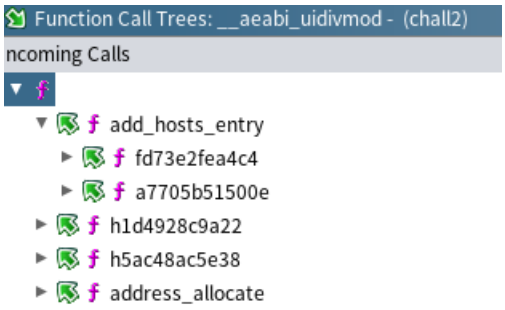
除了第八个函数使用二进制码来查找，其他都是通过 Function Call Trees 功能定位特定函数来进行查找的。

1. effdbd36ad56:h1d4928c9a22

effdbd36ad56 反编译出来的代码调用了 strlen 和 __aeabi_uidivmod

```
sVar3 = strlen(local_24);
__aeabi_uidivmod(local_128,sVar3);
```

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。

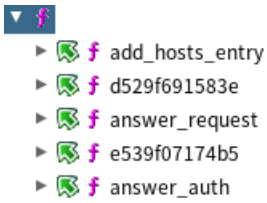
H1 函数只调用了 strlen 和__aeabi_uidivmod。

分析语句的过程和分析第 11 个函数的过程类似。最后会发现嵌套了那么多 while(true),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。

```
f9fe79ac6c84(0x1c,"%s is a CNAME, not giving it to the DHCP lease of %s",param_1,
             *(undefined4 *)(dnsmasq_daemon + 1000));
```

```
uVar2 = ObstrDec(&obstr_9676);
ff323c72c9c2(0x1c,uVar2,param_1,*(undefined4 *)(dnsmasq_daemon + 1000));
```

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

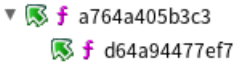
Decompile: a764a405b3c3 - (chall2)

```
void a764a405b3c3(int param_1,int param_2)

{
  int local_14;

  for (local_14 = 0; local_14 < param_2; local_14 = local_14 + 1) {
    printf("%d ",*(undefined4 *)(param_1 + local_14 * 4));
  }
  printf("\n");
  return;
}
```

Decompile: b4523ab6693e - (chall1)

```
void b4523ab6693e(int param_1,int param_2)

{
  int local_14;

  for (local_14 = 0; local_14 < param_2; local_14 = local_14
    printf("%d ",*(undefined4 *)(param_1 + local_14 * 4));
  }
  printf("\n");
  return;
}
```

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@

```
memcpy(auStack_2c,&DAT_00093b14,0x1c);
memcpy(auStack_48,&DAT_00093b30,0x1c);
```

```
memcpy(auStack_24,&DAT_00084038,0x1c);
local_28 = 7;
he6068a22de3(auStack_24,7);//冒泡排序
a764a405b3c3(auStack_24,local_28);//a764a405b3c3(auStack_24,7);
memcpy(auStack_44,&DAT_00084054,0x1c);
c2f1c99a802e(auStack_44,7);//插入排序
a764a405b3c3(auStack_44,7);
```

（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，

```
memcpy(auStack_2c,&DAT_00093b14,0x1c);
memcpy(auStack_48,&DAT_00093b30,0x1c);
a3a324e77f4d(auStack_2c,auStack_48,7);
b4523ab6693e(auStack_2c,7);
b4523ab6693e(auStack_48,7);
```

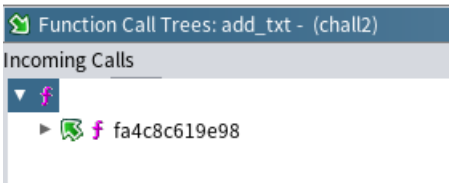
(这是 chall1 bf137dc95bb7)

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。

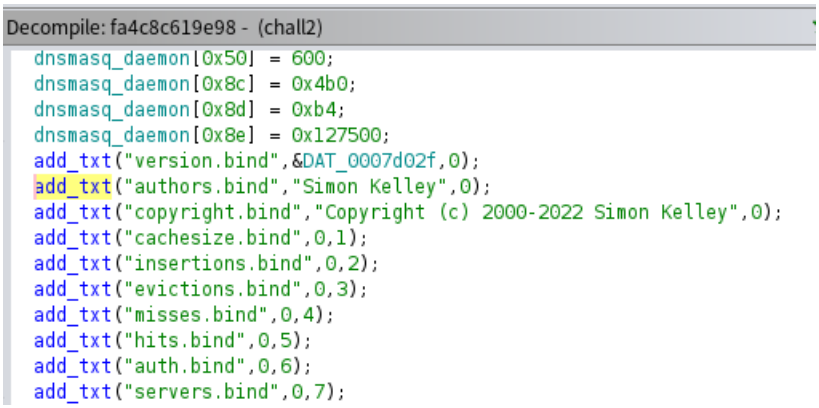
**chall2 相较于 chall1 运用了函数交错的混淆方式。**

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：

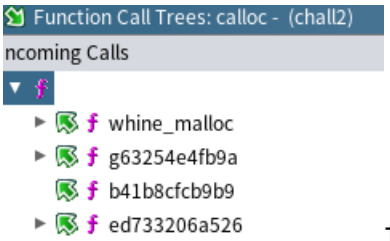
((*dnsmasq_daemon ^ 0xffff7fff) & *dnsmasq_daemon) != 0 变成

(*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。

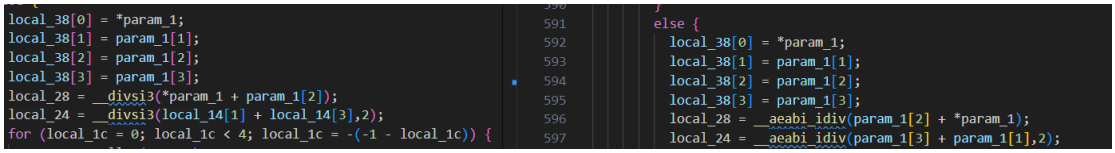
**Chall2 相较于 chall1 使用了标识符重命名的混淆方式。**

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。

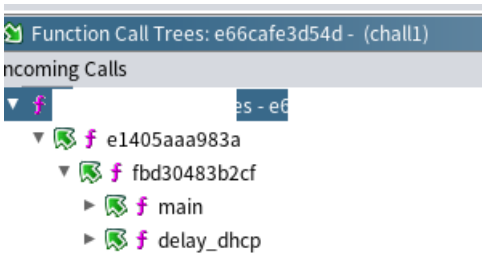
```
do {
  bVar4 = false;
  if (9 < local_4c) {
    bVar4 = ((local_50 + -1) * local_50 & 1U) != 0;
  }
} while (bVar4);
```

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。**

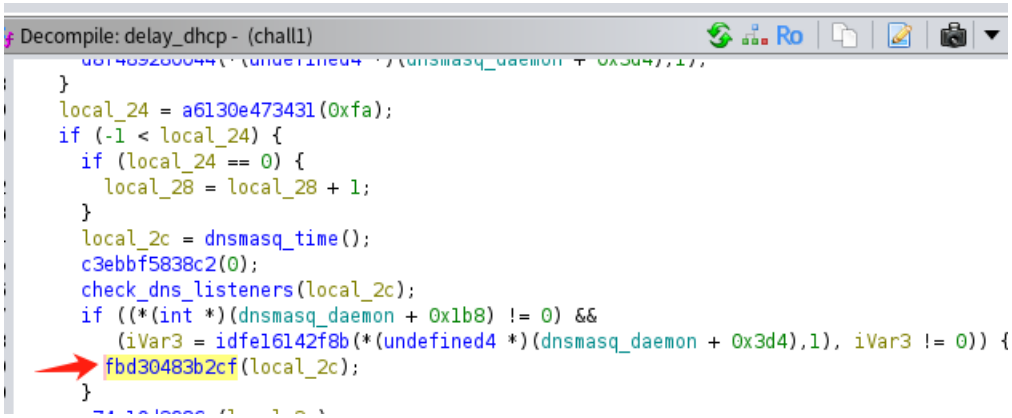
6. e66cafe3d54d: g803863bb097

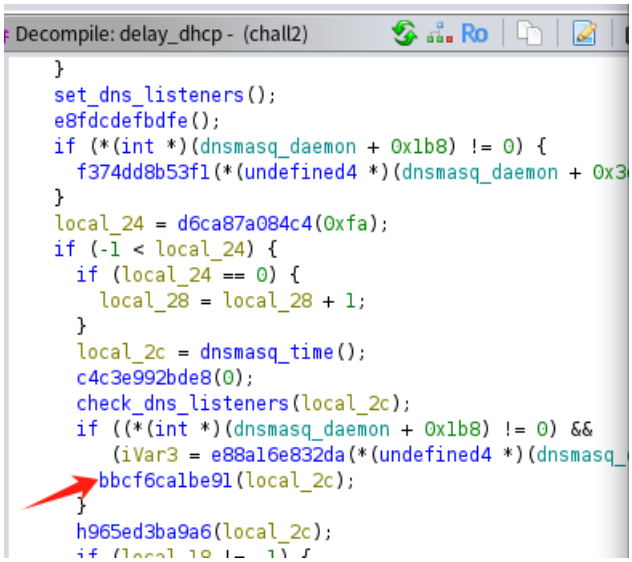
e66cafe3d54d 无法通过自己调用的函数来找到对应的代码，只能通过看调用 e66cafe3d54d 的函数来找。



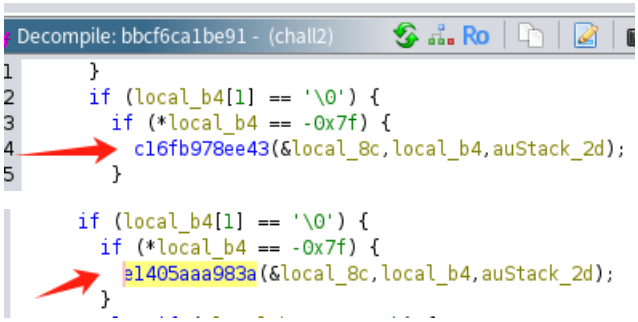

直接找有名字的 delay_dhcp，可以在 chall2 函数中搜索到。

先确定 fbd 函数在 chall2 中的对应函数：

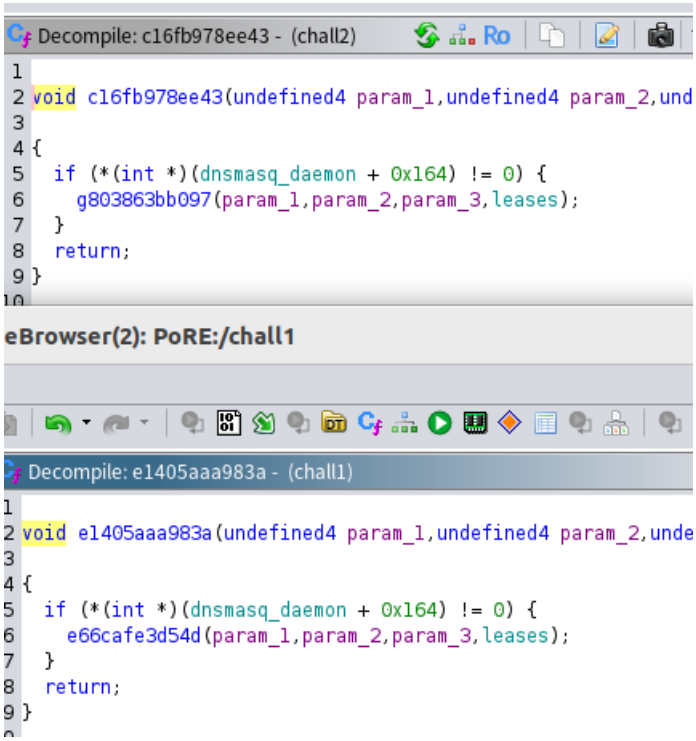



所以能找到是 bbc..函数，接下来再看 bbc 函数调用的函数。0




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。

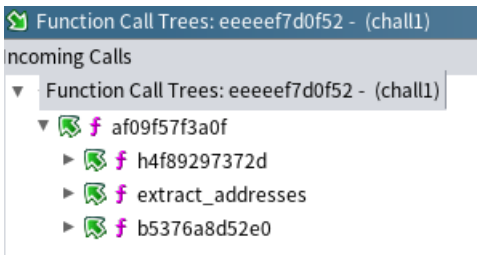
这个函数无法反编译，猜测是进行了自修改。

**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。




```
  *(int *)(dnsmasq_daemon + 0x23c) = -(-1 - *(int *)(dnsmasq_daemon + 0x23c));
  if (*(int *)(dnsmasq_daemon + 0x35c) != -1) {
    local_10 = (char *)cache_get_name(new_chain);
    local_14 = strlen(local_10);
    local_18 = new_chain[9];
    iec9dbdb833d(*(undefined4 *)(dnsmasq_daemon + 0x35c),&local_14,4,0);
    iec9dbdb833d(*(undefined4 *)(dnsmasq_daemon + 0x35c),local_10,local_14,0);
    iec9dbdb833d(*(undefined4 *)(dnsmasq_daemon + 0x35c),new_chain + 7,4,0);
    iec9dbdb833d(*(undefined4 *)(dnsmasq_daemon + 0x35c),&local_18,4,0);
    if (((local_18 ^ 0xbfffae7f) & local_18) != 0) {
      iec9dbdb833d(*(undefined4 *)(dnsmasq_daemon + 0x35c),new_chain + 3,0x10,0);
    }
    if (((local_18 ^ 0xffffffff | 0xbfffffff) != 0xffffffff) &&
       (((local_18 ^ 0xfffffffdf) & local_18) == 0)) {
      eeeeef7d0f52(new_chain[3],*(undefined2 *)(new_chain + 4),
                   *(undefined4 *)(dnsmasq_daemon + 0x35c));
    }
  }
}
```

chall1

```
    cache_hash(new_chain);
    cache_link(new_chain);
    *(int *)(dnsmasq_daemon + 0x23c) = *(int *)(dnsmasq_daemon + 0x23c) + 1;
    if (*(int *)(dnsmasq_daemon + 0x35c) != -1) {
      local_10 = (char *)cache_get_name(new_chain);
      local_14 = strlen(local_10);
      local_18 = new_chain[9];
      b15fbc6811bc(*(undefined4 *)(dnsmasq_daemon + 0x35c),&local_14,4,0);
      b15fbc6811bc(*(undefined4 *)(dnsmasq_daemon + 0x35c),local_10,local_14,0);
      b15fbc6811bc(*(undefined4 *)(dnsmasq_daemon + 0x35c),new_chain + 7,4,0);
      b15fbc6811bc(*(undefined4 *)(dnsmasq_daemon + 0x35c),&local_18,4,0);
      if ((local_18 & 0x40005180) != 0) {
        b15fbc6811bc(*(undefined4 *)(dnsmasq_daemon + 0x35c),new_chain + 3,0x10,0);
      }
      if (((local_18 & 0x40000000) != 0) && ((local_18 & 0x20) == 0)) {
        a6f145b5c265(new_chain[3],*(undefined2 *)(new_chain + 4),
                     *(undefined4 *)(dnsmasq_daemon + 0x35c));
      }
    }
  }
  else {
    cache_free(new_chain);
  }
```

chall2

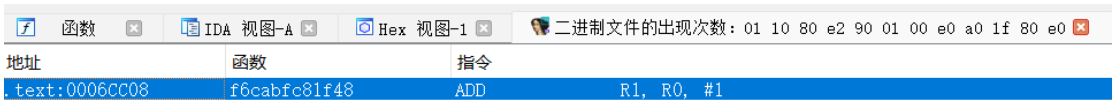
相同的地方也会调用相同的函数，函数的参数也是一致的。

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




```
bool e36366b920d8(int param_1,int param_2)

{
  uint uVar1;
  bool bVar2;

  uVar1 = (((param_1 * (param_1 + 1)) % 2) * 45 + 50) * param_1 +
          (((param_2 * (param_2 + 1)) % 2) * 95 + 106) * param_2;//一定是偶数
  if (uVar1 == (uVar1 - ((int)uVar1 >> 0x1f) & 0xfffffffe)) {//偶数
    bVar2 = (param_1 * param_1 * (param_1 + 1) * (param_1 + 1)) % 4 == 0;//true
  }
  else {//奇数
    bVar2 = (param_1 * param_1 * (param_1 + 1) * (param_1 + 1)) % 4 == 1;//false
  }
  return bVar2;
}
```

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

```
uint f6cabfc81f48(int param_1,int param_2)

{
  uint uVar1;
  int iVar2;
  uint local_28;
  uint local_24;

  uVar1 = (((param_1 * (param_1 + 1)) % 2) * 45 + 45) * 0x10000 +
          ((param_2 * (param_2 + 1)) % 2 + 1) * 95 + 0x5000b;//一定是偶数
  uVar1 = uVar1 + (uVar1 >> 0x10) * (param_1 + -1) * 0x10000;
  uVar1 = uVar1 + (param_2 + -1) * (uVar1 & 0xffff);
  if (((uVar1 >> 0x10) + (uVar1 & 0xffff) & 1) == 0) {
    iVar2 = param_1 * param_1 * (param_1 + 1) * (param_1 + 1);
    local_28 = (param_2 * param_2 * (param_2 + 1) * (param_2 + 1)) % 4;//local_28=0
  }
  else {
    iVar2 = param_1 * param_1 * (param_1 + 1) * (param_1 + 1);
    local_28 = (param_2 * param_2 * (param_2 + 1) * (param_2 + 1)) % 4 + 1;//local_28 = 1
  }
  local_24 = iVar2 % 4 + 1;//local_24=1
  return local_24 ^ local_28;
}
```

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

```
for (local_30 = 0; local_30 < 9; local_30 = local_30 + 1)
  aiStack_2c[local_30] = local_30 + 1;
}
```

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

```
for (local_30 = 0; local_30 < 3; local_30 = local_30 + 1) {
  for (local_34 = 0; local_34 < 3; local_34 = -(-1 - local_34)) {
    aiStack_2c[local_30 * 3 + local_34] = local_30 * 3 + local_34 + 1;
  }
}
```

即 a[i]=i(i=0,1,2…,8)

```
for (local_30 = 0; local_30 < 3; local_30 = local_30 + 1) {
  for (local_34 = 0; local_34 < 3; local_34++) {
    printf("%d ",aiStack_2c[local_30 * 3 + local_34]);
  }
}
printf("\n");
for (local_30 = 0; local_30 < 3; local_30 = local_30 + 1) {
  for (local_34 = local_30; local_34 < 3; local_34 = local_34 + 1) {
    iVar1 = aiStack_2c[local_30 * 3 + local_34];
    aiStack_2c[local_30 * 3 + local_34] = aiStack_2c[local_34 * 3 + local_30];
    aiStack_2c[local_34 * 3 + local_30] = iVar1;//13、26、57互换
  }
}
for (local_30 = 0; local_30 < 3; local_30 = local_30 + 1) {
  for (local_34 = 0; local_34 < 3; local_34 = local_34 + 1) {
    printf("%d ",aiStack_2c[local_30 * 3 + local_34]);
  }
}
printf("\n");
return 0;
```

```
}
for (local_30 = 0; local_30 < 9; local_30 = local_30 + 1) {
  printf("%d ",aiStack_2c[local_30]);
}
printf("\n");
for (local_30 = 0; local_30 < 9; local_30 = local_30 + 1) {
  iVar1 = __aeabi_idiv(local_30,3);
  iVar1 = (local_30 % 3) * 3 + iVar1;
  if (local_30 < iVar1) {
    iVar2 = aiStack_2c[local_30];
    aiStack_2c[local_30] = aiStack_2c[iVar1];
    aiStack_2c[iVar1] = iVar2;//a[iVar1]和a[local_30]互换
  }
}
for (local_30 = 0; local_30 < 9; local_30 = local_30 + 1) {
  printf("%d ",aiStack_2c[local_30]);
}
printf("\n");
return 0;
}
```

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

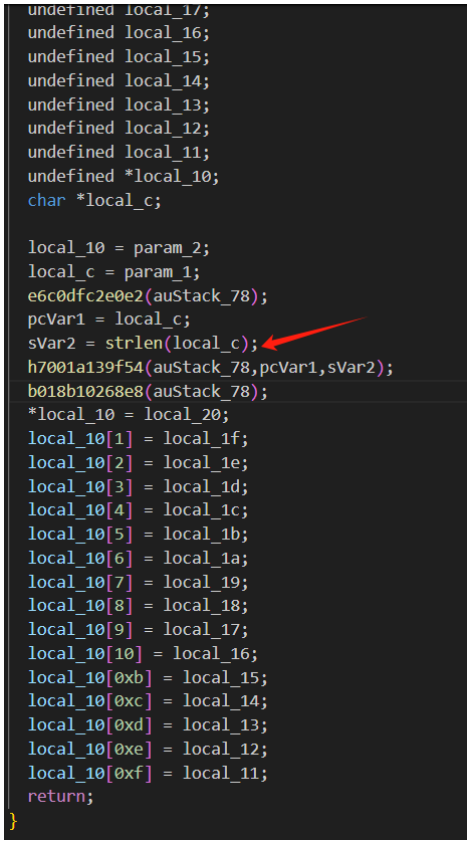
通过 malloc(0x21)来定位，能够先找到 jf2ee6ace808

对比第二个函数和第一个函数(下面是第二个)

```
void * j9cc83f200ae(char *param_1)

{
  void *pvVar1;
  int local_88;
  byte abStack_80 [16];
  size_t local_70;
  MD5_CTX MStack_6c;
  char *local_10;
  void *local_c;

  local_10 = param_1;
  MD5_Init(&MStack_6c);
  local_70 = strlen(local_10);
  MD5_Update(&MStack_6c,local_10,local_70);
  MD5_Final(abStack_80,&MStack_6c);
  pvVar1 = malloc(0x21);
  if (pvVar1 == (void *)0x0) {
    fprintf(stderr,"Error allocating memory\n");
    local_c = (void *)0x0;
  }
  else {
    for (local_88 = 0; local_88 < 0x10; local_88 = local_88 + 1) {
      sprintf((char *)((int)pvVar1 + local_88 * 2),"%02x",(uint)abStack_80[local_88
    }
    printf("%s\n",pvVar1);
    local_c = pvVar1;
  }
  return local_c;
}
```

```
void * jf2ee6ace808(undefined4 param_1)

{
  void *pvVar1;
  uint local_24;
  byte abStack_1c [16];
  undefined4 local_c;

  local_c = param_1;
  ca1394f2686c(param_1,abStack_1c);
  pvVar1 = malloc(0x21);
  for (local_24 = 0; local_24 < 0x10; local_24 = local_24 + 1) {
    sprintf((char *)((int)pvVar1 + local_24 * 2),"%02x",(uint)abStack_1c[local_24]);
  }
  printf("%s\n",pvVar1);
  return pvVar1;
}
```

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：

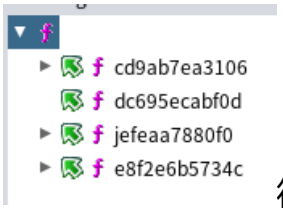


```
undefined local_17;
undefined local_16;
undefined local_15;
undefined local_14;
undefined local_13;
undefined local_12;
undefined local_11;
undefined *local_10;
char *local_c;

local_10 = param_2;
local_c = param_1;
e6c0dfc2e0e2(auStack_78);
pcVar1 = local_c;
sVar2 = strlen(local_c);
h7001a139f54(auStack_78,pcVar1,sVar2);
b018b10268e8(auStack_78);
*local_10 = local_20;
local_10[1] = local_1f;
local_10[2] = local_1e;
local_10[3] = local_1d;
local_10[4] = local_1c;
local_10[5] = local_1b;
local_10[6] = local_1a;
local_10[7] = local_19;
local_10[8] = local_18;
local_10[9] = local_17;
local_10[10] = local_16;
local_10[0xb] = local_15;
local_10[0xc] = local_14;
local_10[0xd] = local_13;
local_10[0xe] = local_12;
local_10[0xf] = local_11;
return;
}
```

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。

```
local_c = param_1;
sVar1 = strlen(param_1);
local_18 = 0xb8727a9;
while( true ) {
  while( true ) {
    while( true ) {
      while( true ) {
        while (local_18 == -0x472d1f29) {
          local_1c = (char *)malloc(local_28 + 1);
          strncpy(local_1c,param_1 + local_24,local_28);
          local_1c[local_28] = '\0';
          local_34 = local_1c;
          local_18 = 0x19a08222;
        }
        if (local_18 != -0x2cff52dc) break;
        local_18 = -0x12eac95d;
        local_28 = 1;
        local_24 = 0;
        local_20 = 0;
      }

      if (local_18 != -0x12eac95d) break;
      local_14 = -0x472d1f29;
      if (local_20 < (int)(sVar1 - 1)) {
        local_14 = 0x4214b3c6;
      }
      local_18 = local_14;
    }
    if (local_18 != 0xb8727a9) break;
    local_10 = -0x2cff52dc;
    if ((int)sVar1 < 2) {
      local_10 = 0x50bddddb;
    }
    local_18 = local_10;
  }
  if (local_18 == 0x19a08222) break;
```

```
  if (local_18 == 0x19a08222) break;
  if (local_18 == 0x2d117d85) {
    local_20 = local_20 + 1;
    local_18 = -0x12eac95d;
  }
  else if (local_18 == 0x4214b3c6) {
    bd534188b73e(param_1,local_20,local_20,&local_28,&local_24);
    bd534188b73e(param_1,local_20,local_20 + 1,&local_28,&local_24);
    local_18 = 0x2d117d85;
  }
  else {
    local_18 = 0x19a08222;
    local_34 = param_1;
  }
}
return local_34;
}
```

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：

```
local_10 = -0x2cff52dc;
if ((int)sVar1 < 2) {
  local_10 = 0x50bddddb;
}
local_18 = local_10;
```

。

如果 sVar1<2，那么 local_18 == 0x50bddddb, 然后直接执行

```
else {
  local_18 = 0x19a08222;
  local_34 = param_1;
}
```

。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

```
if (local_18 != -0x2cff52dc) break;
local_18 = -0x12eac95d;
local_28 = 1;
local_24 = 0;
local_20 = 0;
```

，local_18 = -0x12eac95d,接下来继续执行

```
if (local_18 != -0x12eac95d) break;
local_14 = -0x472d1f29;
if (local_20 < (int)(sVar1 - 1)) {
  local_14 = 0x4214b3c6;
}
local_18 = local_14;
```

会发现当 strlen(param_1)-1>0(local_20)的时候会执行

```
else if (local_18 == 0x4214b3c6) {
  bd534188b73e(param_1,local_20,local_20,&local_28,&local_24);
  bd534188b73e(param_1,local_20,local_20 + 1,&local_28,&local_24);
  local_18 = 0x2d117d85;
}
```

```
if (local_18 == 0x2d117d85) {
  local_20 = local_20 + 1;
  local_18 = -0x12eac95d;
}
```

，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

```
while (local_18 == -0x472d1f29) {
  local_1c = (char *)malloc(local_28 + 1);
  strncpy(local_1c,param_1 + local_24,local_28);
  local_1c[local_28] = '\0';
  local_34 = local_1c;
  local_18 = 0x19a08222;
}
```
。

这一段代码显然和 chall1 a387f1ff72e9 的下面部分代码相同。

```
__dest = (char *)malloc(local_18 + 1);
strncpy(__dest,local_10 + local_1c,local_18);
__dest[local_18] = '\0';
local_c = __dest;
```

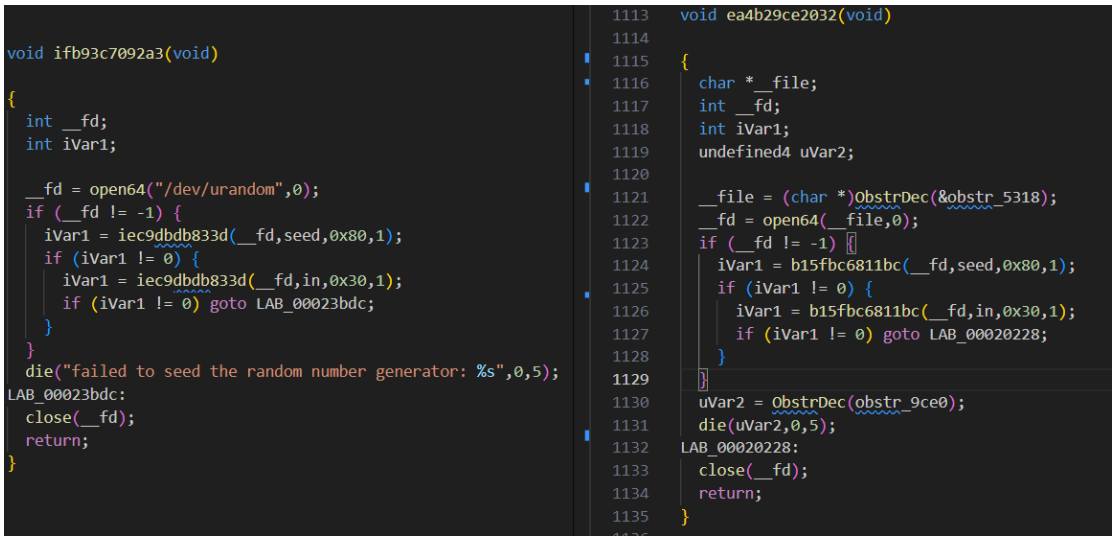
现在从整体上来看，我们可以发现 chall2 dc695ecabf0d 进行了控制流平坦化的混淆，if() break;相当于 switch：break。其实本质上还是在执行 chall1 函数里面的循环：

```
if ((int)local_14 < 2) {
  local_c = local_10;
}
else {
  local_18 = 1;
  local_1c = 0;
  for (local_20 = 0; local_20 < (int)(local_14 - 1); local_20 = local_20 + 1) {
    b368b9c1210f(local_10,local_20,local_20,&local_18,&local_1c);
    b368b9c1210f(local_10,local_20,local_20 + 1,&local_18,&local_1c);
  }
  __dest = (char *)malloc(local_18 + 1);
  strncpy(__dest,local_10 + local_1c,local_18);
  __dest[local_18] = '\0';
  local_c = __dest;
}
return local_c;
```

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



```
void ifb93c7092a3(void)

{
  int __fd;
  int iVar1;

  __fd = open64("/dev/urandom",0);
  if (__fd != -1) {
    iVar1 = iec9dbdb833d(__fd,seed,0x80,1);
    if (iVar1 != 0) {
      iVar1 = iec9dbdb833d(__fd,in,0x30,1);
      if (iVar1 != 0) goto LAB_00023bdc;
    }
  }
  die("failed to seed the random number generator: %s",0,5);
LAB_00023bdc:
  close(__fd);
  return;
}
```

```
1113  void ea4b29ce2032(void)
1114
1115  {
1116    char *__file;
1117    int __fd;
1118    int iVar1;
1119    undefined4 uVar2;
1120
1121    __file = (char *)ObstrDec(&obstr_5318);
1122    __fd = open64(__file,0);
1123    if (__fd != -1) {
1124      iVar1 = b15fbc6811bc(__fd,seed,0x80,1);
1125      if (iVar1 != 0) {
1126        iVar1 = b15fbc6811bc(__fd,in,0x30,1);
1127        if (iVar1 != 0) goto LAB_00020228;
1128      }
1129    }
1130    uVar2 = ObstrDec(obstr_9ce0);
1131    die(uVar2,0,5);
1132  LAB_00020228:
1133    close(__fd);
1134    return;
1135  }
```

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。**