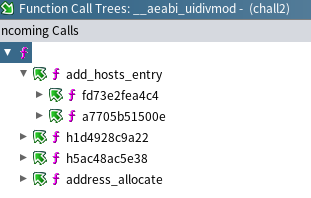
除了第八个函数使用二进制码来查找，其他都是通过Function Call Trees功能定位特定函数来进行查找的。

1. effdbd36ad56:h1d4928c9a22

effdbd36ad56反编译出来的代码调用了strlen和\_\_aeabi\_uidivmod 

先定位来\_\_aeabi\_uidivmod找到chall2中的函数



除了h1函数外其他三个函数都调用了另外的函数，暂时排除。

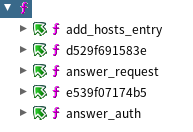
H1函数只调用了strlen和\_\_aeabi\_uidivmod。

分析语句的过程和分析第11个函数的过程类似。会发现嵌套了那么多while(true),大量使用if（）break来阻断，最后还是执行的effdbd36ad56里面的语句。

**chall2相较于chall1运用了控制流平坦化的混淆方式**。

2.g0b4866d0826:d529f691583e

通过cache\_find\_by\_addr、cache\_scan\_free和goto语句来定位函数

符合条件的只有d529f691583e，所以先考虑这个函数。

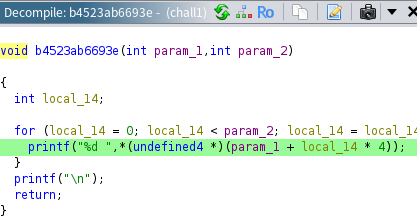
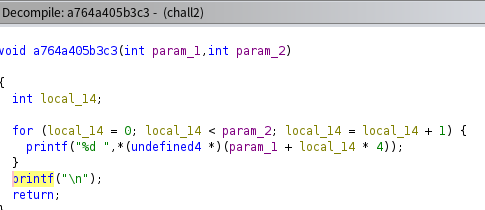
通过对比会发现代码的语义都大致相似，但是在字符串上不一样。



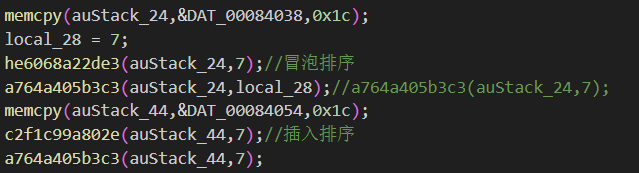
分别调用的这两个函数语句相同，但是在chall2（下图）中进行了加密处理。其中ObstrDec是一个解码/反混淆的函数。

**Chall2相较于chall1函数运用了字符串加密混淆方式。**

3. bf137dc95bb7:d64a94477ef7

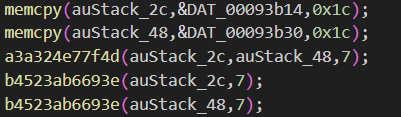
通过printf函数发现chall1和chall2中存在语句相同的函数b4523ab6693e和a764a405b3c3，而chall1的bf137dc95bb7调用了b4523ab6693e，所以通过寻找chall2中调用了a764a405b3c3的函数来定位。从而找到d64a94477ef7。

chall1和chall2中的函数的DAT（memcpy的第二个参数）都是@ 



（第一个图是chall1 bf137dc95bb7，第二个图是chall2 d64a94477ef7）

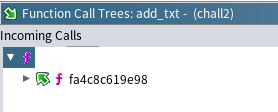
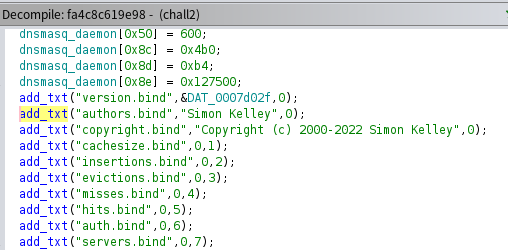
而chall2中的he6068a22de3、c2f1c99a802e分别进行了冒泡排序、插入排序，

 （这是chall1 bf137dc95bb7）

Chall1中调用a3a324e77f4d的结合了冒泡排序和插入排序的思想，分别对两个数组进行不同的排序操作。所以chall2的函数相当于对顺序进行了一个调整。

**chall2相较于chall1运用了函数交错的混淆方式**。

4. e8800448613f:fa4c8c619e98

因为chall1中的add\_txt存在大量可以定位的字符串，所以考虑通过add\_txt来定位chall2中的函数。只有这一个fa4c8c619e98函数，并且函数里面字符串与chall1中bea2330b3c12相对应。接下来考虑这个函数的语义。

通过对比可以明显看出两个函数的语句大致相同。不同部分在于：chall2会把复杂的异或、与计算变成简单的求余。例如：

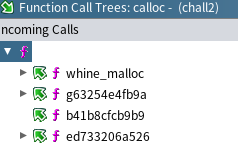
((\*dnsmasq\_daemon ^ 0xffff7fff) & \*dnsmasq\_daemon) != 0变成

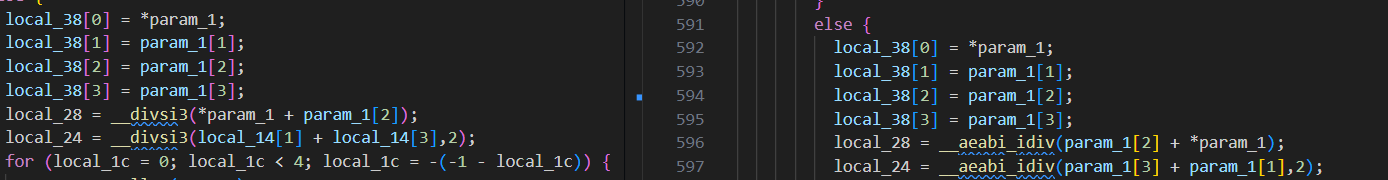
(\*dnsmasq\_daemon & 0x8000) != 0但是意思都是一样的。

所以不一样的地方只在于标识符。

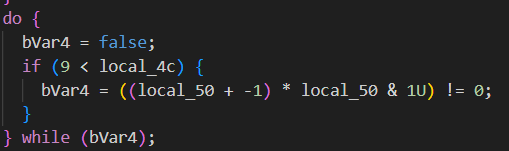
**Chall2相较于chall1使用了标识符重命名的混淆方式。**

5. bea2330b3c12:ed733206a526

通过calloc来定位chall2中的函数。其中第1、2、3个都很短，也没有调用memcpy函数，而ed733206a526调用了calloc、printf、memcpy等函数（和bea2330b3c12相对应），所以只考虑这一个函数。

这两个函数有很多地方语句都是相同的，例如：

只是通过goto语句来让执行顺序发生改变。

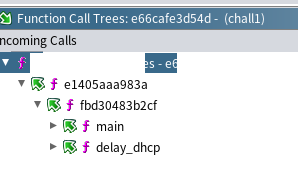


这里只执行一次就会跳出循环，是虚假控制流。类似这样的代码在ed733206a526中还有好几处。

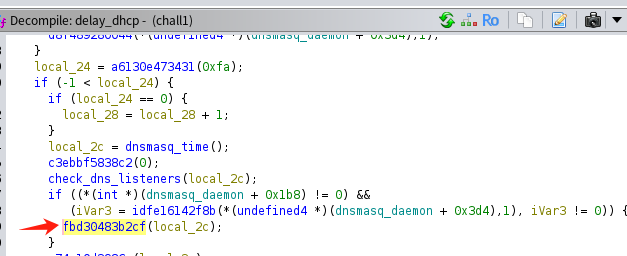
**Chall2相较于chall1使用了虚假控制流的混淆方式。**

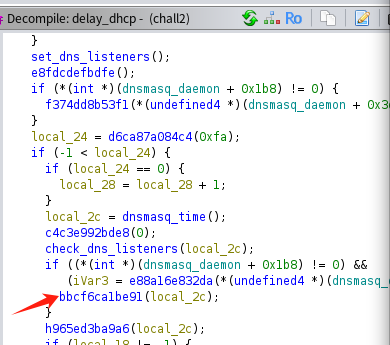
6. e66cafe3d54d: g803863bb097

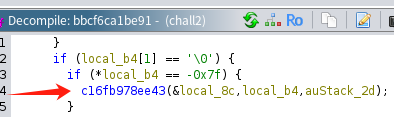
e66cafe3d54d无法通过自己调用的函数来找到对应的代码，只能通过看调用e66cafe3d54d 的函数来找。

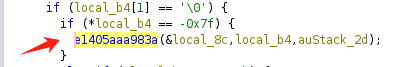
直接找有名字的delay\_dhcp，可以在chall2函数中搜索到。

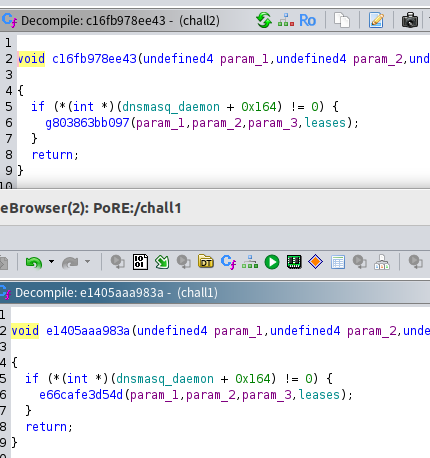
先确定fbd函数在chall2中的对应函数：



 所以能找到是bbc..函数，接下来再看bbc函数调用的函数。0





然后能找到e1405对应的函数，接下来继续同样的操作。

所以能找到和e66cafe3d54d相对应的函数g803863bb097。

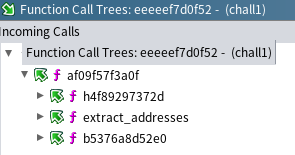
这个函数无法反编译，猜测是进行了自修改。

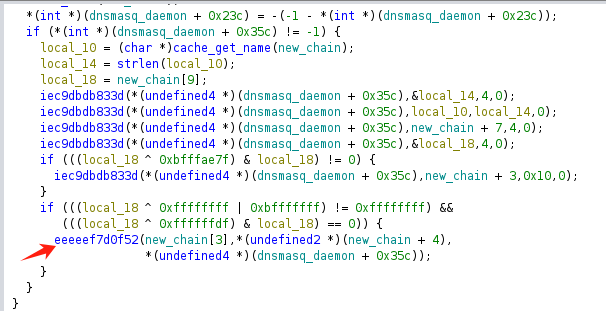
**Chall2相较于chall1运用了自修改代码混淆方式。**

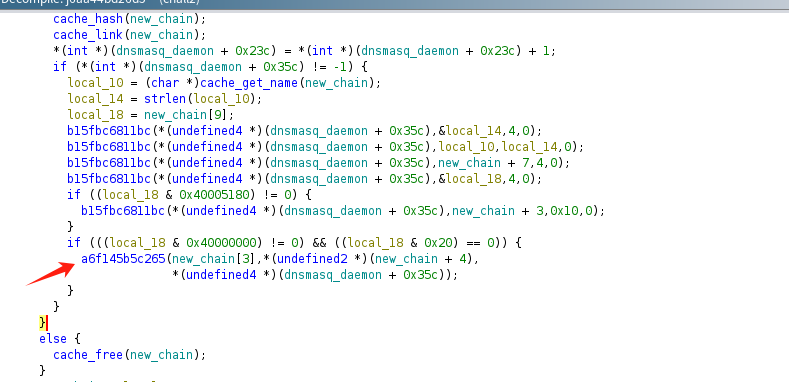
7. eeeeef7d0f52: a6f145b5c265

eeeeef7d0f52调用了iec9dbdb833d函数，在chall2中能找到和iec9dbdb833d语句完全相同的函数，但是无法从调用这个函数的函数中找到和eeeeef7d0f52语义相同的函数，所以这个方法不行。

变成通过寻找调用eeeeef7d0f52的函数，只有af09f57f3a0f。然后通过a函数里面的cache\_\_link能够找到chall2中完全和a函数语义相同的函数。



chall1

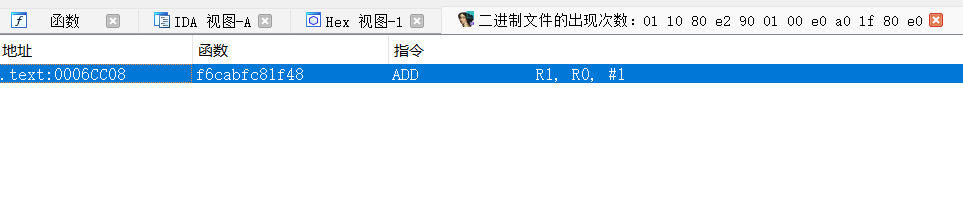
chall2

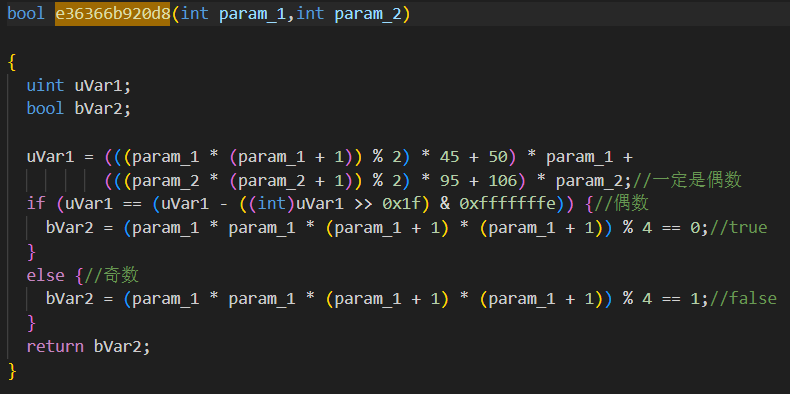
相同的地方也会调用相同的函数，函数的参数也是一致的。

但是a6f145b5c265的反编译函数无法生成，猜测可能进行了一些自修改。

**chall2相较于chall1运用了自修改代码的混淆方式**。

8. e36366b920d8:f6cabfc81f48

左边没有调用任何参数，所以用二进制码来定位。截取一段左边函数的二进制码01 10 80 E2 90 01 00 E0 A0 1F 80 E0，然后在chall2中定位，发现只有一个函数f6cabfc81f48能够对上，那么直接开始分析这个函数

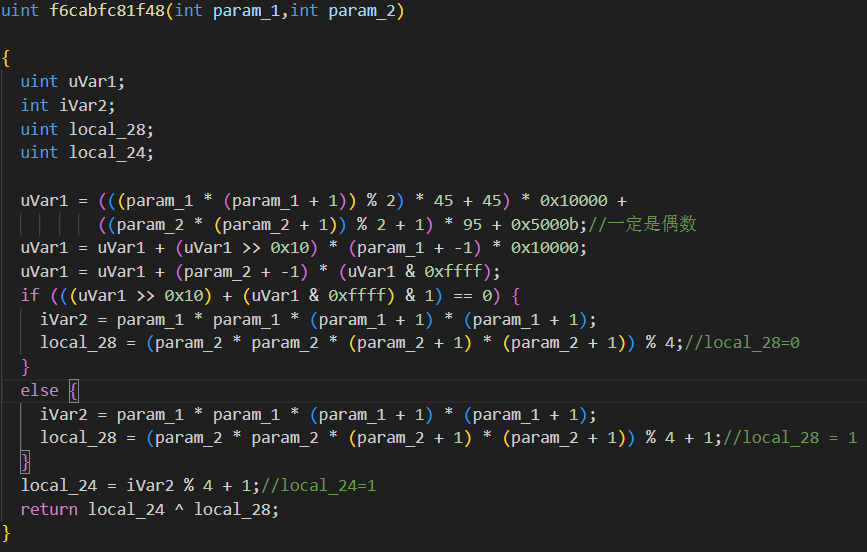


分析左边函数，param\_1和param\_2都是偶数，那么uVar1一定是偶数，如果param\_1是奇数，那么(param\_1 + 1)) % 2)==0，+50后是偶数，(((param\_1 \* (param\_1 + 1)) % 2) \* 45 + 50) \* param\_1是偶数。Param\_2的计算式子也同理，最后一定是偶数。所以uVar1最后得到的一定是偶数。

If语句里面的判断条件，(int)uVar1 >> 0x1f是uVar1的符号位，&0xfffffffe是为了看(uVar1 - ((int)uVar1 >> 0x1f)是不是偶数。

代入偶数会发现判断条件是true，代入奇数会发现判断条件false

(param\_1 \* param\_1 \* (param\_1 + 1) \* (param\_1 + 1)) % 4一定会等于0，所以式子可以化简成：if(uVar1是偶数) return true;否则返回false

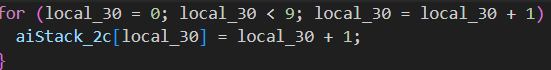
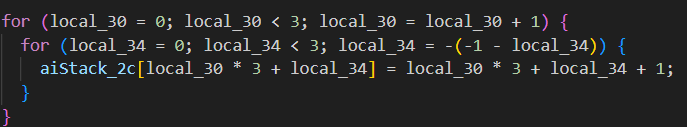


再看右边的函数,同样分析出来uVar1是偶数。然后判断uVar1是否是偶数。而local\_24==1，最后返回的结果是local\_28的取反，只要符合if的条件就返回true，不符合返回false。

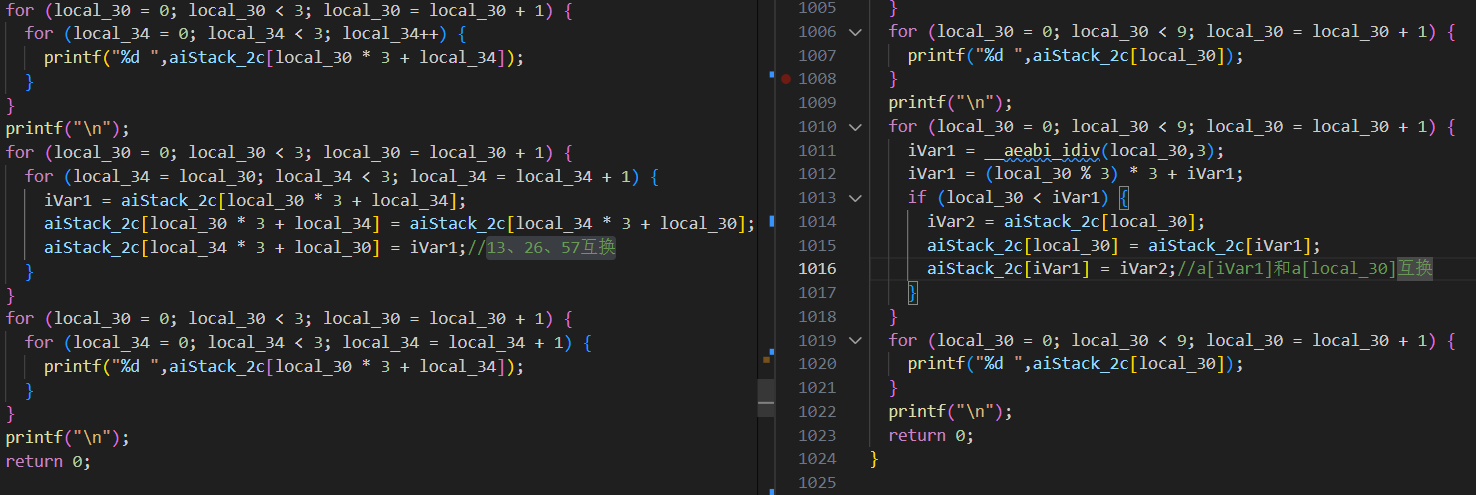
**相较于chall1中的函数使用了变量分割的混淆方式**。把一个布尔值分割成两个变量local\_28和local\_24(iVar2)。根据分析知道最后的结果是一致的。

9. cd9b0a983e07:j6bald8644b5

通过printf("%d ",aiStack\_2c[local\_30 \* 3 + local\_34])和printf("\n")来定位,输出语句里面需要数值乘以三，所以能够先找到大致符合的函数j6bald8644b5

第二函数中和第一个函数等价。直接从循环开始分析可以知道都是给数组分配数字

即a[i]=i(i=0,1,2…,8)



同理第二个图中第一、三个函数的循环也是等价的。相当于第二个函数对第一个函数进行了压缩。

左边的local\_30 \* 3 + local\_34/ local\_34\* 3 + local\_30都是为了保证数组的下标一致。

分析图二的左边第二个循环，进行逐个分析，会发现函数实现了数组a的13、26、57互换（都是下标）。然后分析右边的第二个循环，只有在local\_30=1,2,5时才会进入if语句进行互换，所以和左边语义一致。

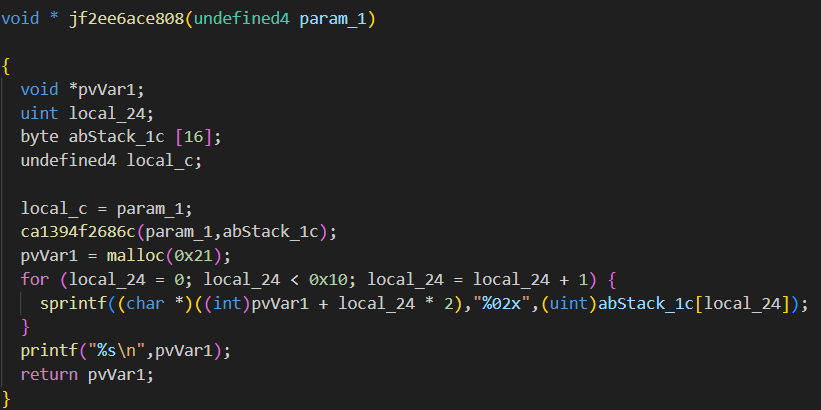
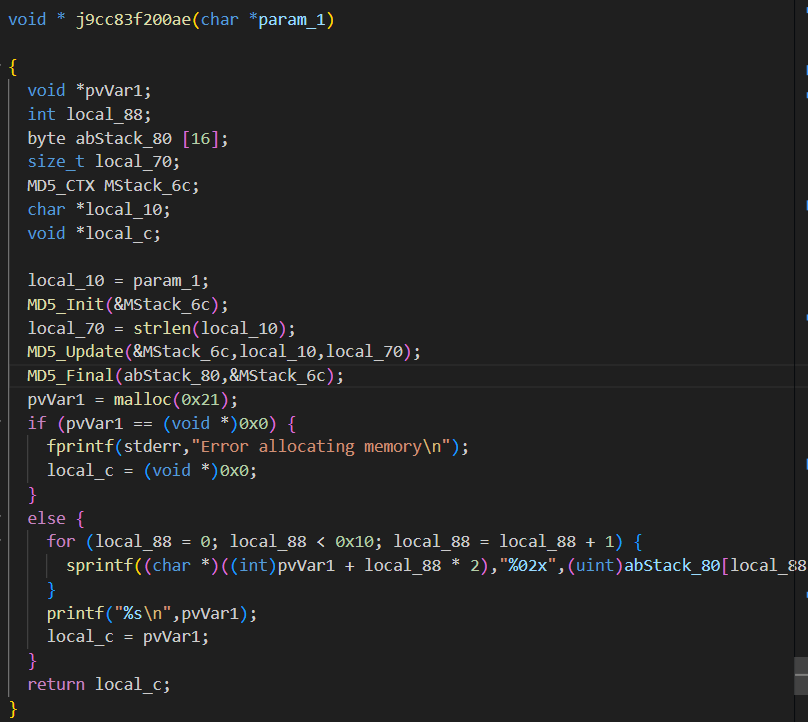
而这个函数又把一个3\*3的数组压缩成了一维的数组，所以使用了数组重构。

**相较于chall1，运用了数组重构的混淆方式**。

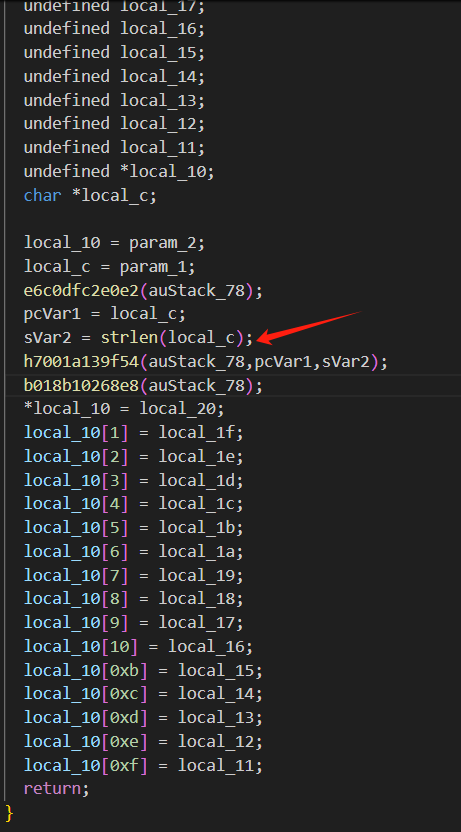
10. j9cc83f200ae:jf2ee6ace808

通过malloc(0x21)来定位，能够先找到jf2ee6ace808

对比第二个函数和第一个函数(下面是第二个)



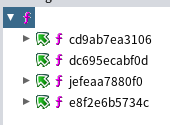
明显可以看得出来pvVar1 = malloc(0x21);以及之后的语句语义都相同，唯一不同的点在于第一个函数调用了strlen，但是第二个函数没有看到strlen，而是调用了ca1394f2686c。现在需要查看ca1394f2686c：

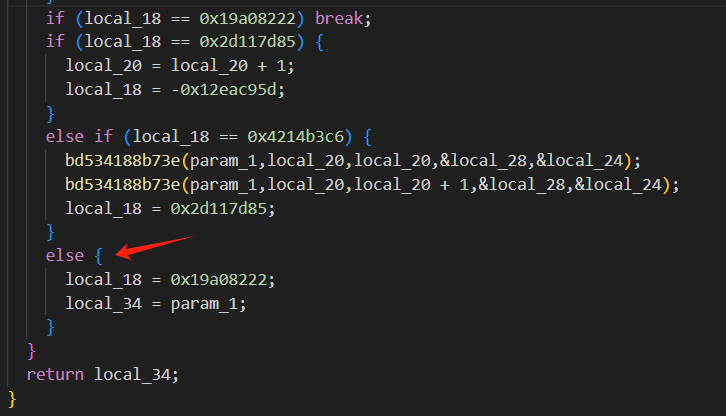
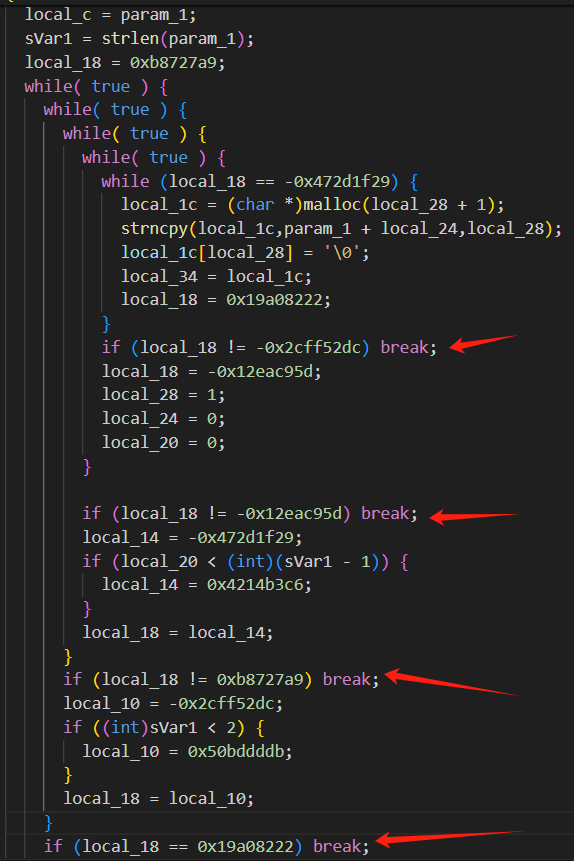


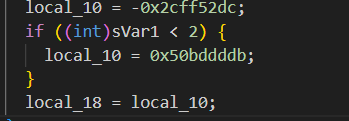
而右边函数没有调用MD5的一系列函数，都是在ca1394f2686c里面实现的。

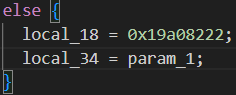
**Chall2函数相较于chall1运用了移除库函数的混淆方式**

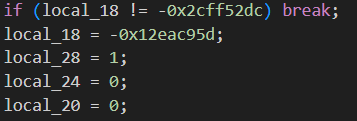
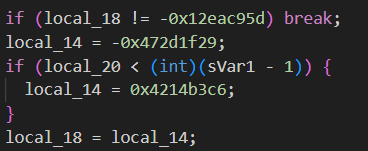
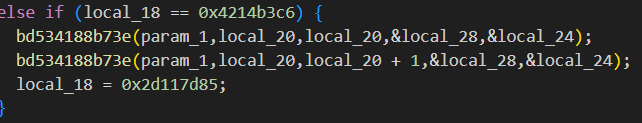
11. a387f1ff72e9:dc695ecabf0d

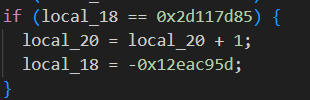
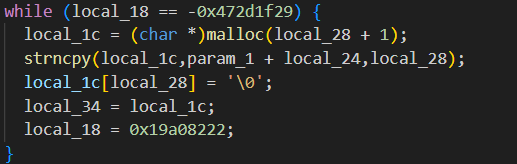
通过strncpy、malloc来定位，符合条件的只有dc695ecabf0d,所以先考虑这个函数。

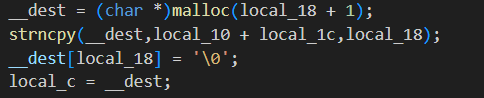


重点关注break语句。第一次进入while循环，会发现会直接跳出三个while，然后执行下列语句：。

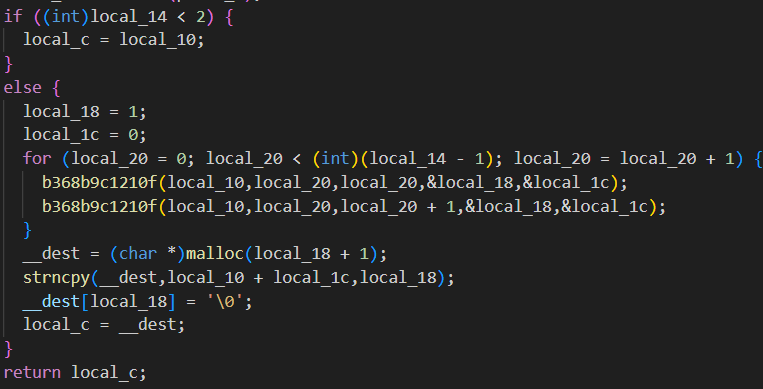
如果sVar1<2，那么local\_18 == 0x50bddddb,然后直接执行。接下来再次进入while循环，会发现直接跳出了循环没有做任何事情。然后返回 param\_1.符合if-else语句。

如果sVar1 >= 2，那么local\_18 == -0x2cff52dc,然后会执行，local\_18 = -0x12eac95d,接下来继续执行会发现当strlen(param\_1)-1>0(local\_20)的时候会执行

，执行完了以后会进入。然后再次判断strlen(param\_1)-1和local\_20的大小关系，只要strlen(param\_1)-1>local\_20，那么就会一直循环执行上面两图。当strlen(param\_1)-1<=local\_20时，执行。

这一段代码显然和chall1 a387f1ff72e9的下面部分代码相同。

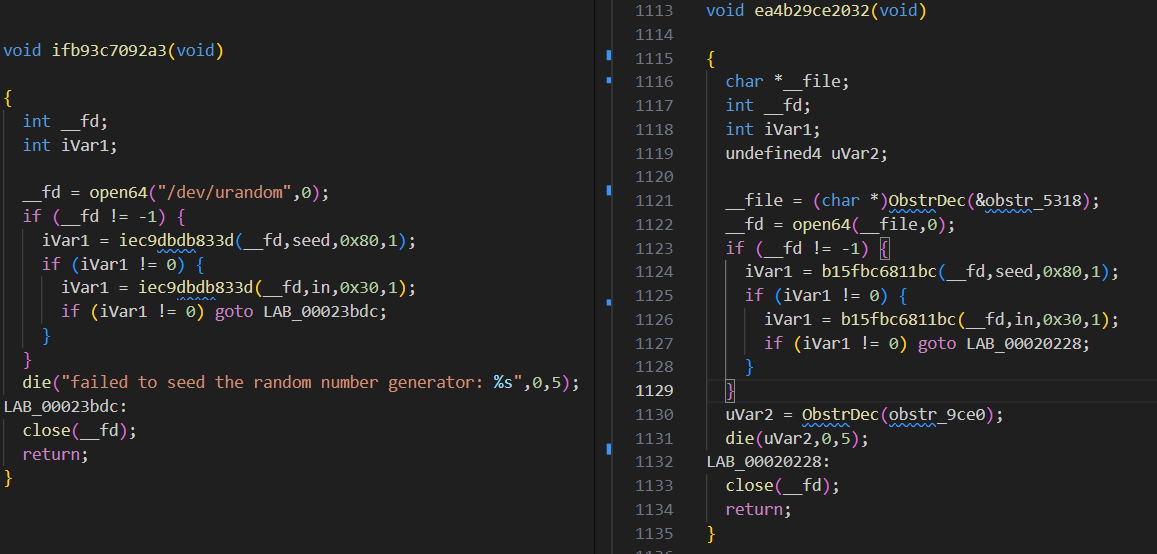
现在从整体上来看，我们可以发现chall2 dc695ecabf0d进行了控制流平坦化的混淆，if() break;相当于switch：break。其实本质上还是在执行chall1函数里面的循环：



综上所述**，chall2相较于chall1运用了控制流平坦化的混淆方式**。

12. ifb93c7092a3:ea4b29ce2032

通过open64和close来定位出ea4b29ce2032



ObstrDec是一个解码/反混淆的函数，

**相较于chall1函数，运用了字符串加密的混淆方式。**