计系实验——“BOMBLAB”Writeup

2025-04-13 15:55:10

~~Crypto手勇闯Reverse新手村现状~~

# 背景介绍

这不是CTF比赛，而是一次~~平平无奇~~的“拆弹”实验。

实验要求：  
给了一个可执行文件~~./bomb~~，要求解析其每一阶段逻辑，将答案存储在./bomb.txt中。

本质上是一道逆向分析题。  
本意这是想让人去用命令行的gdb等去磕汇编代码，但显然上千行的汇编还是稍微有些难顶，因而有必要让IDA这一逆向“外挂”助人一臂之力。  
当然IDA此处利用的主要功能还是反编译协助生成伪C代码进行分析，虽然可读性依然极低（毕竟作为编译期语言，变量名和很多结构体之类的内存特征都被抹去，加上各种~~乱七八糟的寄存器横跳~~导致令人眼花缭乱的类型转换，以及各种调用协定+系统底层函数串门袭扰——  
**但总归是好过汇编吧！！！**

# 程序流程梳理

娴熟地IDA启动，bomb拖入，一通OK，找main函数，f5的结果形如：  
mainfunc

可以大致看出程序的主要流程（初始化以及六个衔接的阶段）。

~~题目的.tar中明明提供了main函数啊喂~~

IDA进行分析的经验法则：敢于“不求甚解”，即对于一些语义不太明确的函数/变量区域等，若能通过一些其它手段（调试、搜索、基础知识和盲猜等）确定其功能/含义，则应相信自身对含义的解析正确并继续分析。~~有时候看不懂不是自身的锅~~

不难推断出read\_line()就是一个取一行字符串的函数。查看其反编译代码后，可发现其中还有一些过长字符串截断和已读字符串总数统计等小功能。  
略去无关紧要的部分，直接开始逐个phase点进去反编译分析：

由于每个人的题目数据不同，加之笔者并非专业Re手，因此题解和解析均仅供参考，欢迎读者交流、指正。

# 阶段一

最友好的一个阶段~~签到题.sage~~

## 反编译代码

\_\_int64 \_\_fastcall phase\_1(\_\_int64 a1)  
{  
 \_\_int64 result; // rax  
  
 result = strings\_not\_equal(a1, "I can see Russia from my house!");  
 if ( (\_DWORD)result )  
 explode\_bomb();  
 return result;  
}

## 解析

虽然逐行阅读后~~可以发现一种废话文学的美~~，但易看出只需要让a1字符串与后续字符串内容相同即可。

“废话文学”是因为由汇编重组成C时，只能还原程序的（大体）执行流程而不能完全还原源代码。源代码完全可以形如：

void phase\_1(const char\* x)  
{  
if(!strings\_not\_equal(x,"I can see Russia from my house!"))  
{  
explode\_bomb();  
}  
}

~~什么？你问我为什么返回值我置而不顾？我也不知道，但反正不重要，孩子对调用协定也没什么研究，不要强求一个crypto手太多~~

解为：I can see Russia from my house!

# 阶段二

反编译代码：

unsigned \_\_int64 \_\_fastcall phase\_2(\_\_int64 a1)  
{  
 int \*v1; // rbx  
 int v3[4]; // [rsp+0h] [rbp-38h] BYREF  
 char v4; // [rsp+10h] [rbp-28h] BYREF  
 unsigned \_\_int64 v5; // [rsp+18h] [rbp-20h]  
  
 v5 = \_\_readfsqword(0x28u);  
 read\_six\_numbers(a1, v3);  
 if ( v3[0] || v3[1] != 1 )  
 explode\_bomb();  
 v1 = v3;  
 do  
 {  
 if ( v1[2] != \*v1 + v1[1] )  
 explode\_bomb();  
 ++v1;  
 }  
 while ( v1 != (int \*)&v4 );  
 return v5 - \_\_readfsqword(0x28u);  
}

## 解析

略去各种奇怪的变量和系统函数调用，核心有几点：

* 调用read\_six\_numbers函数，这个函数会从a1中读取6个数字(int)并存储在v3开始的数组中，并且scanf的返回值不能小于5（否则GG）；
* 第一个if翻译一下，约束了v3[0] == 0 && v3[1] == 1；
* do\_while循环结合if应当翻译成这样： C for(int i=0;i<4;++i) { if(v3[i+2] != v3[i] + v3[i+1]) { explode\_bomb(); } }
* 上述“转译”过程对逆向者的C语言功底要求还是极高的，尤其是涉及内存布局和指针的部分；可以自行体会和复现一下这部分“转译”是怎么进行的。

转译完成后的逻辑就很简单了，即希望我们读取的6个数字能构成前2项为(0,1)的斐波那契数列。

解为：0 1 1 2 3 5

# 阶段三

## 反编译代码

unsigned \_\_int64 \_\_fastcall phase\_3(\_\_int64 a1)  
{  
 int v1; // eax  
 int v3; // [rsp+0h] [rbp-18h] BYREF  
 int v4; // [rsp+4h] [rbp-14h] BYREF  
 unsigned \_\_int64 v5; // [rsp+8h] [rbp-10h]  
  
 v5 = \_\_readfsqword(0x28u);  
 if ( (int)\_\_isoc99\_sscanf(a1, "%d %d", &v3, &v4) <= 1 )  
 explode\_bomb();  
 switch ( v3 )  
 {  
 case 0:  
 v1 = 104;  
 break;  
 case 1:  
 v1 = 735;  
 break;  
 case 2:  
 v1 = 815;  
 break;  
 case 3:  
 v1 = 943;  
 break;  
 case 4:  
 v1 = 924;  
 break;  
 case 5:  
 v1 = 93;  
 break;  
 case 6:  
 v1 = 757;  
 break;  
 case 7:  
 v1 = 208;  
 break;  
 default:  
 explode\_bomb();  
 }  
 if ( v4 != v1 )  
 explode\_bomb();  
 return v5 - \_\_readfsqword(0x28u);  
}

## 解析

~~难得的第三关比第二关简单的一天~~

没啥好说的，很容易就看出有七组可能解，随便选一组。

解为： 0 104（不唯一）

# 阶段四

原来实验中“选做题”的出现不是师出无名……

## 反编译代码

和本题有关的函数有两个，除了phase\_4外还有一个func4。

### phase\_4

unsigned \_\_int64 \_\_fastcall phase\_4(\_\_int64 a1)  
{  
 unsigned int v2; // [rsp+0h] [rbp-18h] BYREF  
 int v3; // [rsp+4h] [rbp-14h] BYREF  
 unsigned \_\_int64 v4; // [rsp+8h] [rbp-10h]  
  
 v4 = \_\_readfsqword(0x28u);  
 if ( (unsigned int)\_\_isoc99\_sscanf(a1, "%d %d", &v3, &v2) != 2 || v2 - 2 > 2 )  
 explode\_bomb();  
 if ( v3 != (unsigned int)func4(7LL, v2) )  
 explode\_bomb();  
 return v4 - \_\_readfsqword(0x28u);  
}

### func4

\_\_int64 \_\_fastcall func4(int a1, \_\_int64 a2)  
{  
 \_\_int64 result; // rax  
 int v3; // r12d  
  
 result = 0LL;  
 if ( a1 > 0 )  
 {  
 result = (unsigned int)a2;  
 if ( a1 != 1 )  
 {  
 v3 = func4((unsigned int)(a1 - 1), a2) + a2;  
 return v3 + (unsigned int)func4((unsigned int)(a1 - 2), (unsigned int)a2);  
 }  
 }  
 return result;  
}

## 解析

单论phase\_4的逻辑实际上还是比较简单的，约束整理：

* 读取两个整数~~并且还必须是两个整数~~，并且要求v2 - 2 <= 2；
* v3 == func4(7,v2)

v2 - 2 > 2中，由于v2的变量类型是unsigned int，因此最终的约束是 v2 <= 4 && v2 >= 2。

强制类型转换的处理应更为灵活：有些可以忽略，而有些则需认真考量之。  
主要原因是这些转换多数并不是源代码中显式出现，或与变量的隐式转换有关（如返回值类型int但调用函数使用了一个unsigned long long来接收），或与编译过程中寄存器类型横跳有关~~反正寄存器都在那，ax变rax/eax/sax等等就是编译器大手一挥的事(bushi)~~，具体对逆向逻辑是否有影响无法一概而论。

而func4不难看出是一个递归函数，其递归定义如下：对于f(i)(i >= 0，此处忽略v2的不变传递)，

* f(0) == 0；
* f(1) == v2；
* f(i) == f(i-1) + f(i-2) + v2 (i >= 2)。

数学上应该也碰到过不少这种递归定义，由于i == 7，因此手动计算即可满足要求。  
取合法的v2 = 2时：

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| f(i) | 0 | 2 | 4 | 8 | 14 | 24 | 40 | 66 |

解为：66 2（不唯一）

# 阶段五

Crypto入门之古典密码？

## 反编译代码

unsigned \_\_int64 \_\_fastcall phase\_5(\_\_int64 a1)  
{  
 \_\_int64 i; // rax  
 char v3[7]; // [rsp+1h] [rbp-17h] BYREF  
 unsigned \_\_int64 v4; // [rsp+8h] [rbp-10h]  
  
 v4 = \_\_readfsqword(0x28u);  
 if ( (unsigned int)string\_length() != 6 )  
 explode\_bomb();  
 for ( i = 0LL; i != 6; ++i )  
 v3[i] = array\_0[\*(\_BYTE \*)(a1 + i) & 0xF];  
 v3[6] = 0;  
 if ( (unsigned int)strings\_not\_equal(v3, "flames") )  
 explode\_bomb();  
 return v4 - \_\_readfsqword(0x28u);  
}

## 解析

反编译代码在某些情况下还是可能让人发懵的——比如函数调用的传参问题~~需要“自行体会”~~。

整理约束时起手就遇上了“意外”：出现了一个没有形参的string\_length()，但点进看string\_length()的反编译代码却是这样的：

\_\_int64 \_\_fastcall string\_length(\_BYTE \*a1)  
{  
 \_\_int64 result; // rax  
  
 if ( !\*a1 )  
 return 0LL;  
 LODWORD(result) = 0;  
 do  
 {  
 ++a1;  
 result = (unsigned int)(result + 1);  
 }  
 while ( \*a1 );  
 return result;  
}

该函数本身就是一个非常正常的strlen~~高仿~~实现，但反编译代码中调用方没有传递参数。  
如上情景和函数调用协定有关。本次程序中所有函数的调用协定都采用了\_\_fastcall，\_\_fastcall调用的特点：

* 前2个(x86)/前4个(x64)整型参数使用速度更快的寄存器\*CX和\*DX传递；x64的Linux更为复杂，最前2个整型参数通过RDI、RSI传递；
* 其余参数从右至左压栈；
* 被调用者(x86)/调用者(x64)负责清理栈。

回到main函数中，以phase\_5的调用前后为例，可以发现有如下汇编代码片段：~~关键时候还是得回去磕汇编~~

call phase\_defused  
lea rdi, aSoYouGotThatOn ; "So you got that one. Try this one."  
call \_puts  
call read\_line  
mov rdi, rax  
call phase\_5  
call phase\_defused  
lea rdi, aGoodWorkOnToTh ; "Good work! On to the next..."

read\_line会将返回值（一个const char\*）放在rax中，随后mov rdi,rax将rax的值移入rdi中，再调用phase\_5时rdi就是隐式的第一参数。

虽然看上去phase\_5“接受”了这个形参，但因为其并没有改变rdi的值，因此再去call string\_length时rdi还是第一参数，因而上述调用其实就是string\_length(a1)，亦即获取a1指向字符串的长度（a1的类型被标注为\_\_int64，本质上就是x64系统下指针的数据类型，参考C++的std::ptrdiff\_t）。

需要注意的是，阶段六中还会出现一次相同的情况（“消失的第一参数”）。

各位无需担心编程的问题——毕竟作为~~不那么高级的~~高级语言，编程时函数的调用方式一定为string\_length(str)（也就是只要~~不像可变参数越界那样~~乱搞，函数参数传递一定是正确的）。  
此现象是因为IDA反汇编时，由于其“观测”到phase\_5中没有对rdi进行操作，故会以为调用string\_length时没有传参——毕竟IDA的反汇编的目标不是对函数语义进行检查，亦不是去完全重构原函数，而是为了将汇编语言的逻辑以尽可能“眼熟”的方式展现出来。同时不自动填充参数的另一个原因是phase\_5自身并不知道谁在调用它，或给它传递的参数如何。

~~当然即使不知道汇编语言和调用协定之类的逻辑，盲猜也能知道这个函数获取的就是传入字符串的长度~~

解决这一小插曲后，对于一个目标字符串char v3[7]，其进行了一次单表映射加密，令v3[i] = array\_0[a1[i] & 0xF]（翻译后）；  
需满足的目标约束是令字符串v3为"flames"。

首先找到array\_0，双击array\_0可以在栈内存区中找到其定义：  
Stack

实际上这片内存被称为“字符串常量区”，C/C++程序开始运行时会在栈中开辟一段只读内存区，存放所有的字符串常量，即~~妇孺皆知~~的const char\*。  
因此其中也能发现一些有趣的内容，~~包括但不限于So you think you can stop the bomb with ctrl-c, do you?~~

直接逐个目标字符搜索其下标，即为令a1[i] & 0xF应达成的目标值；为令a1[i]~~在string.printable中~~可以打印，给目标下标加上16的整数倍后转化成字符即可，解同样不唯一。

如果当作CTF-Crypto题的话，应试着尽可能高效地写出Python解题代码：  
```python from string import ascii\_letters as lts

sec = "maduiersnfotvbyl"  
ls = [sec.index(i) for i in "flames"]  
res = ''  
  
for i in ls:  
 while not chr(i) in lts:  
 i += 0x10  
 res += chr(i)  
  
print(res)  
```

解为：IOAPEG（不唯一）

# 阶段六

## 反编译代码

unsigned \_\_int64 \_\_fastcall phase\_6(\_\_int64 a1)  
{  
 int \*v1; // r13  
 \_\_int64 i; // r14  
 \_\_int64 v3; // rbx  
 \_\_int64 j; // rsi  
 int v5; // ecx  
 int v6; // eax  
 \_QWORD \*v7; // rdx  
 \_\_int64 v8; // rbx  
 \_\_int64 v9; // rax  
 \_\_int64 v10; // rdx  
 \_\_int64 v11; // rax  
 \_\_int64 v12; // rdx  
 \_\_int64 v13; // rax  
 int v14; // ebp  
 int v16[8]; // [rsp+0h] [rbp-88h] BYREF  
 \_\_int64 v17; // [rsp+20h] [rbp-68h]  
 \_\_int64 v18; // [rsp+28h] [rbp-60h]  
 \_\_int64 v19; // [rsp+30h] [rbp-58h]  
 \_\_int64 v20; // [rsp+38h] [rbp-50h]  
 \_\_int64 v21; // [rsp+40h] [rbp-48h]  
 \_\_int64 v22; // [rsp+48h] [rbp-40h]  
 unsigned \_\_int64 v23; // [rsp+58h] [rbp-30h]  
  
 v23 = \_\_readfsqword(0x28u);  
 v1 = v16;  
 read\_six\_numbers(a1, v16);  
 for ( i = 1LL; ; ++i )  
 {  
 if ( (unsigned int)(\*v1 - 1) > 5 )  
 explode\_bomb();  
 if ( (int)i > 5 )  
 break;  
 v3 = i;  
 do  
 {  
 if ( \*v1 == v16[v3] )  
 explode\_bomb();  
 ++v3;  
 }  
 while ( (int)v3 <= 5 );  
 ++v1;  
 }  
 for ( j = 0LL; j != 6; ++j )  
 {  
 v5 = v16[j];  
 v6 = 1;  
 v7 = &node1;  
 if ( v5 > 1 )  
 {  
 do  
 {  
 v7 = (\_QWORD \*)v7[1];  
 ++v6;  
 }  
 while ( v6 != v5 );  
 }  
 \*(&v17 + j) = (\_\_int64)v7;  
 }  
 v8 = v17;  
 v9 = v18;  
 \*(\_QWORD \*)(v17 + 8) = v18;  
 v10 = v19;  
 \*(\_QWORD \*)(v9 + 8) = v19;  
 v11 = v20;  
 \*(\_QWORD \*)(v10 + 8) = v20;  
 v12 = v21;  
 \*(\_QWORD \*)(v11 + 8) = v21;  
 v13 = v22;  
 \*(\_QWORD \*)(v12 + 8) = v22;  
 \*(\_QWORD \*)(v13 + 8) = 0LL;  
 v14 = 5;  
 do  
 {  
 if ( \*(\_DWORD \*)v8 > \*\*(\_DWORD \*\*)(v8 + 8) )  
 explode\_bomb();  
 v8 = \*(\_QWORD \*)(v8 + 8);  
 --v14;  
 }  
 while ( v14 );  
 return v23 - \_\_readfsqword(0x28u);  
}

## 解析

phase\_6的代码逻辑就比较冗长了，当然一部分是因为其将全部流程均打包在了一个函数中。

将程序的流程分为以下几部分：

1. 读取六个整数存储于v16指向的数组中，用第一个for循环进行约束；
2. 第二个for循环进行了链表转写的操作；
3. 一系列非常奇怪的赋值语句；
4. do-while循环再次进行约束。

### Part 1

第一个for循环的约束逻辑：伪C部分形如

for ( i = 1LL; ; ++i )  
 {  
 if ( (unsigned int)(\*v1 - 1) > 5 )  
 explode\_bomb();  
 if ( (int)i > 5 )  
 break;  
 v3 = i;  
 do  
 {  
 if ( \*v1 == v16[v3] )  
 explode\_bomb();  
 ++v3;  
 }  
 while ( (int)v3 <= 5 );  
 ++v1;  
 }

依然还是编译-反编译带来的语义崩坏通病~~有点类似于将语言A的句子翻译成语言B再翻译回来的感觉~~。梳理其逻辑可以转写如下：

* \*v1访问过的所有元素（也就是v16的所有元素）数值必须在 ；
* v1和&v16[v3]都是指针，其中v3在do-while循环前的初始值必为v1+1，相当于其访问元素初始下标比v1大1。进而，do-while循环的约束拓展、简写后可写作“v16中的元素两两不同”。

合并起来就是：**v16[6]中唯一地存放了1~6这六个数字。**

### Part 2

再把对应部分的伪代码搬过来：

for ( j = 0LL; j != 6; ++j )  
 {  
 v5 = v16[j];  
 v6 = 1;  
 v7 = &node1;  
 if ( v5 > 1 )  
 {  
 do  
 {  
 v7 = (\_QWORD \*)v7[1];  
 ++v6;  
 }  
 while ( v6 != v5 );  
 }  
 \*(&v17 + j) = (\_\_int64)v7;  
 }

首先又涉及到了一个栈变量node1，查看其内容：  
node1 可以发现其邻域还有node2，node3等，结合题目描述推测其与链表有关。但链表的struct在此处已无法还原，所以只能看循环体内对内存的操作：

* 先令v5 = v16[j](~~j in range(6)~~)，显然若v5 <= 1，则直接令v17[j] = &node1；
* 否则进入do-while循环，发现其寻址方式特别有趣——v7 = (\_QWORD \*)v7[1]，强制类型转换由于没有发生截断（原来是\_\_int64，现在QWORD四字类型依然是64位，相当于将该位置开始的8Byte数据按指针的方式解读，参考C++的reinterpret\_cast）故近似可忽略（但注意此时其已经具备了“指针”的属性，可以执行下标访问等操作）。 随后结合operator[]，相当于v7移动到了根据v7后的下一个变量（即8Byte后）的“数据地址”。

结合上图可能会更加直观：  
如假设v5 = 2，相当于其只前进了一次，首先v7 = 0x405330，其获取0x405338开始8Byte的数据，由于x86-64架构小端序的特性，其得到的数据是0x405340。  
随后，v7移动到这个“获得”的数据所对应的地址上，即现在v7就是0x405340。

不难发现这一部分的链表~~还算比较友好~~都是“连续”存放的，**除了node5的地址指向了0x405210**，回滚后发现其就是node6，该node存放“下一个结点地址”是0（NULL,~~nullptr~~）。

由于这个Part的前置条件就是v16[6]覆盖了1~6的所有数字，因此此处不应出现越界之类的意外情况。

### Part 3

这一段的伪代码会让人看着非常摸不着头脑：

v8 = v17;  
 v9 = v18;  
 \*(\_QWORD \*)(v17 + 8) = v18;  
 v10 = v19;  
 \*(\_QWORD \*)(v9 + 8) = v19;  
 v11 = v20;  
 \*(\_QWORD \*)(v10 + 8) = v20;  
 v12 = v21;  
 \*(\_QWORD \*)(v11 + 8) = v21;  
 v13 = v22;  
 \*(\_QWORD \*)(v12 + 8) = v22;  
 \*(\_QWORD \*)(v13 + 8) = 0LL;

其涉及了大量乱飞的变量名，而这些变量名看似“空穴来风”。其源代码大概率亦并非如此编写~~日常甩锅给IDA/反编译~~。

推测这大概率是构建struct的某个函数被重复调用、内联展开后的结果，解析见下。

破局之法有二：首先可以试着回到反汇编代码头，查看其定义的注释：

\_\_int64 v8; // rbx  
 \_\_int64 v9; // rax  
 \_\_int64 v10; // rdx  
 \_\_int64 v11; // rax  
 \_\_int64 v12; // rdx  
 \_\_int64 v13; // rax  
 int v14; // ebp  
 int v16[8]; // [rsp+0h] [rbp-88h] BYREF  
 \_\_int64 v17; // [rsp+20h] [rbp-68h]  
 \_\_int64 v18; // [rsp+28h] [rbp-60h]  
 \_\_int64 v19; // [rsp+30h] [rbp-58h]  
 \_\_int64 v20; // [rsp+38h] [rbp-50h]  
 \_\_int64 v21; // [rsp+40h] [rbp-48h]  
 \_\_int64 v22; // [rsp+48h] [rbp-40h]

或者直接看栈空间：  
varstk

可以发现两点：

* v8-v13是“名义上”的变量，但它们实际并不存在于C源程序中，注释亦指明之——它们实际上是寄存器在伪C中的别名。 > 这也是一些反编译代码“废话文学”的来源，寄存器实际上是作为部分变量/返回值间赋值的“中间人”。
* v17-v22并非互相独立的变量，实际上它们在内存空间中的位置是连续的（其类型视为\_\_int64的情况下）。事实上可以认为存在一个数组arr，其中v17就是arr[0]，v18就是arr[1]，以此类推，v22就是arr[5]。

还记得此前Part 2中给\*(&v17 + j)赋值的语句吗？结合上文分析，按变量名的角度理解，Part2等价于在将node0的地址写入v17，node1的地址写入v18，等等。  
那此处的一些赋值语句就不再让人完全发懵了：  
\*(\_QWORD \*)(v17 + 8)其实就是node0中存放下一结点地址的该部分内存，对之赋值v18~~说c话~~就是：

node0.next = &node(???);

而这里的(???)就是之前\*&(v17 + 1)被重写时使用的node地址，追溯回去也就是node(v16[j])。

至此分析结束。  
这段代码(Part2 + Part3)的作用是实现了链表“重组”：先建立一个链表，然后把链表中的地址用一个数组存储起来，再根据另一个index式的数组将链表重新连接。写成C语言代码应该长这样：

struct Node  
{  
 int data;  
 struct Node\* next;  
};  
  
struct Node\* head = /\*\*/;  
int index\_ls[6] = /\*\*/;  
  
struct Node\* arr[6] = {};  
struct Node\* cur = NULL,\*tp = NULL;  
  
for(int i=0;i<6;++i)  
{  
 cur = head;  
 int temp = index\_ls[i];  
 if(temp)  
 {  
 for(int j=0;j<temp;++j)  
 {  
 cur = cur->next;  
 }  
 }  
 arr[i] = cur;  
}  
  
for(int i=1;i<6;++i)  
{  
 cur = arr[i];  
 tp = cur->next;  
 cur->next =   
}

### Part 4

现在v8定位到了v17处，而此处其比较的约束条件是\*(\_DWORD \*)v8 <= \*\*(\_DWORD \*\*)(v8 + 8)。  
此处\_DWORD \*的强制类型转换说明：比较时，将其按2字（4字节，32位）的方式解读。但v8本身具有\_\_int64的属性（是一个地址），其+8的偏移量相当于从数据部分（一个int）“偏”到了地址部分，再进行二阶解引用，相当于解引用了“下一个结点”的int。  
总而言之，翻译过来就是约束**按链表的访问顺序，其对应数据区的int应是非递减的**。  
因此我们直接查看node0-node5对应的int数据区，将之按非递减序排列，对应的下标+1即为解。

数据区的DWORD表示：

| node编号 | 内存数据 |
| --- | --- |
| 1 | ad 03 00 00 |
| 2 | e7 03 00 00 |
| 3 | 1b 01 00 00 |
| 4 | 1b 01 00 00 |
| 5 | 8c 01 00 00 |
| 6 | db 00 00 00 |

按小端序读取，不难得出一种非递减序为6 3 4 5 1 2。

解为：6 3 4 5 1 2或6 4 3 5 1 2

# 阶段七（“隐藏关”）

## 何谓“隐藏”

其一，其函数名为secret\_phase(bushi)  
其二，该阶段的触发不是显式进行的，而是依赖了一个名为phase\_defused的函数，其在每个阶段结束后被调用：

unsigned \_\_int64 phase\_defused()  
{  
 char v1; // [rsp+8h] [rbp-70h] BYREF  
 char v2; // [rsp+Ch] [rbp-6Ch] BYREF  
 char v3[88]; // [rsp+10h] [rbp-68h] BYREF  
 unsigned \_\_int64 v4; // [rsp+68h] [rbp-10h]  
  
 v4 = \_\_readfsqword(0x28u);  
 send\_msg(1LL);  
 if ( num\_input\_strings == 6 )  
 {  
 if ( (unsigned int)\_\_isoc99\_sscanf(&unk\_405910, "%d %d %s", &v1, &v2, v3) == 3  
 && !(unsigned int)strings\_not\_equal(v3, "DrEvil") )  
 {  
 puts("Curses, you've found the secret phase!");  
 puts("But finding it and solving it are quite different...");  
 secret\_phase();  
 }  
 puts("Congratulations! You've defused the bomb!");  
 puts("Your instructor has been notified and will verify your solution.");  
 }  
 return v4 - \_\_readfsqword(0x28u);  
}

注意到其前置的触发条件是num\_input\_strings == 6，而num\_input\_strings变量是在read\_line函数（读取字符串）中被更改的。  
故总结起来就是其**必须在完成前六关后才能触发**。

里层if的约束条件为：

* 先对一个奇怪的位置0x405910开始的字符串进行了sscanf，读取到v1、v2、v3中，其分别为整型、整型、字符串，要求scanf返回值为3；
* strcmp(v3,"DrEvil") == 0（等价）。

回到栈空间及汇编代码中，可以看到以下片段：  
stack\_partition

这个0x405910的位置特殊在于，其恰好在char input\_strings[240]的隔壁。而input\_strings只在read\_line函数中有涉及过，查看其反编译代码：

实践证明，不求甚解的可行性会随着时间、形势和目标等而不断转化。如最初我们认为对read\_line函数不需要过于深究，但现在仍需回到其中才能探明“隐藏关”的启动方法。

const char \*read\_line()  
{  
 int v0; // ebp  
 const char \*v1; // rbx  
 int v2; // eax  
 int v4; // eax  
  
 if ( !skip() )  
 {  
 if ( infile == (FILE \*)stdin )  
 {  
 puts("Error: Premature EOF on stdin");  
 exit(8);  
 }  
 if ( getenv("GRADE\_BOMB") )  
 exit(0);  
 infile = (FILE \*)stdin;  
 if ( !skip() )  
 {  
 puts("Error: Premature EOF on stdin");  
 exit(0);  
 }  
 }  
 v0 = num\_input\_strings;  
 v1 = &input\_strings[80 \* num\_input\_strings];  
 v2 = strlen(v1);  
 if ( v2 > 78 )  
 {  
 puts("Error: Input line too long");  
 v4 = num\_input\_strings++;  
 strcpy(&input\_strings[80 \* v4], "\*\*\*truncated\*\*\*");  
 explode\_bomb();  
 }  
 input\_strings[80 \* v0 - 1 + v2] = 0;  
 num\_input\_strings = v0 + 1;  
 return v1;  
}

不难推断，input\_strings是一个存放了输入的char[???]，但显然???不应为240（因为每个字符串都预留了80的空间）；IDA将之标记为char[240]依然和编译-反编译过程等有关（~~中途杀出来的变量搅混水~~）。  
但既然如此，鉴于input\_strings的起始地址为0x405820，则对于下标为i的题目，存放输入字符串的地址就是0x405820 + i\*80。  
所以0x405910恰好就是下标为3，即阶段四时我们的输入。

Surprise m\*\*\*\*\* f\*\*\*er!(bushi)

确定了sscanf的作用对象后，回到阶段四的答案（两个整数，以66 2为例），在其后再追加一个DrEvil即可触发隐藏关；隐藏关的输入则在第七行正常进行。

## 反编译代码

### secret\_phase

\_\_int64 secret\_phase()  
{  
 const char \*line; // rdi  
 int v1; // ebx  
  
 line = (const char \*)read\_line();  
 v1 = strtol(line, 0LL, 10);  
 if ( (unsigned int)(v1 - 1) > 0x3E8 )  
 explode\_bomb();  
 if ( (unsigned int)fun7((\_\_int64)&n1, v1) != 5 )  
 explode\_bomb();  
 puts("Wow! You've defused the secret stage!");  
 return phase\_defused();  
}

### fun7

\_\_int64 \_\_fastcall fun7(\_\_int64 a1, int a2)  
{  
 \_\_int64 result; // rax  
  
 if ( !a1 )  
 return 0xFFFFFFFFLL;  
 if ( \*(\_DWORD \*)a1 > a2 )  
 return 2 \* (unsigned int)fun7(\*(\_QWORD \*)(a1 + 8));  
 result = 0LL;  
 if ( \*(\_DWORD \*)a1 != a2 )  
 return 2 \* (unsigned int)fun7(\*(\_QWORD \*)(a1 + 16)) + 1;  
 return result;  
}

## 解析

首先secret\_phase部分还是读取了一个字符串，并将之用strtol函数转换为整型(long)。  
第一个if的约束翻译为要求 ；  
第二个if则要求fun7(&n1,v1)的返回值为5，因此顺势继续看向fun7的逻辑。

fun7接受的第一个参数是一个指针，返回值类型为\_\_int64。

这里可以发现fun7的递归调用中同样出现了“失踪”实参的情况，其原因也是\_\_fastcall下寄存器rsi的数值没有发生改变。  
理解作每次递归时a2的值不变即可（实现了其类似一个“静态”值的效果，如此亦能发挥\_\_fastcall的优越性（复用寄存器，无需重复复制变量、进出栈，效率极高）。

若a1为空(NULL,~~nullptr~~)，则返回 。

显式指定LL后缀是避免默认字面量类型int截断（最大值 ）。  
趣话C/C++的字面量：C/C++对于输入的整型字面量(literals)会根据其大小“自适应”地匹配对应的数据类型，但可以通过指定后缀来覆写之。默认取在其范围内尽可能小的类型（int-unsigned int-long long-unsigned long long）；  
```C++ auto a = 1; //a is int(1 is of int type by default) auto b = -3; //b is int(-3 is of int type by default) auto c = 1LL; //c is long long(default is overridden by explicit literal suffix)

auto d = 0xFFFFFFFF; //d is unsigned int(since int cannot contain such number as 0xFFFFFFFF) auto e = 0x7FFFFFFF; //e is int(actually maximum possible value of int) auto f = 0xFFFFFFFFFFFFFFFF; //f is unsigned long long auto g = 0x7FFFFFFFFFFFFFFF; //g is long long //auto h = 0x10000000000000000; //Error: Integer constant is too big(exceeds 2^{64} - 1 limit of unsigned long long) > 需要注意的是C/C++中\*\*负号并不是字面量的一部分\*\*，字面量的类型仅取决于其整数部分，负号只是一个运算符。因此如`-0x80000000`并不会被视作一个取值最小的`int`，而会被视为取值最大的`unsigned int`取负~~巧的是运算结果依然是`0x80000000`~~：C++ static\_assert(-0x80000000 == 0x80000000); //passes static\_assert(-0xFFFFFFFFFFFFFFFF == 1); //passes, no error in constant resolution ```

回归题解，在a1不为空时，判定\*(\_DWORD \*)a1和a2的大小关系：

* 大于，返回2\*fun7(\*(\_QWORD \*)(a1 + 8))；
* 小于，返回2\*fun7(\*(\_QWORD \*)(a1 + 16)) + 1；
* 等于，返回0。

有了阶段六的链表解析经验，不难看出这也是一种链表访问的结构。

DWORD和QWORD互化还是因为内存解读方式的问题——这个链表结点的数据区类型是2字(32位)的int，指针类型则是4字。

点进n1的栈内存区，可以发现以下内容：  
n1\_stack  
由此不难发现，这是一个二叉树结构，n1就是该二叉树的根结点；  
结合地址可以判断，比较结果为大于时，返回左结点的2倍，小于时返回右子结点的2倍加1，等于时返回零。  
若待解引用的结点是空结点，返回 。

由于栈内存区已确定，逐个地址构建二叉树结构，随后按该规则遍历即可，二叉树结构~~及邪道速通法~~见下。

“抄近道”：由于节点数较多，逐个寻址较为繁琐。继续往下翻，可以发现结点名似乎有所蹊跷：  
nname\_1  
注意到n21恰好是n1的左子结点，n22恰好是n1的右子结点。  
再往下寻址+翻栈内存，可以找到n31n34、n41n48，其中n41~n48是叶结点。  
不难验证，变量名末位是在二叉树结构中层序遍历的次序。由此结合数据区内容，整理得到二叉树结构如下：  
bintree

但仅完成双链表的整理尚不够。回顾最开始的if约束，要求fun7(&n1,v1) == 5。  
而因为其中返回值传递用的都是\_\_int64，显然我们不希望其中的0xFFFFFFFF被加入计算（或者能在某一步计算被“截断”）。

又因为fun7的返回值只能是以下四种之一：2倍左结点返回值、2倍右结点返回值+1、0、0xFFFFFFFF；  
结合该二叉树只有4层，显然我们需要获得的返回值5只能由4+1，即2\*2+1获得。

继续推断，2\*2+1应再解作2\*(2\*1)+1（展开一个隐式的因子1）。按递归的逆序分析形如：

* 对于第四层的某个结点，若其访问子结点，结果必定为0xFFFFFFFF（因为其为空），而这个值不能被传递上去，因此该结点的数值只能等于v2来实现截断。由于这个条件具有充要条件的属性，因此反过来，v2的值也只能是四个叶结点所含有的int之一，此时第四层的返回值就是0；
* 对于第三层，因为第四层返回的一定是0，再将内层表达式2\*1转写为2\*((2\*0)+1)，不难看出其要求第三层的结点数据i必须小于v2，方能实现返回(2\*0)+1的效果；
* 对于第二层，往外看一层括号，显然需要让其结点数据大于v2；
* 对于第一层，同理可得其结点数据应小于v2。

除结点数据要求外，还要求该访问顺序在二叉树中可以实现。  
逐个尝试可得只有v2 = 0x2f满足要求，即v2 = 47。

解为：（不考虑对阶段四答案的追加）

47

# .txt及结果

I can see Russia from my house!  
0 1 1 2 3 5  
0 104  
66 2 DrEvil  
IOAPEG  
6 3 4 5 1 2  
47

# 彩蛋：C代码“还原”

咕咕咕，曲项向天咕