

前言

根据一番定位，最终定位到解密函数在这个位置：

```
package com.fenbi.android.leo.imgsearch.sdk.utils;

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.util.zip.GZIPInputStream;
import kotlin.Metadata;
import org.jetbrains.annotations.NotNull;
import org.jetbrains.annotations.Nullable;

@Metadata(d1 = {"\u0000\u0010\n\u0002\u0018\u0002\n\u0002\u0010\u0000\u0000\u0000\u0002\u0010\u0012\n\u0002\b\u0006\b\u0000"}, loadedFrom = "classes2.dex")
public final class j {
    @NotNull
    /* renamed from: a reason: collision with root package name */
    public static final j f22712a = new j();

    @Nullable
    public final byte[] a(@Nullable byte[] bArr) throws IOException {
        boolean z11;
        if (bArr != null) {
            if (bArr.length == 0) {
                z11 = true;
            } else {
                z11 = false;
            }
        }
        if (!z11) {
            return kotlin.io.a.c(new GZIPInputStream(new ByteArrayInputStream(b(bArr))));
        }
        return null;
    }

    public final byte[] b(byte[] bArr) {
        return e.c(bArr);
    }
}
```

点入 c 方法，发现是 native 层的 c 方法，算法在 libContentEncoder.so 当中。

```
1 package com.fenbi.android.leo.imgsearch.sdk.utils;
2
3 /* loaded from: classes2.dex */
4 public class e {
5     static {
6         System.loadLibrary("ContentEncoder");
7     }
8
9     public static native byte[] c(byte[] bArr);
10 }
```

一、frida 的 hook

先用 frida 来 hook 一下这个方法的参数和返回值。可以看到结果为：[msm8996::小猿口算]-> e.c is called:

```
bArr=-53, -78, -87, -73, 41, -32, -92, 107, -96, 26, -  
25, 115, 102, -82, -119, -16, -114, -55, 54, -113, -19, 66, -  
21, 119, -15, -122, 36, 101, -76, 81, .....  
  
e.c result=31, -117, 8, 0, 0, 0, 0, 0, 0, -83, -43, 61, 111, -  
44, 48, 24, 7, -16, -17, -30, 57, -36, -39, 73, -100, -  
28, 34, 49, 84, 17, 18, 55, -16, 34, 122, -123, -127, 48, 24, -  
57, -71, 70, -92, 118, -22, 56, -108, 42, -118, ..... (太长了就不展示了, 自己 hook 一下就行)
```

可以看出参数和返回值的长度是一样的。这不禁令人想到了循环遍历等等操作.....

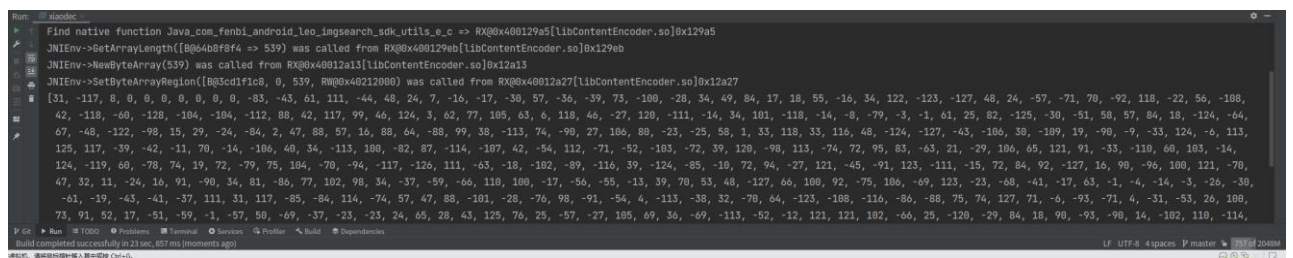
二、Unidbg 搭架子：

```

02. package com.course.xyks;
03.
04. import com.github.unidbg.AndroidEmulator;
05. import com.github.unidbg.Module;
06. import com.github.unidbg.debugger.Debugger;
07. import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
08. import com.github.unidbg.linux.android.AndroidResolver;
09. import com.github.unidbg.linux.android.dvm.*;
10. import com.github.unidbg.linux.android.dvm.array.ByteArray;
11. import com.github.unidbg.memory.Memory;
12.
13. import java.io.File;
14. import java.io.FileNotFoundException;
15. import java.util.Arrays;
16.
17. public class xiaodec extends AbstractJni {
18.     private final AndroidEmulator emulator;
19.     private final VM vm;
20.     private final Module module;
21.
22.     public DvmClass EClass;
23.     public String apkPath = "./apks/xiaoyuan/xiaoyuan.apk";
24.
25.     xiaodec() throws FileNotFoundException {
26.         emulator = AndroidEmulatorBuilder.for32Bit().build();
27.         final Memory memory = emulator.getMemory();
28.         memory.setLibraryResolver(new AndroidResolver(23));
29.         vm = emulator.createDalvikVM(new File(apkPath));
30.         vm.setVerbose(true);
31.         DalvikModule dm = vm.loadLibrary(new File("./apks/xiaoyuan/libContentEncoder.so"), true); // 加载so到虚拟内存
32.         vm.setJni(this);
33.         module = dm.getModule();
34.         dm.callJNI_OnLoad(emulator);
35.         EClass = vm.resolveClass("com/fenbi/android/leo/imgsearch/sdk/utlis/e");
36.     }
37.
38.     public void decrypt() {
39.         String methodId = "c([B][B";
40.         byte[] data = {-53,-78,-87,-73,41,-32,-92,107,-96,26,-25,115,102,-82,-119,-16,-114,-55,54,-113,-19,66,-21,119,-15,
41.         DvmObject<?> res = EClass.callStaticJniMethodObject(
42.             emulator, methodId,
43.             new ByteArray(vm,data)
44.         );
45.         ByteArray result = (ByteArray) res;
46.         String arr = Arrays.toString(result.getValue());
47.         System.out.println(arr);
48.     }
49.
50.
51.     public static void main(String[] args) throws FileNotFoundException {
52.         xiaodec obj = new xiaodec();
53.         obj.decrypt();
54.     }
55.
56.
57. }
58.

```

直接运行，可以发现不用补充任何环境，直接出结果，如下图所示。至此可以说明没有调用任何 java 的方法，是纯 so 算法实现，输出的结果和 hook 的一模一样。



三 算法还原

打开 IDA 进行反编译，可以看到 unidbg 帮我们定位到所有调用的 java 方法的地址：

```
[13:37:40 323] INFO [com.github.unidbg.linux.AndroidElfLoader] (AndroidElfLoader:471) - libContentEncoder.so load dependency libandroid.so failed
JNIEnv->FindClass(com/fenbi/android/leo/imgsearch/sdk/Utils/e) was called from RX@0x40012a91[libContentEncoder.so]0x12a91
JNIEnv->RegisterNatives(com/fenbi/android/leo/imgsearch/sdk/Utils/e, unidbg@0xbffff6e8, 1) was called from RX@0x40012aa3[libContentEncoder.so]0x12aa3
RegisterNative(com/fenbi/android/leo/imgsearch/sdk/Utils/e, c{[B, RX@0x400129a5[libContentEncoder.so]0x129a5)
Find native function Java_com_fenbi_android_leo_imgsearch_sdk_utils_e_c => RX@0x400129a5[libContentEncoder.so]0x129a5
JNIEnv->GetArrayLength([B@64b8f8f4 => 539) was called from RX@0x400129eb[libContentEncoder.so]0x129eb
JNIEnv->NewByteArray(539) was called from RX@0x40012a13[libContentEncoder.so]0x12a13
JNIEnv->SetByteArrayRegion([B@3cd1f1c8, 0, 539, RW@0x40212000) was called from RX@0x40012a27[libContentEncoder.so]0x12a27
```

我们直接跳转到 0x129a5 这个地址（我这里直接导入了 JNI 的头文件，所以直接显示出了 JNI 的 API 函数，第一个参数就是 JNI 指针 env，第二个参数是 jobject）

```
1 struct _jobject *__fastcall sub_129A4(JNIEnv_ *env, int a2, int a3)
2 {
3     struct _jobject *v5; // r10
4     const struct _JNIEnv *functions; // r0
5     jbyte *v7; // r8
6     jsize v8; // r6
7     jbyte *v9; // r4
8     void *v11; // [sp+4h] [bp-24h] BYREF
9     char v12; // [sp+8h] [bp-1Dh] BYREF
10
11     v5 = 0;
12     functions = env->functions;
13     v12 = 0;
14     v7 = functions->GetByteArrayElements(&env->functions, a3, &v12);
15     v11 = &off_275DC;
16     v8 = env->functions->GetArrayLength(env, a3);
17     v9 = operator new[](v8 | (v8 >> 31));
18     if ( sub_12C88(&v11, v7, v8, v9, v8) )
19     {
20         v5 = env->functions->NewByteArray(env, v8);
21         env->functions->SetByteArrayRegion(env, v5, 0, v8, v9);
22     }
23     operator delete[](v9);
24     env->functions->ReleaseByteArrayElements(env, a3, v7, 2);
25     return v5;
26 }
```

看到这我们可以发现，大部分只有 JNI 的内置函数，和加密的逻辑毫无关系，一眼看过去只有 sub_12C88 这个函数比较可疑，我们并不清楚内部干了什么事，所以我们先 hook 一下这个函数的参数和返回值，这样比较安心。在第 18 行按下 tab，来到汇编的界面：

0129EC	40 EA E0 70	ORR.W	R0, R0, R0, ASR#31	; unsigned int
0129F0	13 F0 36 E9	BLX	j__Znaj	; operator new[](uint)
0129F0				
0129F4	04 46	MOV	R4, R0	
0129F6	00 96	STR	R6, [SP, #0x28+var_28]	
0129F8	01 A8	ADD	R0, SP, #0x28+var_24	
0129FA	41 46	MOV	R1, R8	
0129FC	32 46	MOV	R2, R6	
0129FE	23 46	MOV	R3, R4	
012A00	00 F0 42 F9	BL	sub_12C88	
012A00				
012A04	78 B1	CBZ	R0, loc_12A26	
012A04				

这里我们在 0x12A00 这个位置下个断点，因为根据 ARM32 的调用约定，第一个参数是放在 R0 寄存器的，第二个参数放在 R1 寄存器，以此类推。为什么选择这个地址？因为这里存放参数的寄存器已经赋值完毕，下一步就是跳转函数进行调用了。

在类的构造函数末尾增加两行(32 位是 thumb 指令集，地址需要加 1)：

```
01. Debugger debugger = emulator.attach();
02. debugger.addBreakPoint(module.base + 0x12A00 + 1);
03.
```

断下之后可以看到 r1、r3、r4 都指向一块内存地址，那么分别看看他们指向的内存中的内容吧：

```
>>> r0=0xbffff6f4(-1073744140) r1=0x40191000 r2=0x21b r3=0x40212000 r4=0x40212000 r5=0xffffe12a0 r6=0x21b r7=0xbffff718 r8=0x40191000
>>> SP=0xbffff6f0 LR=RX@0x40115371[libc.so]0x4b371 PC=RX@0x40012a00[libContentEncoder.so]0x12a00 cpsr: N=0, Z=0, C=0, V=0, T=1, mode=
>>> d0=0xffffffffffffffff(NaN) d1=0x3220302034203720(3.002229861217884E-67) d2=0x3436333832203236(3.5366761868402984E-57) d3=0x312034
.2027122125173386E-153) d5=0x2030203020302030(1.2027122125173386E-153) d6=0x2030203020302030(1.2027122125173386E-153) d7=0x20302030
>>> d8=0x0(0.0) d9=0x0(0.0) d10=0x0(0.0) d11=0x0(0.0) d12=0x0(0.0) d13=0x0(0.0) d14=0x0(0.0) d15=0x0(0.0)
=> *[libContentEncoder.so@0x12a01]*[00f042f9]*0x40012a00:*"bl #0x40012c88"
[libContentEncoder.so 0x12a05] [78b1 ] 0x40012a04: "cbz r0, #0x40012a26"
[libContentEncoder.so 0x12a07] [2868 ] 0x40012a06: "ldr r0, [r5]"
[libContentEncoder.so 0x12a09] [3146 ] 0x40012a08: "mov r1, r6"
```

```
=cbb2a9b729e0a46ba01ae77366ae89f08ec9368fed42eb77f1862465b451b25e6fcd846bdc79febde8
bc75933895bc425
size: 112
0000: CB B2 A9 B7 29 E0 A4 6B A0 1A E7 73 66 AE 89 F0 ....).k...sf...
0010: 8E C9 36 8F ED 42 EB 77 F1 86 24 65 B4 51 B2 5E ..6..B.w..$.Q.^
0020: 6F CD 84 6B DC 79 FE BD E8 5C D2 54 F1 64 4B 71 o..k.y...\..T.dKq
0030: 85 C1 B7 59 76 FD 53 05 28 A3 A8 9E 22 5D 29 ED ...Yv.S(...").
0040: 1E 7D 44 D0 D5 93 10 EA 41 8B EE F1 FB DA 20 9F .}D.....A.....
0050: 75 00 AF 83 04 88 4E 6D A3 CD 66 F0 CA 68 E1 25 u....Nm..f..h.%
0060: 8E 5D C7 C0 62 BD A5 B6 CB C7 59 33 89 5B C4 25 ..b.....Y3.[.%
^-----^
```

最终可以看到只有 r1 指向的内存有值，而且经过我们的对比发现，这就是我们传入的字节数组。（可以自行百度一下 java 中字节数组怎么转换为 16 进制，最终对比是吻合的）

等函数执行完毕，也就是可以直接跳转到 0x12A04 这个地址，我们再来看看所有的寄存器的值：

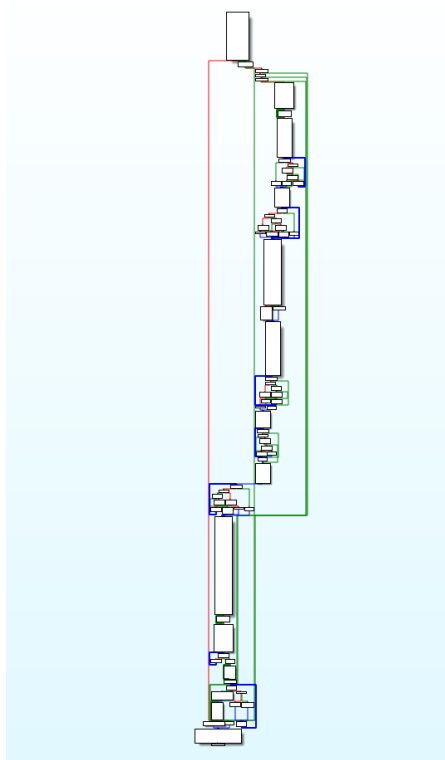
```
0x12A04
Add breakpoint: 0x40012a04 in libContentEncoder.so [0x12a04]
debugger break at: 0x40012a04 @ Function32 address=0x400129a5, arguments=[unidbg@0xffffe12a0, -670285875, 1013423070]
>>> r0=0x1 r1=0x0 r2=0x8 r3=0xd r4=0x40212000 r5=0xffffe12a0 r6=0x21b r7=0xbffff718 r8=0x40191000 sb=0x3c679bde sl=0x0 fp=0x402020a8
>>> SP=0xbffff6f0 LR=RX@0x40122617[libc.so]0x58617 PC=RX@0x40012a04[libContentEncoder.so]0x12a04 cpsr: N=0, Z=1, C=1, V=0, T=1, mode=0b10000
>>> d0=0xffffffffffffffff(NaN) d1=0x386b393164363861(6.400192910396156E-37) d2=0x3436333832203238(3.5366761868402995E-57) d3=0x3120323938343
.2027122125173386E-153) d5=0x2030203020302030(1.2027122125173386E-153) d6=0x2030203020302030(1.2027122125173386E-153) d7=0x2030203020302030
```

可以看到这里只有 r4 指向了内存地址，然后 mr4 看看它指向内存的内容：

```
↑
↓
=1f8b080000000000000000add53d6fd4301807f0efe239dcd9499ce4223154111237f0227a85813018
9105840a863268f
size: 112
0000: 1F 8B 08 00 00 00 00 00 00 00 AD D5 3D 6F D4 30 .....=0.0
0010: 18 07 F0 EF E2 39 DC D9 49 9C E4 22 31 54 11 12 .....9..I.."1T..
0020: 37 F0 22 7A 85 81 30 18 C7 B9 46 A4 76 EA 38 94 7."z..0...F.v.8.
0030: 2A 8A C4 80 98 98 90 58 2A 75 63 2E 7C 03 3E 4D *.X*uc.|.>M
0040: 69 3F 06 76 2E E5 78 91 F2 22 65 8A F2 F8 B1 FD i?.v..x.."e....
0050: FF 3D 19 52 83 E2 CD 3A 39 54 12 84 C0 43 D0 86 .=.R...:9T...C..
0060: 9E 0F 1D E8 AC 02 2F 58 39 10 58 40 A8 63 26 8F ...../X9.X@.c&.
^-----^
```

经过对比可以发现这就是我们返回值数组的 16 进制。那么这就意味着确实在这个函数当中进行了加密操作！传入的数组放在了第二个参数，然后函数执行完毕后再把加密结果存放在最后那个参数中。

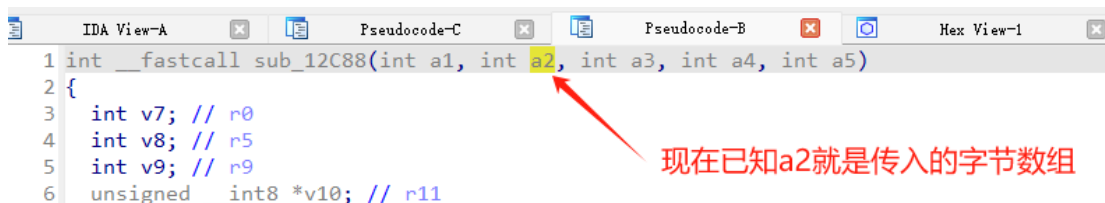
目标明确。当我们进到这个函数的时候就傻眼了，里面有无数个分支和跳转。看一眼流程图，这混淆的还挺严重。。。。



这时候一般有 2 种方法：要么是 trace 流程，把无用的分支手动去除，这种太费时间和精力。另外就是有点取巧的办法，而且半靠猜了，也就是 hook 我们认为比较关键的流程，这里明显采用的是后者。

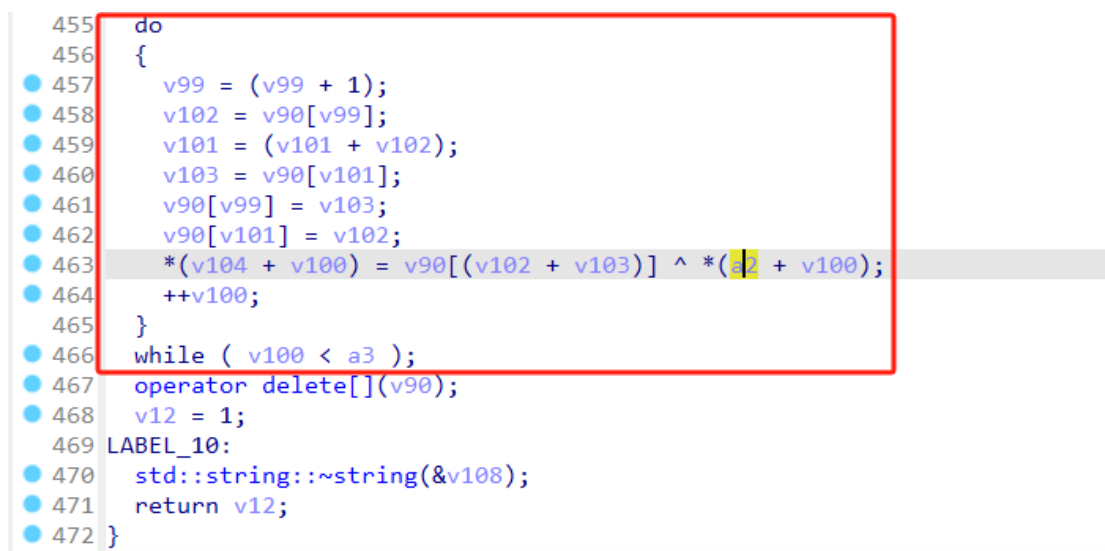
-----我是分隔符-----

事实上这里有一个十分关键的地方。之前我们说到第二个参数 **a2** 就是指向了我们传入的字节数组。



```
1 int __fastcall sub_12C88(int a1, int a2, int a3, int a4, int a5)
2 {
3     int v7; // r0
4     int v8; // r5
5     int v9; // r9
6     unsigned __int8 *v10; // r11
```

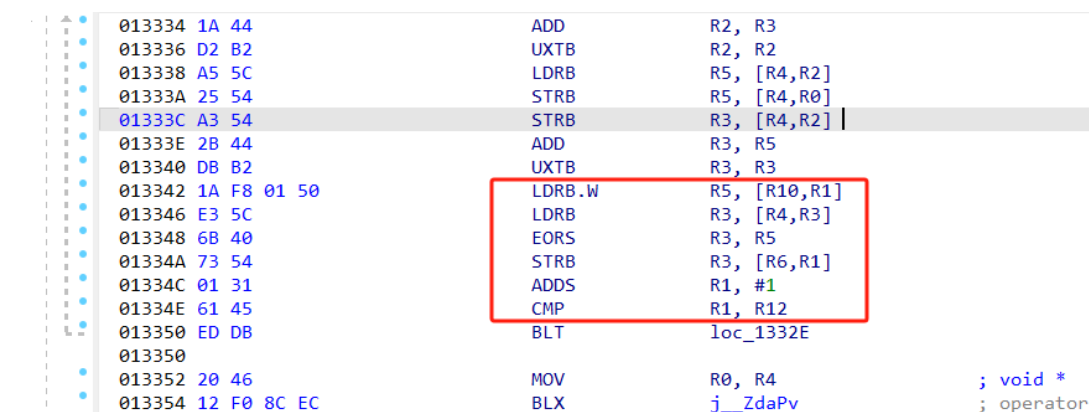
然后我们要找他对参与计算的地方，从上往下翻，发现只有末尾这个 **do-while** 循环利用到了 **a2** 指针。（一开始我还没注意到，因为就这一处，太不起眼了）



```
455 do
456 {
457     v99 = (v99 + 1);
458     v102 = v90[v99];
459     v101 = (v101 + v102);
460     v103 = v90[v101];
461     v90[v99] = v103;
462     v90[v101] = v102;
463     *(v104 + v100) = v90[(v102 + v103)] ^ *(a2 + v100);
464     ++v100;
465 }
466 while ( v100 < a3 );
467 operator delete[](v90);
468 v12 = 1;
469 LABEL_10:
470 std::string::~string(&v108);
471 return v12;
472 }
```

看到这里其实已经明白了吧？**这里就是加密运算的地方，解读一下这个循环吧：每次从 a2 指针指向的内存拿数据，并与 v90 指向的内存（也是个大数组）做异或操作，最后把结果存放在 v104 指向的内存中去，然后把数组指针的偏移量每次自增 1。**

如何验证我们的这个想法对不对呢？很简单，我们只需要查看一下 **v104** 指向的内存就可以。先来看看汇编，确定一下 **v104** 是哪个寄存器。



```
013334 1A 44      ADD     R2, R3
013336 D2 B2      UXTB    R2, R2
013338 A5 5C      LDRB    R5, [R4,R2]
01333A 25 54      STRB    R5, [R4,R0]
01333C A3 54      STRB    R3, [R4,R2]
01333E 2B 44      ADD     R3, R5
013340 DB B2      UXTB    R3, R3
013342 1A F8 01 50 LDRB.W   R5, [R10,R1]
013346 E3 5C      LDRB    R3, [R4,R3]
013348 6B 40      EORS    R3, R5
01334A 73 54      STRB    R3, [R6,R1]
01334C 01 31      ADDS    R1, #1
01334E 61 45      CMP     R1, R12
013350 ED DB      BLT     loc_1332E
013352 20 46      MOV     R0, R4
013354 12 F0 8C EC BLX     j__ZdaPv
```


那么我们直接在 0x1334C 下个断点看看，直接看 R6 指向的内存。

可以看到随着每次循环，v104 的确写入了加密后的字节数组（同样把 frida 返回的字节数组转 16 进制进行验证是正确的）。

现在事情就比较简单了，确定是最后的循环生成的结果。但是现在还有 v90 数组没有分析。我们向上看：

```

414 v106[25] = sub_12C38(*v88);
415 v89 = (v9 + 26);
416 if ( !v105 )
417     v89 = &v115 + 3;
418 v106[26] = sub_12C60(*v89);
419 v90 = operator new[](0x100u);
420 memset(v90, 0, 0x100u);
421 for ( i = 0; i != 256; ++i )
422     v90[i] = i;
423 v92 = 0;
424 LOBYTE(v93) = 0;
425 do
426 {
427     v94 = v90[v92];
428     v93 = (v93 + v94 + v106[v92 & 0x1F]);
429     v90[v92++] = v90[v93];
430     v90[v93] = v94;
431 }
432 while ( v92 != 256 );
433 v95 = 255;
434 v96 = v90[255];
435 while ( v95 >= 0 )
436 {
437     if ( v95 )
438     {
439         v97 = &v90[--v95];
440         v98 = v90[v95];
441     }
442     else
443     {
444         v95 = -1;
445         v97 = v90 + 255;
446         v98 = v96;
447     }
448     *v97 = v96;
449     v96 = v98;
450 }
451 operator delete[](v106);
452 LOBYTE(v99) = 0;
453 v100 = 0;

```

初始化v90指针，分配内存空间

对v90赋初值

对v90前32字节赋值

这里主要是给v97和v98赋值，下面没用到可以不看

可以看到 v90 是先初始化了一些值，我们可以先看下，在 0x132E0 下一个断点，直接读 R4 指向的内存：

0132DC 00 F0 1F 02	AND.W	R2, R0, #0x1F	
0132E0 23 5C	LDRB	R3, [R4,R0]	
0132E2 B2 5C	LDRB	R2, [R6,R2]	
0132E4 19 44	ADD	R1, R3	
0132E6 11 44	ADD	R1, R2	
0132E8 C9 B2	UXTB	R1, R1	
0132EA 62 5C	LDRB	R2, [R4,R1]	
0132EC 22 54	STRB	R2, [R4,R0]	
0132EE 01 30	ADDS	R0, #1	
0132F0 B0 F5 80 7F	CMP.W	R0, #0x100	
0132F4 63 54	STRB	R3, [R4,R1]	
0132F6 F1 D1	BNE	loc_132DC	
0132F6			
0132F8 20 46	MOV	R0, R4	
0132FA FF 22	MOVS	R2, #0xFF	
0132FC 10 F8 FF 1F	LDRB.W	R1, [R0,#0xFF]!	
0132FC			

427行对应的位置

R4就是v90

```

=000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f2021222324252627
8696a6b6c6d6e6f
size: 112
0000: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F .....
0010: 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
0020: 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F !"#$%&'()*+,-./
0030: 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 0123456789:;<=>?
0040: 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F @ABCDEFGHJKLMNO
0050: 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F PQRSTUVWXYZ[\]^_
0060: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F `abcdefghijklmno
^-----^

```

```

=606162636465666768696a6b6c6d6e6f707172737475767778797a7b7c7d7e7f80818283848586
8c9cacbcccdcecf
size: 112
0000: 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F `abcdefghijklmno
0010: 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F pqrstuvwxyz{|}~.
0020: 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F .....
0030: 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F .....
0040: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF .....
0050: B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF .....
0060: C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF .....
^-----^

```

可以看到，在 425 行的 do-while 循环之前，现在前 32 字节还是没有值的。后面几行就是初始化循环赋值的。

然后我们等循环结束再看看 v90 是什么。这里比较取巧的是，我们直接在最后那个加密循环里面看 v90 了，因为不管前面做了什么操作，最后加密运算的时候一定是最终版的 v90。

同样的思路，v90 生成的时候用到了 v106，而 v106 的生成和输入数组是无关的，因此是固定的数组可以不用管，我们直接读 v90 的值就行。可以看到 v106 是一个一个值赋值的，如果一个个去分析要累死，如下图所示。

```

396     v82 = &v114 + 3;
397     v106[22] = (*v82 | ((*v82 & 0x7Fu) + 127)) >> 7;
398     v83 = (v9 + 23);
399     if ( !v105 )
400         v83 = &v115;
401     v84 = *v83;
402     v85 = 0;
403     do
404         v86 = dword_114A0[v85++];
405         while ( v86 < v84 );
406         v106[23] = v85 - 2;
407         v87 = (v9 + 24);
408         if ( !v105 )
409             v87 = &v115 + 1;
410         v106[24] = sub_12C16(*v87);
411         v88 = (v9 + 25);
412         if ( !v105 )
413             v88 = &v115 + 2;
414         v106[25] = sub_12C38(*v88);
415         v89 = (v9 + 26);
416         if ( !v105 )
417             v89 = &v115 + 3;
418         v106[26] = sub_12C60(*v89);
419         v90 = operator new[](0x100u);
420         memset(v90, 0, 0x100u);
421         for ( i = 0; i != 256; ++i )
422             v90[i] = i;
423         v92 = 0;
424         LOBYTE(v93) = 0;
425         do
426         {
427             v94 = v90[v92];
428             v93 = (v93 + v94 + v106[v92 & 0x1F]);
429             v90[v92++] = v90[v93];
430             v90[v93] = v94;
431         }
432         while ( v92 != 256 );
433         v95 = 255;
434         v96 = v90[255];
435         while ( v95 >= 0 )
436         {

```

根据前面的分析，这三个间接寻址就分别代表了三个指针，那么 R10 是 a2，R6 是 v104，那么 R4 就是 v90 了，我们直接读一下 R4 指向的内存。

001332E	loc_1332E	
001332E 01 30	ADDS	R0, #1
0013330 C0 B2	UXTB	R0, R0
0013332 23 5C	LDRB	R3, [R4,R0]
0013334 1A 44	ADD	R2, R3
0013336 D2 B2	UXTB	R2, R2
0013338 A5 5C	LDRB	R5, [R4,R2]
001333A 25 54	STRB	R5, [R4,R0]
001333C A3 54	STRB	R3, [R4,R2]
001333E 2B 44	ADD	R3, R5
0013340 DB B2	UXTB	R3, R3
0013342 1A F8 01 50	LDRB.W	R5, [R10,R1]
0013346 E3 5C	LDRB	R3, [R4,R3]
0013348 6B 40	EORS	R3, R5
001334A 73 54	STRB	R3, [R6,R1]
001334C 01 31	ADDS	R1, #1
001334E 61 45	CMP	R1, R12
0013350 ED DB	BLT	loc_1332E

```
=16fe4486b69db7c0ce8f9b0f5e392f090d56dc439c4a1fa3cf8a0581666184a7135d8334cb46e832
35beb67c6b3a548
size: 112
0000: 16 FE 44 86 B6 9D B7 C0 CE 8F 9B 0F 5E 39 2F 09    ..D.....^9/.
0010: 0D 56 DC 43 9C 4A 1F A3 CF 8A 05 81 66 61 84 A7    .V.C.J.....fa..
0020: 13 5D 83 34 CB 46 E8 32 7E AC DF 26 7C 7F 85 28    .].4.F.2~..&|..(
0030: A9 B0 A1 DE 2D 0A 0E 41 51 27 59 87 C7 6F 91 1A    ....-..AQ'Y...o..
0040: 8C FC 17 2E 9F 64 6D 7A A0 53 69 29 FA B5 4F 79    .....dmz.Si)..0y
0050: B9 3A F9 57 BB 10 6C 19 71 9E 3F BF 89 D4 DA 5F    ...W..l.q.?...._
0060: 74 95 FF 37 2A D1 02 BA C3 5B EB 67 C6 B3 A5 48    t..7*....[.g...H
^-----^
```

根据初始化的内存，这个数组应该有 256 个字节。一张图显示不完。

```
Run: xiaodec x
^-----^
m0x40218060
>-----<
[15:20:23 884]RW@0x40218060, md5=815af06f26424f00bf00b941025be9a5,
hex
=7495ff372ad102bac35beb67c6b3a54893c294d778d21c97d507abb1ae8bd6d3145aa86ab86520
f183d24f160ada6
size: 112
0000: 74 95 FF 37 2A D1 02 BA C3 5B EB 67 C6 B3 A5 48    t..7*....[.g...H
0010: 93 C2 94 D7 78 D2 1C 97 D5 07 AB B1 AE 8B D6 D3    ....X.....
0020: 14 5A A8 6A B8 65 20 4E E2 B4 58 99 8E DD 35 7D    .Z.j.e N..X...5}
0030: A2 30 6E EE F0 E0 C4 54 BE CD EF C5 82 72 F3 E3    .0n....T....r..
0040: 0B B2 2C C9 3B 98 E1 F5 00 76 ED 4B E4 70 7B C1    ..,.;...v.K.p{.
0050: D9 75 CC 6B 1B 4D 1D 47 BD 04 12 92 88 08 45 EA    .u.k.M.G.....E.
0060: 80 E5 36 03 4C D8 D0 DB AF 18 3D 24 F1 60 AD A6    ..6.L.....=$.`..
^-----^
```

直接拿到 v90 的值即可还原整个算法，至此分析完毕！

总结：

我们直接通过 hook 拿到大数组 v90 赋值完毕后的值，体现了 hook 的强大。而加密操作就是利用这个大数组中的值对输入字节数组进行循环遍历的异或操作得到的。同时，由于异或是可逆操作，我们拿着密文数组和 v90 按同样逻辑异或一下，同样可以得到明文数组！（经验证，把输出的数组放到输入里面调用一下方法，我们得到的就是本来的输入数组）