0xf(%rsp), %al`将输入的字符与刚才存入rax的`0x7a`比较，因此输入的字符是'z'。"0 z 52"是一个可行的答案。

# Phase 4: Recur

gdb中`x/s 0x402a6d`查看scanf的pattern string是"%d %d"，说明要输入两个数字，记为a，b。它们会分别被存入rsp+4和rsp。

`sub` `cmp` `jbe`三条语句等价于需要$2 \leq b \leq 4$。然后会使用参数`(6, b)`调用`func4`，调用后比较a是否等于返回值。

`func4`写成C语言可以是这样：

```
int func4(int x, int y) {
  if (x == 0) return 0;
  if (x == 1) return y;
  return y + func4(x-1, y) + func4(x-2, y);
}
```

结果实际上就是$y * (fib(x+2) - 1)$，$fib$是首项为零的斐波那契数列。因此`func(6, b)`的结果是$20 * b$，"60 3"是一个可行的答案。

# Phase 5：Permutation

与phase 4相同，输入两个数字，存入rsp和rsp+4。

`and $0xf,%eax`和`cmp $0xf,%eax`提取eax的最低4位，并且不能是0xf（即15）。

`mov 0x4027e0(,%rax,4),%eax`看起来是int数组索引（取a[i]），前面15提示长度可能是16，gdb`x/16x 0x4027e0`查看发现存放的的确是一个数组，而且是一个0到15的permutation。

接下来的指令写成C语言可以是这样（其中i的初始值由第一个输入确定）：

```
// int a[], i;
int sum = 0, cnt = 0;
do {
  cnt++;
  i = a[i];
  sum += i;
} while (i != 15);
```

如果把每一项看成从i到a[i]的一条边，整个数组就确定了一个环：不断i := a[i]最后一定会回到起点。后续指令检查`cnt`为15且`sum`等于第二个输入，因此需要跳15次，由于a[15] = 5，从5开始跳恰好能满足。这样最后`sum`为115，因此答案是"5 115"。

# Phase 6: LinkedList

与phase 2相同，这个phase也是调用`read_six_numbers`获取输入。后面一段指令是检查输入的每个数都在1到6之间，然后检查它们两两不相等，因此是要输入一个1到6的permutation。随后会对每个数做x := 7-x。`mov $0x6042f0,%edx`出现了一个地址，gdb中`x/32x`查看发现数据有规律，继而发现每两个值是后一个的地址，而且前面有个`<node#>`label，因此应该是存放了如下的结构体数组：

```
struct node {
  int val;
  node* next;
};
```

也就是一个链表（长度为6）。后面的代码似乎是根据`val`域进行排序，尝试了几种，最后是按照`val`逆序的节点编号"3 2 6 1 5 4"每个用7减，变成"4 5 1 6 2 3"作为输入，即为答案。

# Secret Phase: BST

查看汇编代码可以发现有一个叫`<secret_phase>`的标签，在`phase_defused`中有调用。调用前有一个sscanf，格式为"%d %d %s"，source字符串是"0 0"，因此大概是要在有相同格式的phase输入后面加一个字符串来进入。后面有一个字符串比较，`x/s 0x402ac0`得到目标字符串是"DrEvil"，因此加在同样需要输入两个整数的phase4输入后面，果然进入了secret phase。

secret phase接受一个整数输入。查看secret phase中的地址`x/128x 0x604110`发现也是结构体数组，前面的标签是`<n##>`。每32个字节是一个节点，其中分别是value、地址1、地址2。继而发现是二叉树，而且还是一棵二叉搜索树，`<nij>`表示第i层的从左往右第j个节点。对应结构体如下：

```
struct n {
  int val;
  n* left, right;
};
```

secret phase中用根节点和输入值调用了`fun7`，是一个在树上搜索的递归函数。逻辑如下：

```
int fun7(n* node, int tar) {
  if (node == NULL) return -1;
  if (tar == node->val) return 0;
  if (tar < node->val) return fun7(node->left, tar)*2;
  if (tar > node->val) return fun7(node->right, tar)*2 + 1;
}
```

最后需要返回值为7，而之前对输入的搜索值`tar`的范围也作了限制（1到0x3e9即1001）。树中第三层最右边的数恰好为0x3e9，这条搜索路径上返回值会是0->1->3->7，正好满足。因此答案是"1001"。