Stuuuuub writeup

1. 题目设计

基本考察点：

- Android一代壳
- Android 10 InMemoryClassLoader实现Dex不落地加载
- 双亲委派机制
- 二代函数抽取壳(ART模式)
- inlineHook
- ollvm
- 字符串加密恢复
- APK签名校验
- JNIFunction

2. 解题

分析壳

JEB打开，发现dex已经经过混淆，找到StubApp类

在该类中的attachBaseContext方法中，首先通过e.a方法判断，之后调用e.d方法返回一个classloader v0

```
if(e.a(arg1)) {
    ClassLoader v0 = e.d();
    StubApp.classLoader = v0;
    if(v0 != null) {
        StubApp.flag = 1;
        return;
    }
}
```

在该类的onCreate方法中，直接使用classLoader加载类MainActivity，而App启动顺序是先调用attachBaseContext，再调用onCreate方法

```
 if(StubApp.flag == 1) {
    try {
        Class v4 =
StubApp.classLoader.loadClass("com.mrctf.android2022.MainActivity");
        this.startActivity(new Intent(this.getApplicationContext(), v4));
    }
    catch(ClassNotFoundException v3) {
        v3.printStackTrace();
    }

    return;
}
```

查看e.a实际上是进行了一系列架构、API以及ROOT权限的检测

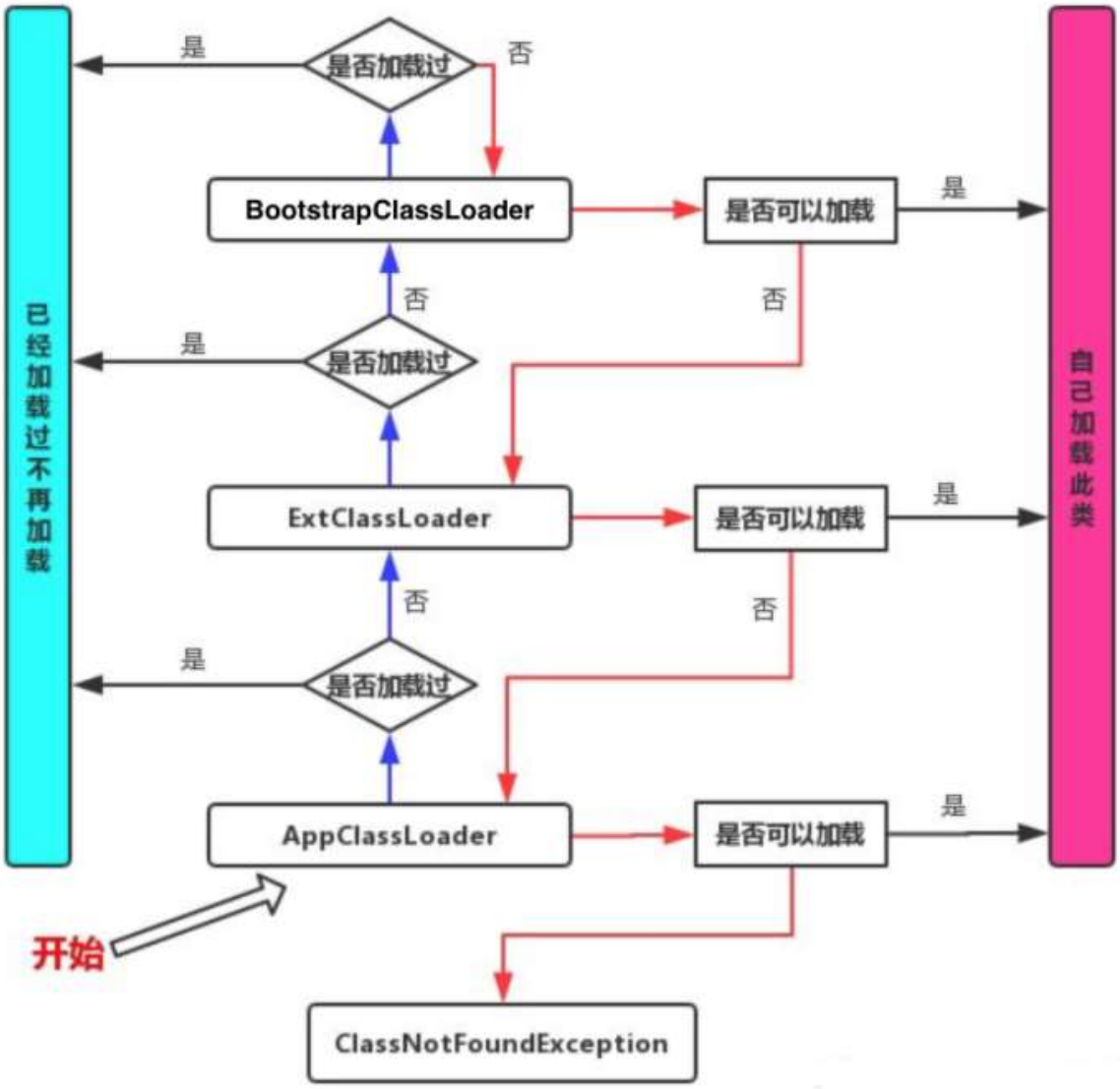
```
public class e {
    public static String a = null;
    public static String b = null;
    public static Context c = null;
    public static int d = -1;

    public static {
    }

    public static Context GetContext() {
        return e.c;
    }

    public static boolean a(Context arg1) {
        e.c = arg1;
        e.a = e.c.getApplicationInfo().dataDir + "/libnative.so";
        e.b = e.c.getApplicationInfo().dataDir + "/libc++_shared.so";
        if((Build.CPU_ABI.equals("armeabi-v7a")) && Build.VERSION.SDK_INT <= 30 && e.f() != 1) {
            return 1;
        }

        Log.e("Error", "Something wrong!");
        return 0;
    }

    public static boolean b() {
        String[] v0 = new String[]{"/sbin/su", "/system/bin/su", "/system/xbin/su", "/data/local/xbin/su", "/data/local/bin/su", "/system/sd/xbin/su", "/system/bin/failsafe/su", "/data/local/su", "/system/usr/we-nee
        int v2;
        for(v2 = 0; v2 < 10; ++v2) {
            String v3 = v0[v2];
            if(new File(v3).exists()) {
                Log.i("VS-RootUtils", "rooted file " + v3);
                return 1;
            }
        }

        return 0;
    }
```

在e.d中进行了so的解密释放和dex的解密以及使用InMemoryClassLoader装载的过程
其中libnative.so释放时使用decodeSo进行了解密，而libc++_shared.so并未进行解密直接释放，dex直接异或49后加载

```
public static ClassLoader d() {
    try {
        System.loadLibrary("stub");
        InputStream soInputStream = e.context.getAssets().open("res.dat");
        int soLength = soInputStream.available();
        byte[] soByte = new byte[soLength];
        soInputStream.read(soByte);
        BufferedOutputStream v3 = new BufferedOutputStream(new FileOutputStream(e.soPath));
        v3.write(soByte, 0, soLength);
        soInputStream.close();
        v3.close();
        e.decodeSo(soByte, soLength, e.soPath, e.soPath.length());
        InputStream v0_2 = e.context.getAssets().open("libc++_shared.so");
        int sharedsoLength = v0_2.available();
        byte[] sharedsoByte = new byte[sharedsoLength];
        v0_2.read(sharedsoByte);
        BufferedOutputStream bufferOutputStream = new BufferedOutputStream(new FileOutputStream(e.b));
        bufferOutputStream.write(sharedsoByte, 0, sharedsoLength);
        v0_2.close();
        bufferOutputStream.close();
        InputStream dexInputStream = e.context.getAssets().open("build.json");
        int dexLength = dexInputStream.available();
        byte[] dexByte = new byte[dexLength];
        dexInputStream.read(dexByte);
        int v3_2;
        for(v3_2 = 0; v3_2 < dexLength; ++v3_2) {
            dexByte[v3_2] = (byte)(dexByte[v3_2] ^ 49);
        }

        dexInputStream.close();
        InMemoryDexClassLoader v1_3 = new InMemoryDexClassLoader(new ByteBuffer[]{ByteBuffer.wrap(dexByte)}, e.context.getApplicationInfo().dataDir, e.context.getClassLoader());
        e.g(e.context, v1_3);
        return v1_3;
    }
    catch(IOException v0) {
        v0.printStackTrace();
        return null;
    }
}
```

在e.d最后调用的e.g实际上是通过替换classloader来使得加载的Activity能被找到并且正常调用，因为ClassLoader加载的是一个Activity而不仅仅是一个类。这里涉及到Android的双亲委派机制，如下图所示，为考虑安全性，当类进行加载时首先向其父类的classloader进行查询这个类是否被加载过，如果加载过就不再加载，如果没加载过则再向上一级查，知道最上层，如果依旧没查到就向下逐级看看能否加载类，直到最底层。

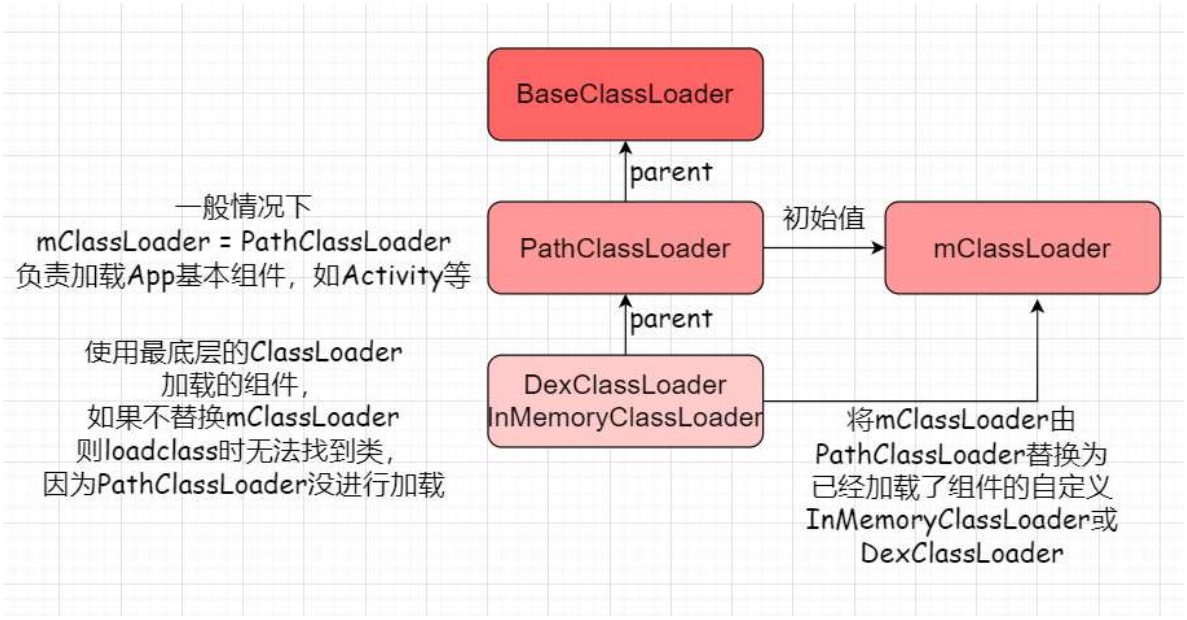
这样做的好处：避免了类重复被加载，其次就是避免系统类被替换从而造成的安全问题。



这里使用InMemoryClassLoader进行加载，向上找肯定没有找到其他的loader已经加载，那么最终还是会交给InMemoryClassLoader进行加载。问题就在这里，因为加载的是Activity，而负责Activity加载的ClassLoader是在程序启动时就已经被赋值的，即图中的mClassLoader（如下图所示），这个mClassLoader其实是PathClassLoader，PathClassLoader是DexClassLoader的父级，BootClassLoader又是PathClassLoader的父级。

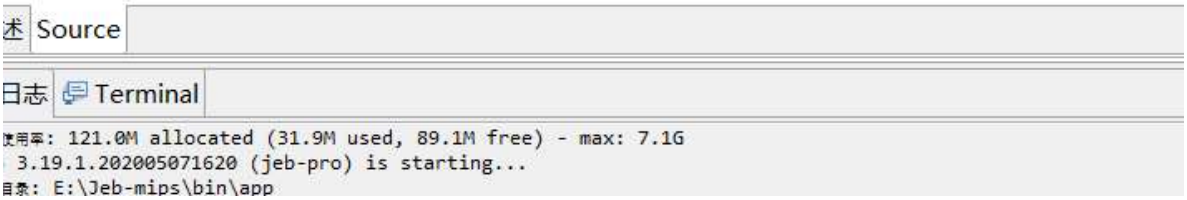
```java
public final class LoadedApk {

    private static final String TAG = "LoadedApk";

    private final ActivityThread mActivityThread;
    final String mPackageName;
    private ApplicationInfo mApplicationInfo;
    private String mAppDir;
    private String mResDir;
    private String[] mSplitAppDirs;
    private String[] mSplitResDirs;
    private String[] mOverlayDirs;
    private String[] mSharedLibraries;
    private String mDataDir;
    private String mLibDir;
    private File mDataDirFile;
    private File mDeviceProtectedDataDirFile;
    private File mCredentialProtectedDataDirFile;
    private final ClassLoader mBaseClassLoader;
    private final boolean mSecurityViolation;
    private final boolean mIncludeCode;
    private final boolean mRegisterPackage;
    private final DisplayAdjustments mDisplayAdjustments = new DisplayAdjustments();
    /** WARNING: This may change. Don't hold external references to it. */
    Resources mResources;
    private ClassLoader mClassLoader;
    private Application mApplication;

    private final ArrayMap<Context, ArrayMap<BroadcastReceiver, ReceiverDispatcher>> mReceivers
        = new ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>>();
    private final ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>> mUnregister
        = new ArrayMap<Context, ArrayMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>>();
    private final ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>> mServices
        = new ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>>();
    private final ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>> mUnboundSer
        = new ArrayMap<Context, ArrayMap<ServiceConnection, LoadedApk.ServiceDispatcher>>();

    int mClientCount = 0;
```

每次加载组件时通过mClassLoader即PathClassLoader来加载，但是因为App在启动时这个类没有被加载进入PathClassLoader，所以后面就无法通过PathClassLoader找到此类，就需要替换mClassLoader为加载自定义类的ClassLoader。



通过反射获取到android.app.LoadedApk类中的mClassLoader字段并进行修改

```
    public static void g(Context arg4, ClassLoader arg5) {
        try {
            Class v0 = arg5.loadClass("android.app.ActivityThread");
            Object v1 = v0.getDeclaredMethod("currentActivityThread").invoke(null);
            v0.getDeclaredField("mPackages").setAccessible(true);
            Object v4_4 = ((WeakReference)((ArrayMap)v0.getDeclaredField("mPackages").get(v1)).get(arg
            arg5.loadClass("android.app.LoadedApk").getDeclaredField("mClassLoader").setAccessible(tru
            arg5.loadClass("android.app.LoadedApk").getDeclaredField("mClassLoader").set(v4_4, arg5);
        }
        catch(ClassNotFoundException | NoSuchMethodException v4_3) {
            ((ReflectiveOperationException)v4_3).printStackTrace();
        }
        catch(InvocationTargetException v4_2) {
            v4_2.printStackTrace();
        }
        catch(IllegalAccessException v4_1) {
            v4_1.printStackTrace();
        }
        catch(NoSuchFieldException v4) {
            v4.printStackTrace();
        }
    }
}
```



到此壳就分析完毕

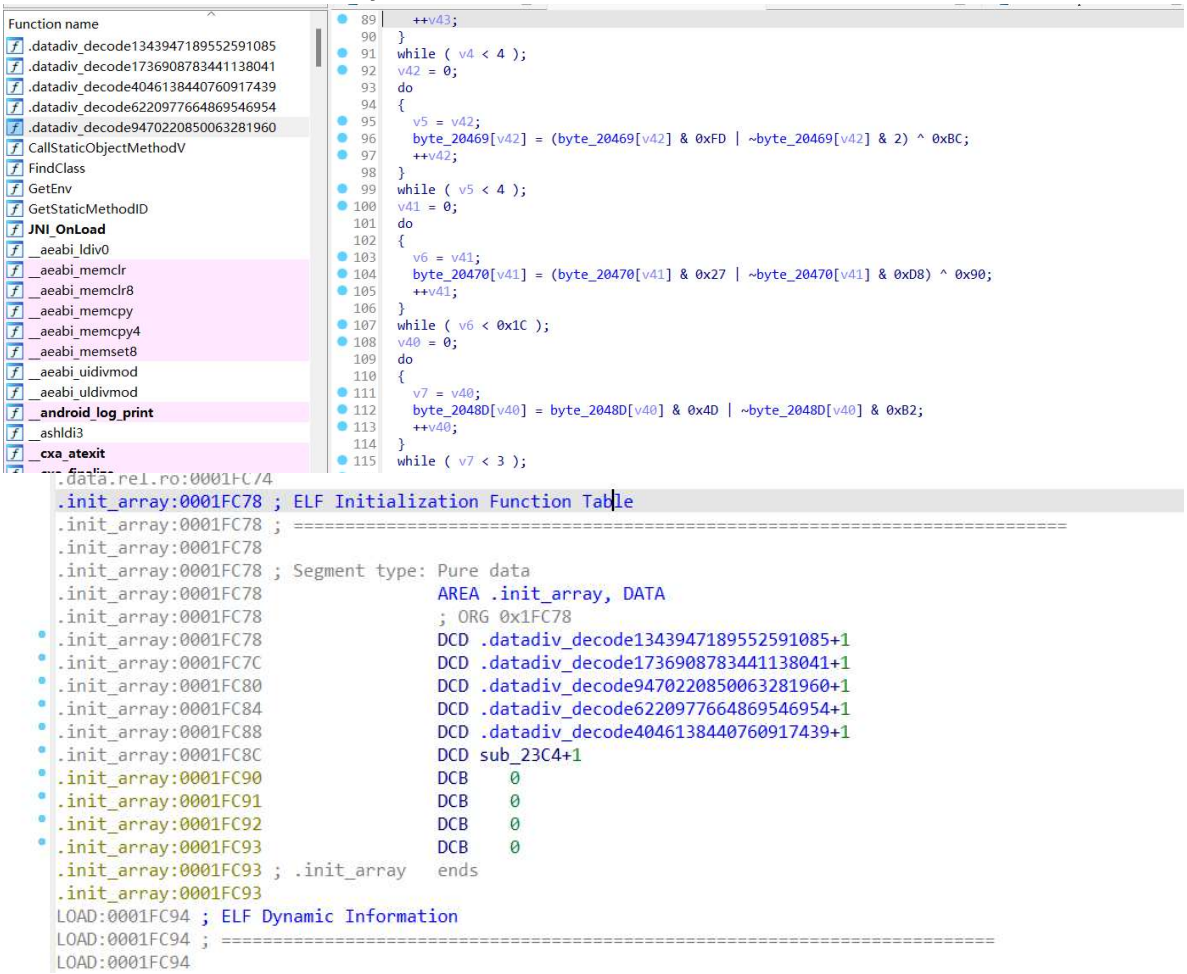
获取加密后的Dex和so

首先需要解密dex和so，dex很容易解密，直接拿出assets中的build.json逐字节异或49即可
so解密需要分析libstub.so查看decodeSo的实现
发现并没有直接将decodeSo导出，应该是使用动态注册的方式

大量字符串被加密

```
.data:000201B2                 DCB 0xFC
.data:000201B3                 DCB 0xE9
.data:000201B4                 DCB 0xFA
.data:000201B5                 DCB 0x9F
.data:000201B6 unk_201B6       DCB 0x3E ; >            ; DATA XREF: sub_2DEC+27E↑o
.data:000201B6                                         ; sub_2DEC+282↑o ...
.data:000201B7                 DCB 0x3B ; ;
.data:000201B8                 DCB 0x30 ; 0
.data:000201B9                 DCB 0x33 ; 3
.data:000201BA                 DCB 0x20
.data:000201BB                 DCB 0x26 ; &
.data:000201BC                 DCB 0x7C ; |
.data:000201BD                 DCB 0x21 ; !
.data:000201BE                 DCB 0x3D ; =
.data:000201BF                 DCB 0x52 ; R
.data:000201C0 unk_201C0       DCB 0x36 ; 6            ; DATA XREF: JNI_OnLoad+138↑o
.data:000201C0                                         ; .datadiv_decode1343947189552591085+180↑o ...
.data:000201C1                 DCB 0x37 ; 7
.data:000201C2                 DCB 0x31 ; 1
.data:000201C3                 DCB 0x3D ; =
.data:000201C4                 DCB 0x36 ; 6
.data:000201C5                 DCB 0x37 ; 7
.data:000201C6                 DCB    1
.data:000201C7                 DCB 0x3D ; =
.data:000201C8                 DCB 0x52 ; R
.data:000201C9                 DCB    0
.data:000201CA                 DCB    0
.data:000201CB                 DCB    0
.data:000201CC                 DCB    0
.data:000201CD                 DCB    0
.data:000201CE                 DCB    0
```

而这些解密函数都是在.init_array段，也就是会最开始执行



此处有多种解法：

- 一种是直接启动APP在内存里找对应的so dump解密后的数据
- 一种是使用unicorn模拟so的执行，使其完成初始化后自动修补so

这里提供第二种做法的脚本,使用AndroidNativeEmu开源框架，或者使用其加强版ExAndroidNativeEmu

```
import logging
import sys

from unicorn import *
import struct
from androidemu.emulator import Emulator

# Configure logging
logging.basicConfig(
    stream=sys.stdout,
    level=logging.DEBUG,
    format="%(asctime)s %(levelname)7s %(name)34s | %(message)s"
)


dstr_datas = {}


def hook_mem_write(uc, type, address, size, value, userdata):
    try:
        curdata = struct.pack("I", value)[:size]
```

```
            dstr_datas[address] = curdata
        except:
            print(size)
        #print(curdata)


logger = logging.getLogger(__name__)

emulator = Emulator(vfp_inst_set=True, vfs_root="vfs")

# 设置内存的写入监控
emulator.mu.hook_add(UC_HOOK_MEM_WRITE, hook_mem_write)


# 后面的do_init为true就会调用.init_array
lib_module = emulator.load_library("libstub.so", do_init=True)

#emulator.call_symbol(lib_module, "JNI_OnLoad", emulator.java_vm.address_ptr,
0x0)

base_addr = lib_module.base

sofile = open("libstub.so", "rb")

# 我们要将真实的字符串回填到sodata中。然后再保存
sodata = sofile.read()
for address, v in dstr_datas.items():
    # 仅仅将so范围内的保存原字符串进行写回
    if address > base_addr and address < base_addr+lib_module.size:
        offset = address-base_addr-0x1000
        print("address:0x%x data:%s offset:0x%x" % (address, v, offset))
        sodata = sodata[:offset]+v+sodata[offset+len(v):]

# 保存成一个新的so
savepath = "./libstub_new.so"
nfile = open(savepath, "wb")
nfile.write(sodata)
nfile.close()
```

解密效果

```
.data:000201A7                                         ; sub_2A2C+8↑o ...
.data:000201AF aExecve         DCB "execve",0          ; DATA XREF: sub_2A2C+14↑o
.data:000201AF                                         ; sub_2A2C+16↑o ...
.data:000201B6 aLibartSo       DCB "libart.so",0       ; DATA XREF: sub_2DEC+27E↑o
.data:000201B6                                         ; sub_2DEC+282↑o ...
.data:000201C0 aDecodeso       DCB "decodeSo",0        ; DATA XREF: JNI_OnLoad+138↑o
.data:000201C0                                         ; .datadiv_decode1343947189552591085+180↑o
.data:000201C9                 ALIGN 0x10
.data:000201D0 aBiljavaLangStr DCB "([BILjava/lang/String;I)V",0
.data:000201D0                                         ; DATA XREF: JNI_OnLoad+13A↑o
.data:000201D0                                         ; .datadiv_decode1343947189552591085+1AA↑o
.data:000201EA                 ALIGN 0x10
.data:000201F0 aAndroidxFakeSt DCB "androidx/fake/stub/e",0
.data:000201F0                                         ; DATA XREF: JNI_OnLoad+146↑o
.data:000201F0                                         ; JNI_OnLoad+148↑o ...
.data:00020205 aGetcontext     DCB "GetContext",0      ; DATA XREF: sub_4268+1E↑o
.data:00020205                                         ; sub_4268+20↑o ...
.data:00020210 aLandroidConten DCB "()Landroid/content/Context;",0
.data:00020210                                         ; DATA XREF: sub_4268+22↑o
.data:00020210                                         ; sub_4268+26↑o ...
.data:0002022C aGetpackagename DCB "getPackageName",0  ; DATA XREF: sub_42B8+2A↑o
.data:0002022C                                         ; sub_42B8+2E↑o ...
.data:0002023B                 ALIGN 0x10
.data:00020240 aLjavaLangStrin DCB "()Ljava/lang/String;",0
.data:00020240                                         ; DATA XREF: sub_42B8+30↑o
.data:00020240                                         ; sub_42B8+34↑o ...
.data:00020255                 ALIGN 0x10
.data:00020260 aComMrctfAndroi DCB "com.mrctf.android2022",0
.data:00020260                                         ; DATA XREF: sub_42B8+56↑o
.data:00020260                                         ; sub_42B8+5A↑o ...
.data:00020276 aStub           DCB "Stub",0            ; DATA XREF: sub_42B8:loc_4426↑o
```

分析JNI_OnLoad

能够根据JNI_OnLoad的固定结构对符号进行部分恢复

```
48    }
49    if ( v11 == -408696684 )
50      break;
51    switch ( v11 )
52    {
53      case 237161578:
54        v12 = GetContext((int)env);
55        v5 = CheckSign(env, v12);
56        v6 = -800858156;
57        if ( v5 )
58          v6 = 2086914350;
59        v11 = v6;
60        break;
61      case 1636971859:
62        v13 = -1;
63        v11 = -408696684;
64        break;
65      case 1705224739:
66        dword_206C0 = (int)env;
67        v16[2] = (int)sub_3A74;
68        v16[1] = (int)"([BILjava/lang/String;I)V";
69        v16[0] = (int)"decodeSo";
70        v3 = RegisterNatives(env, "androidx/fake/stub/e", v16, 1);
71        v4 = -920099282;
72        if ( v3 )
73          v4 = 237161578;
74        v11 = v4;
75        break;
76      case 2076069969:
77        v13 = -1;
78        v11 = -408696684;
79        break;
80      case 2086914350:
81        v7 = sub_47D4();
82        v8 = 2107649377;
83        if ( hookexecve(v7) == -1 )
```

获取到APP的签名进行异或后与密文比较进行校验

```
54  while ( v13 == -2041373925 )
55  {
56    v12 = GetMethodID(a1, ObjectClass, "getPackageManager", "()Landroid/content/pm/PackageManager;");
57    v35 = CallObjectMethodV(a1, a2, v12);
58    v33 = GetObjectClass(a1, v35);
59    v31 = GetMethodID(a1, v33, "getPackageInfo", "(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;");
60    v29 = CallObjectMethodV(a1, v35, v31, v37, 64);
61    v27 = GetObjectClass(a1, v29);
62    FieldID = GetFieldID(a1, v27, "signatures", "[Landroid/content/pm/Signature;");
63    ObjectField = GetObjectField(a1, v29, FieldID);
64    ObjectArrayElement = GetObjectArrayElement(a1, ObjectField, 0);
65    v19 = GetObjectClass(a1, ObjectArrayElement);
66    v40 = GetMethodID(a1, v19, "toCharsString", "()Ljava/lang/String;");
67    v17 = CallObjectMethodV(a1, ObjectArrayElement, v40);
68    sign_global = GetStringUTFChars(a1, v17, 0);
69    i = 0;
70    v13 = 1289137879;
71  }
72  if ( v13 == -1298733280 )
73    break;
74  switch ( v13 )
75  {
76    case -878417483:
77      v13 = -569594979;
78      _android_log_print(3, "Stub", "Wrong package name");
79      break;
80    case -659868594:
81      ++i;
82      v13 = 325758440;
83      break;
84    case -569594979:
85      v3 = -2041373925;
86      v4 = -2041373925;
87      if ( y_56 < 10 )
88        v4 = 1289137879;
89      if ( ((x_55 * (x_55 - 1)) & 1) == 0 )
90        v3 = v4;
91      if ( (((x_55 * (x_55 - 1)) & 1) == 0) != y_56 < 10 )
92        v3 = 1289137879;
93      v13 = v3;
94      break;
95    case -320225562:
96      v13 = 325758440;
97      break;
98    case -63649095:
99      v10 = 840974552;
100     if ( (enc[i] & ~(i % 43) | (i % 43) & ~enc[i]) != sign_global[i] )// enc[i] ^ (i%43) == sign[i]
101       v10 = 1797566148;
```

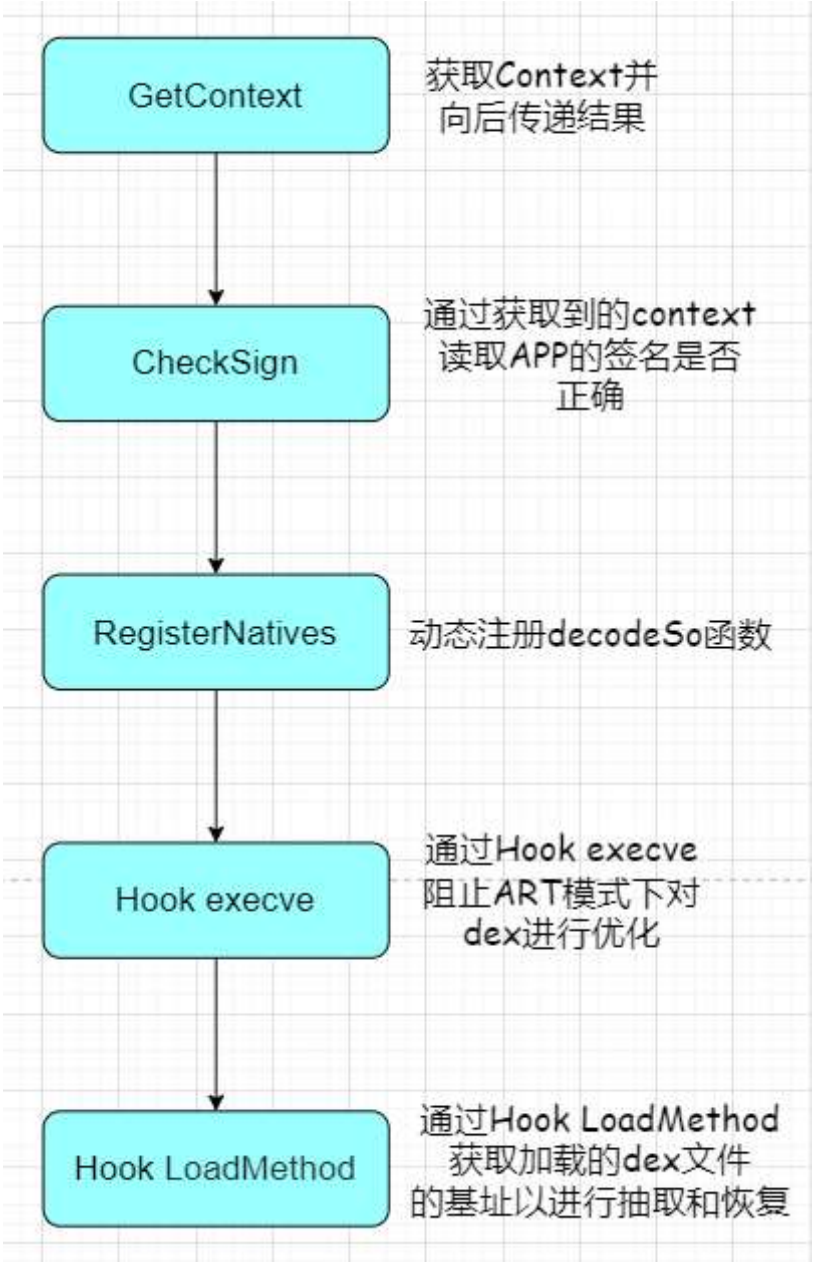
动态注册decodeSo到该函数，发现其实是是将内容异或0x22后释放到/data/data/com.mrctf.android2022/目录下

```
int __fastcall decodeSo(JNIEnv *a1, int a2, int a3, int a4, int a5)
{
  int result; // r0
  int v6; // r1
  int v7; // r12
  int v8; // r1
  int v9; // r2
  int v10; // r12
  int v11; // r1
  int v12; // r1
  int v13; // r12
  int v14; // r1
  int v15; // r12
  int v16; // r1
  int v17; // r12
  int v18; // r1
  int v19; // r12
  int v20; // r1
  int v21; // [sp+28h] [bp-48h]
  FILE *stream; // [sp+2Ch] [bp-44h]
  char *filename; // [sp+30h] [bp-40h]
  char *bytestream; // [sp+34h] [bp-3Ch]
  int v25; // [sp+3Ch] [bp-34h]
  bool v28; // [sp+5Fh] [bp-11h]

  v25 = 0;
  bytestream = GetByteArrayElements(a1, a3, 0);
  filename = GetStringUTFChars(a1, a5, 0);
  stream = fopen(filename, "wb");
  v21 = -262400580;
  while ( 1 )
  {
    while ( v21 == -2094469408 )
    {
      fclose(stream);
      v15 = -209530435;
```

```
91      case -209530435:
92        v21 = -2094469408;
93        fclose(stream);
94        break;
95      case -60608151:
96        v21 = 1210251633;
97        fputc((bytestream[v25] & 0x9C | ~bytestream[v25] & 0x4B9F7763) ^ 0x4B9F7741, stream);
98        break;
99      case 914927916:
100       v25 = 0;
```

到此为止即可直接恢复 dex 和 libnative.so

但是实际上JNI_OnLoad还没完全分析完，后面会有两个Hook,先把整个JNI_OnLoad流程图放在这



- 第一个Hook是用于hook libc.so 导出函数exexve，因为在Android5之后将dalvik虚拟机逐渐改为ART虚拟机用于提示性能，一般在安装APK时就会进行dex2oat的优化，将dex字节码转为直接可以在cpu上执行的机器码，如果是动态加载的dex则无法在安装时进行此操作，但是一旦通过LoadClass加载，在加载过程中会调用相关函数对其进行oat转换，虽然不一定会转换也不一定成功，但是如果一旦转换，对内存中的dex字节码进行修改就会无效，因为系统会执行优化后的机器码，这个时候抽取壳便无效了。
  之所以hook该函数是因为，在dex2oat的过程中最终会通过该函数执行命令进行转换，只要识别到命令中有dex2oat命令就返回null表示失败，其他命令照常执行。（该过程详见参考资料）
- 第二个Hook是Hook libart.so中的LoadMethod函数，该函数是加载流程中走的最底层的函数，能够获取到ArtMethod结构体指针和方法地址偏移，在题目中获取的是其第二个参数==const DexFile& dex_file==,能通过这个地址指向的结构获取内存中的dex的起始位置和size，只需要修改地址空间保护属性对指定方法指令段偏移进行修改即可进行抽取
- 以下是位加载类时执行流程
  `ClassLoader.java::loadClass -> DexPathList.java::findClass -> DexFile.java::defineClass -> class_linker.cc::LoadClass ->`

`class_linker.cc::LoadClassMembers -> class_linker.cc::LoadMethod`

```
void ClassLinker::LoadMethod(Thread* self,
                             const DexFile& dex_file,
                             const ClassDataItemIterator& it,
                             Handle<mirror::Class> klass,
                             ArtMethod* dst) {
  uint32_t dex_method_idx = it.GetMemberIndex();
  const DexFile::MethodId& method_id = dex_file.GetMethodId(dex_method_idx);
  const char* method_name = dex_file.StringDataByIdx(method_id.name_idx_);

  ScopedAssertNoThreadSuspension ants(self, "LoadMethod");
  dst->SetDexMethodIndex(dex_method_idx);
  dst->SetDeclaringClass(klass.Get());
  dst->SetCodeItemOffset(it.GetMethodCodeItemOffset());

  dst->SetDexCacheResolvedMethods(klass->GetDexCache()->GetResolvedMethods(), image_pointer_size_);
  dst->SetDexCacheResolvedTypes(klass->GetDexCache()->GetResolvedTypes(), image_pointer_size_);

  uint32_t access_flags = it.GetMethodAccessFlags();

  if (UNLIKELY(strcmp("finalize", method_name) == 0)) {
    // Set finalizable flag on declaring class.
    if (strcmp("V", dex_file.GetShorty(method_id.proto_idx_)) == 0) {
      // Void return type.
      if (klass->GetClassLoader() != nullptr) {  // All non-boot finalizer methods are flagged.
        klass->SetFinalizable();
      } else {
        std::string temp;
        const char* klass_descriptor = klass->GetDescriptor(&temp);
        // The Enum class declares a "final" finalize() method to prevent subclasses from
        // introducing a finalizer. We don't want to set the finalizable flag for Enum or its
        // subclasses, so we exclude it here.
        // We also want to avoid setting the flag on Object, where we know that finalize() is
```

- 这里不细分析这部分的具体实现了，有兴趣可查阅参考资料和开源项目

分析解密后的libnative. so 和 dex

首先是dex文件，包含两个类，一个MainActivity，另一个Utils
在MainActivity的onCreate开始实例化了一个Utils类，之后便监听按钮截取输入传入nativeCheck

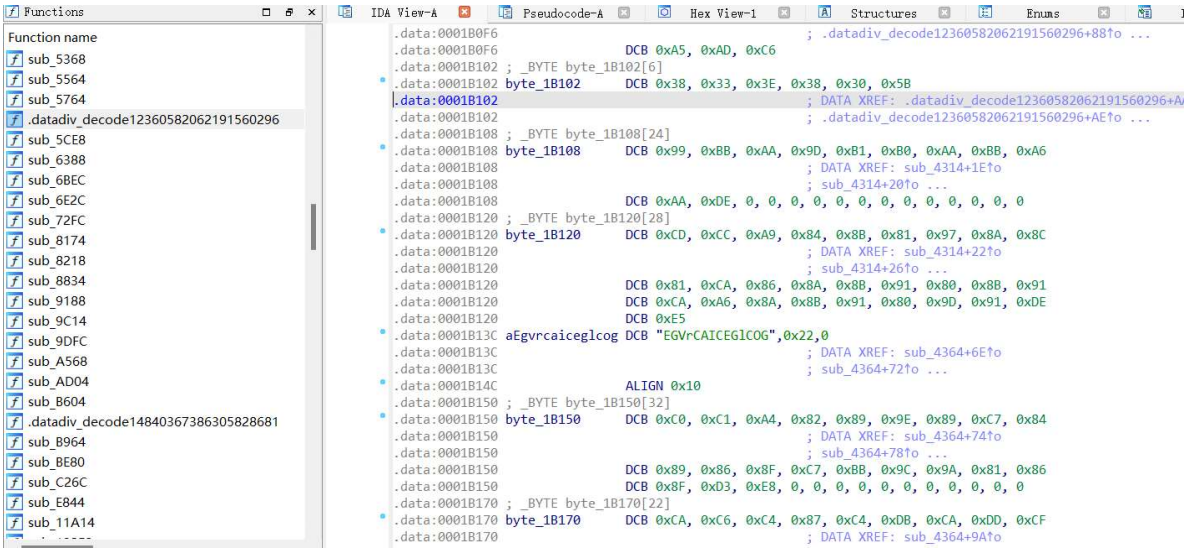
实例化的时候先load了释放完成的libnative. so，随即进行了删除

```
@SuppressLint({"UnsafeDynamicallyLoadedCode"})
public Utils(Context context) {
    Utils.globalContext = context;
    String path = context.getApplicationInfo().dataDir + "/libnative.so";
    System.load(path);
    File sofile = new File(path);
    if(sofile.exists()) {
        sofile.delete();
    }
}
```

查看nativeCheck，也是native层的函数

```
    }

    public static native boolean nativeCheck(String arg0) {
    }

    public static boolean test(String input) {
        return 1;
    }
}
```

分析libnative.so,依旧是字符串经过了加密，用上面的任意一种方法进行解密



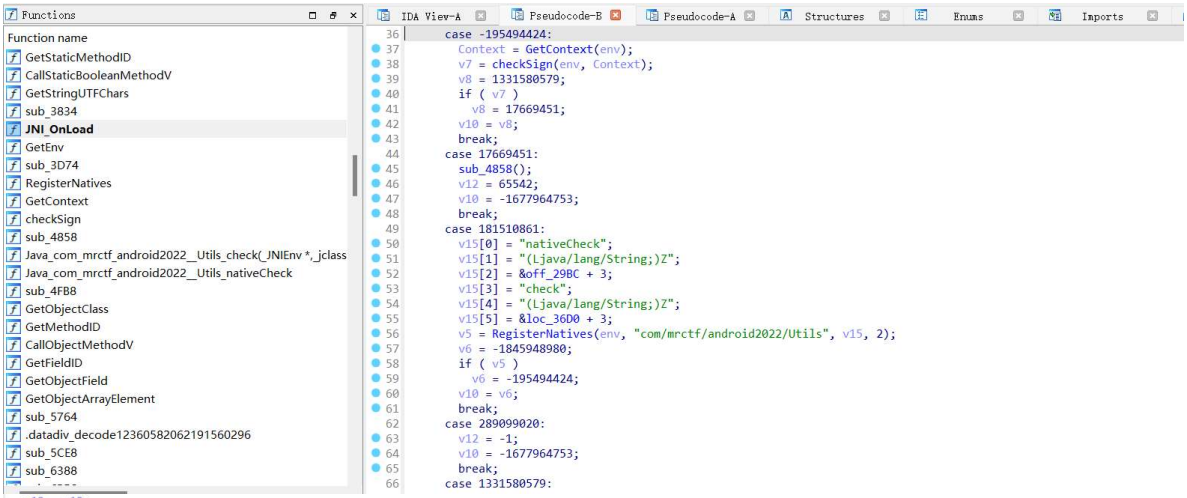
这次发现好像有nativeCheck的导出函数，但是实际是假的，函数名_Utils前多了一个下划线

函数逻辑是如果调用Java层 test返回结果为1则返回调用Java_com_mrctf_android2022__Utils_check的返回值

实际上Java_com_mrctf_android2022__Utils_check提示是错误的flag

```
15  char *s; // [sp+28h] [bp-50h]
16  int v17; // [sp+2Ch] [bp-4Ch]
17  bool v18; // [sp+3Eh] [bp-3Ah]
18  bool v19; // [sp+3Fh] [bp-39h]
19  char v20[44]; // [sp+40h] [bp-38h] BYREF
20
21  strcpy(v20, "MRCTF{great_job_and_this_is_a_right_flag??}");
22  s = GetStringUTFChars(a1, a3, 0);
23  strlen(s);
24  v17 = 0;
25  v15 = 74720615;
26  while ( v15 != -2018027952 )
27  {
28    switch ( v15 )
29    {
30      case -1318674142:
31        v15 = 74720615;
32        break;
33      case -1249945568:
34        v9 = 581860768;
35        if ( v19 )
36          v9 = 1437298230;
37        v15 = v9;
38        break;
39      case -147889686:
40        v6 = 0;
41        v19 = s[v17] != v20[v17];
42        if ( y_58 < 10 )
43          v6 = 1;
```

JNI_OnLoad查看，依旧是签名校验后注册函数，但是这次注册函数是两个且无法定位到native地址



在注册上面有一个函数，发现其读取了数据目录的一个shm文件并且将值保存最后调用java层的fs方法将其删除

```
      v8 = 1331580579;
      if ( v7 )
        v8 = 17669451;
      v10 = v8;
      break;
    case 17669451:
      get_shm();
      v12 = 65542;
      v10 = -1677964753;
      break;
    case 181510861:
      v15[0] = "nativeCheck";
      v15[1] = "(Ljava/lang/String;)Z";
      v15[2] = &off_29BC + 3;
      v15[3] = "check";
```

```
s = malloc(0x64u);
memset(s, 0, 0x64u);
ObjectClass = GetObjectClass(dword_1B378, dword_1B374);
MethodID = GetMethodID(dword_1B378, ObjectClass, "getApplicationInfo", "()Landroid/content/pm/ApplicationInfo;");
v12 = CallObjectMethodV(dword_1B378, dword_1B374, MethodID);
v5 = dword_1B378;
v4 = GetObjectClass(dword_1B378, v12);
FieldID = GetFieldID(v5, v4, "dataDir", "Ljava/lang/String;");
ObjectField = GetObjectField(dword_1B378, v12, FieldID);
StringUTFChars = GetStringUTFChars(dword_1B378, ObjectField, 0);
strcpy(s, StringUTFChars);
v0 = strlen(s);
strcpy(&s[v0], "/shm");
stream = fopen(s, "rb");
for ( i = -2094208916; ; i = -2042973776 )
{
  while ( i == -2094208916 )
  {
    v2 = -2042973776;
    if ( stream )
      v2 = -674417935;
    i = v2;
  }
  result = -2042973776;
  if ( i == -2042973776 )
    break;
  fscanf(stream, "%p", &dword_1B388);
  fclose(stream);
  v8 = FindClass(dword_1B378, "androidx/fake/stub/e");
  v7 = GetStaticMethodID(dword_1B378, v8, "fs", "()V");
```

在JNI_OnLoad末尾有一个函数，目前还不知道是什么功能


```
16  int v14; // [sp+10h] [bp-10h]
17  int v15; // [sp+14h] [bp-Ch]
18
19  v14 = sub_5CE8(*(dword_1B37C + 860), sub_3834, &unk_1B380);
20  v12 = 1066206631;
21  while ( 1 )
22  {
23    while ( 1 )
24    {
25      while ( 1 )
26      {
27        while ( v12 == -1666601460 )
28          v12 = 1640299289;
29        if ( v12 != -1019040711 )
30          break;
31        v1 = -259441916;
32        if ( sub_8834(*(dword_1B37C + 860)) )
33          v1 = 945189506;
34        v12 = v1;
35      }
36      if ( v12 != -1010397476 )
37        break;
38      v13 = 0;
39      v12 = -262007843;
40    }
41    if ( v12 == -278674195 )
42      break;
43    switch ( v12 )
44    {
45      case -262007843:
```

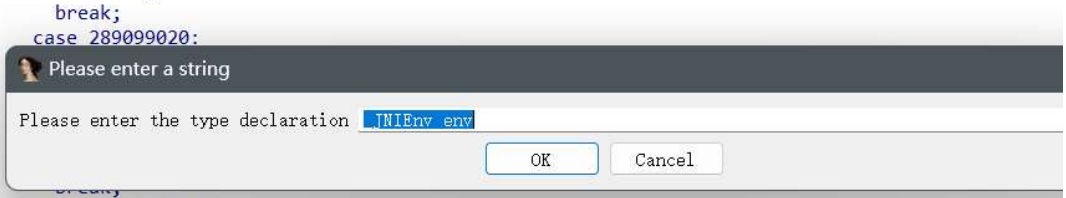
分析dword_1837C,发现仅有一个写，是保存了env的地址，

Directio	Ty	Address	Text
Up	o	sub_3834+2A	LDR R0, =(dword_1B37C - 0x3864)
Up	o	sub_3834+2C	ADD R0, PC; dword_1B37C
Up	r	sub_3834+2E	LDR R0, [R0]
Up	o	sub_3834:loc_38F6	LDR R0, =(dword_1B37C - 0x38FC)
Up	o	sub_3834+C4	ADD R0, PC; dword_1B37C
Up	r	sub_3834+C6	LDR R0, [R0]
Up	o	.text:off_3930	DCD dword_1B37C - 0x3864
Up	o	.text:off_3934	DCD dword_1B37C - 0x38FC
Up	o	JNI_OnLoad+126	LDR R1, =(dword_1B37C - 0x3A64)
Up	o	JNI_OnLoad+128	ADD R1, PC; dword_1B37C
Up	w	JNI_OnLoad+12A	STR R0, [R1]
Up	o	.text:off_3B70	DCD dword_1B37C - 0x3A64
Up	o	sub_3D74+6	LDR R0, =(dword_1B37C - 0x3D80)
Up	o	sub_3D74+8	ADD R0, PC; dword_1B37C
Up	r	sub_3D74+A	LDR R0, [R0]
Do...	o	sub_3D74:loc_3E9C	LDR R0, =(dword_1B37C - 0x3EA2)
Do...	o	sub_3D74+12A	ADD R0, PC; dword_1B37C
Do...	r	sub_3D74+12C	LDR R0, [R0]
Do...	o	.text:off_406C	DCD dword_1B37C - 0x3D80
Do...	o	.text:off_4098	DCD dword_1B37C - 0x3EA2
Do...	o	RegisterNatives:loc_41F8	LDR R0, =(dword_1B37C - 0x41FE)

```
54        v15[4] = "(Ljava/lang/String;)Z";
55        v15[5] = &loc_36D0 + 3;
56        v5 = RegisterNatives(env, "com/mrctf/android2022/Utils", v15, 2);
57        v6 = -1845948980;
58        if ( v5 )
59          v6 = -195494424;
60        v10 = v6;
61        break;
62      case 289099020:
63        v12 = -1;
64        v10 = -1677964753;
65        break;
66      case 1331580579:
67        v10 = 17669451;
68        break;
69      case 1555978094:
70        v12 = -1;
71        v10 = -1677964753;
72        break;
73      default:
74        dword_1B37C = *env;
75        dword_1B378 = env;
76        v3 = sub_3D74();
77        v4 = 1555978094;
78        if ( v3 )
79          v4 = 181510861;
80        v10 = v4;
81        break;
82    }
```

将jni.h导入IDA，重新定义该结构为_JNIEnv 类型，发现其通过env的function指针获取到了RegisterNatives的地址

```
56        v5 = RegisterNatives(env, "com/mrctf/android2022/Utils", v15, 2);
57        v6 = -1845948980;
58        if ( v5 )
59          v6 = -195494424;
60        v10 = v6;
61        break;
62      case 289099020:
63
64
65
66
67
68
69      case 1555978094:
70        v12 = -1;
71        v10 = -1677964753;
72        break;
73      default:
74        ::env.functions = *env;
75        dword_1B378 = env;
76        v3 = sub_3D74();
77        v4 = 1555978094;
78        if ( v3 )
79          v4 = 181510861;
80        v10 = v4;
81        break;
82    }
83  }
84  return v12;
```




```
11  int v8; // r1
12  int v9; // r12
13  int v10; // r1
14  int v12; // [sp+8h] [bp-18h]
15  int v13; // [sp+Ch] [bp-14h]
16  int v14; // [sp+10h] [bp-10h]
17  int v15; // [sp+14h] [bp-Ch]
18
19  v14 = sub_5CE8(env.functions->RegisterNatives, sub_3834, &unk_1B380);
20  v12 = 1066206631;
21  while ( 1 )
22  {
23    while ( 1 )
24    {
25      while ( 1 )
26      {
27        while ( v12 == -1666601460 )
28          v12 = 1640299289;
29        if ( v12 != -1019040711 )
30          break;
31        v1 = -259441916;
32        if ( sub_8834(env.functions->RegisterNatives) )
33          v1 = 945189506;
34        v12 = v1;
35      }
36      if ( v12 != -1010397476 )
37        break;
38      v13 = 0;
39      v12 = -262007843;
40    }
41    if ( v12 == -278674195 )
42      break;
43    switch ( v12 )
```

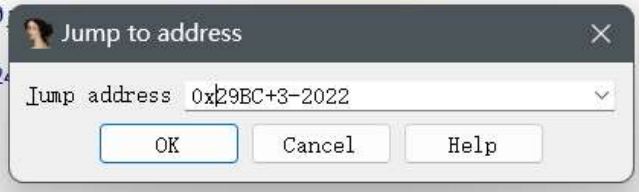
查看sub_5834函数，其实很清晰了，就是将a3，也就是JNINativeMethod结构体指针指向结构体的fnptr - 2022，而该结构体是用于指明动态注册函数的各种信息的，第三个即为注册到的函数地址，那sub_5CE8就可能是Hook函数，Hook到RegisterNatives后对其参数进行修改-2022，之后调用

```
IDA View-A  Pseudocode-B  Pseudocode-A  Structures  Enums  Imports
1  jint __fastcall sub_3834(JNIEnv *a1, void *a2, const JNINativeMethod *a3, jint a4)
2  {
3    int v4; // r1
4    int i; // [sp+1Ch] [bp-24h]
5    jint v11; // [sp+30h] [bp-10h]
6    int v12; // [sp+34h] [bp-Ch]
7
8    a3->fnPtr = a3->fnPtr - 2022;
9    a3[1].fnPtr = a3[1].fnPtr - 2022;
10   v12 = sub_6E2C(env.functions->RegisterNatives);
11   for ( i = -1464945114; ; i = -193677112 )
12   {
13     while ( 1 )
14     {
15       while ( i == -1594123718 )
16       {
17         v11 = env.functions->RegisterNatives(a1, a2, a3, a4);
18         i = -193677112;
19       }
20       if ( i != -1464945114 )
21         break;
22       v4 = -1594123718;
23       if ( v12 )
24         v4 = 1295222973;
25       i = v4;
26     }
27     if ( i == -193677112 )
28       break;
29     v11 = -1;
30   }
31   return v11;
32 }
```

```
125      JNIGlobalRefType = 2,
126      JNIWeakGlobalRefType = 3
127  } jobjectRefType;
128
129  typedef struct {
130      const char* name;
131      const char* signature;
132      void*       fnPtr;
133  } JNINativeMethod;
134
135  struct _JNIEnv;
```

对其进行验证，跳转到对应的位置，果然发现逻辑，猜想正确

```
50        v15[0] = "nativeCheck";
51        v15[1] = "(Ljava/lang/String;)Z";
52        v15[2] = &off_29BC + 3;
53        v15[3] = "check";
54        v15[4] = "(Ljava/lang/String;)Z";
55        v15[5] = &loc_36D0 + 3;
56        v5 = RegisterNatives(env, "com/mrctf/android2022/Utils", v15, 2);
57        v6 = -1845948980:
      v10 = -1677964753;
      break;
    case 181510861:
      v15[0] = "nativeCheck";
      v15[1] = "(Ljava/lang/String;)Z";
      v15[2] = &off_29BC + 3;
      v15[3] = "check";
      v15[4] = "(Ljava/lang/String;)Z";
      v15[5] = &loc_36D0 + 3;
      v5 = RegisterNatives(env, "com/mrctf/android2022/Utils", v15, 2);
      v6 = -1845948980;
      if ( v5 )
        v6 = -19549442
      v10 = v6;
      break;
    case 289099020:
      v12 = -1;
      v10 = -1677964753;
      break;
    case 1331580579:
```




```
96                                    v24 = 0;
97                                    *(shm + v37) = 0;
98                                    if ( y < 10 )
99                                      v24 = 1;
100                                   v25 = -1233480869;
101                                   if ( (((x * (x - 1)) & 1) == 0) != v24 )
102                                     v25 = -951427033;
103                                   v26 = v25;
104                                   if ( y < 10 )
105                                     v26 = -951427033;
106                                   if ( ((x * (x - 1)) & 1) == 0 )
107                                     v25 = v26;
108                                   v32 = v25;
109                                 }
110                                 if ( v32 != -1968515834 )
111                                   break;
112                                 Class = FindClass(a1, "com/mrctf/android2022/Utils");
113                                 StaticMethodID = GetStaticMethodID(
114                                                    a1,
115                                                    Class,
116                                                    "test",
117                                                    "(Ljava/lang/String;)Z");
118                                 v33 = CallStaticBooleanMethodV(
119                                         a1,
120                                         Class,
121                                         StaticMethodID,
122                                         a3,
123                                         "(Ljava/lang/String;)Z");
124                                 v32 = 411906854;
125                               }
126                               if ( v32 != -1841686497 )
127                                 break;
128                               v20 = 1991675850;
129                               if ( v42 )
130                                 v20 = -897853344;
000027B2 sub_21D8:95 (27B2)
```

该nativeCheck函数调用了Java层的test函数并返回了test的返回值，这个时候就有意思了，test明显返回的是1，这里究竟有什么玄机，其实就是抽取壳的实现，之前我们发现一个shm文件到目前为止都没有用到，但是在该函数里出线了shm

- 共三处使用，第一处是赋值操作

```
            break;
          *(shm + v37) = byte_1B010[v37];
          v32 = 1381819862;
        }
        if ( v32 != -45173303 )
          break;
        v27 = 219049051;
        if ( (((x * (x - 1)) & 1) == 0) != y < 10 )
          v27 = 1767627602;
        v28 = v27;
        if ( y < 10 )
          v28 = 1767627602;
        if ( ((x * (x - 1)) & 1) == 0 )
          v27 = v28;
        v32 = v27;
      }
      if ( v32 != 193692003 )
        break;
      v3 = 324740790;
      if ( v40 )
        v3 = 1321911560;
      v32 = v3;
    }
    if ( v32 != 219049051 )
      break;
    ++v37;
    v32 = 1767627602;
  }
  if ( v32 != 324740790 )
    break;
  v14 = -1968515834;
  if ( (((x * (x - 1)) & 1) == 0) != y < 10 )
    v14 = 411906854;
  v15 = v14;
  if ( y < 10 )
    v15 = 411906854;
  if ( ((x * (x - 1)) & 1) == 0 )
    v14 = v15;
  v32 = v14;
}
if ( v32 != 411906854 )
  break;
v35 = FindClass(a1, "com/mrctf/android2022/Utils");
v16 = GetStaticMethodID(a1, v35, "test", "(Ljava/lang/String;)Z");
ret = CallStaticBooleanMethodV(a1, v35, v16, a3);
v17 = shm;
```

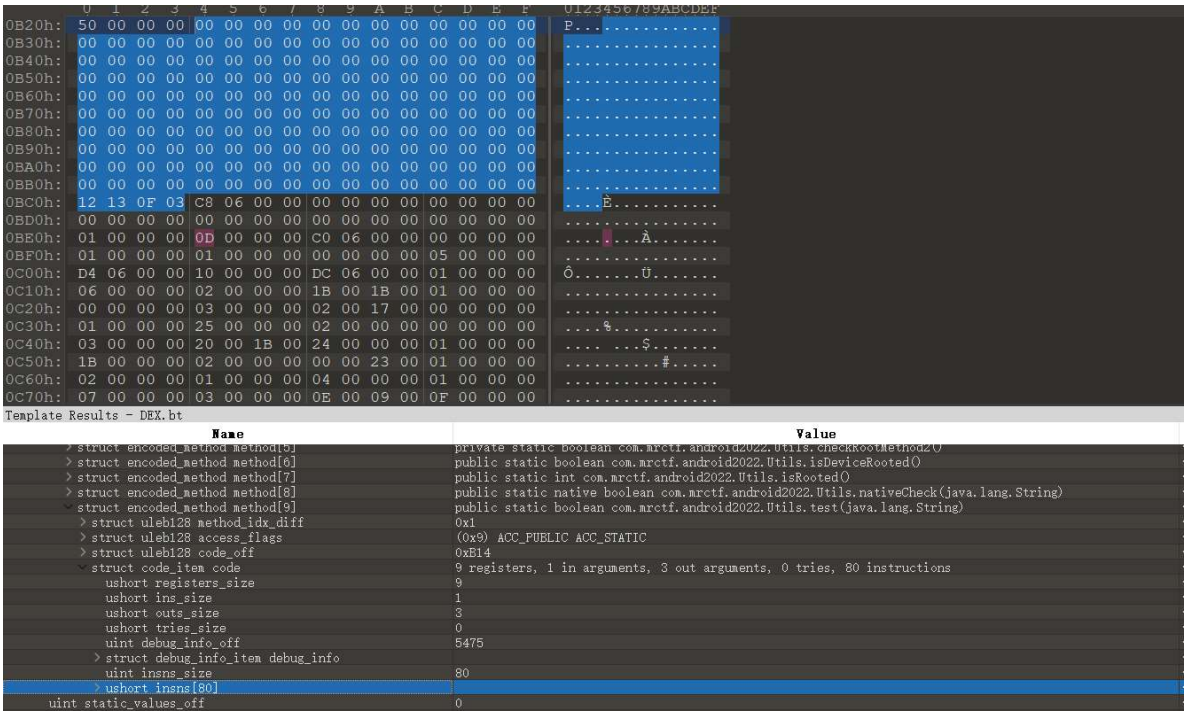
- 第二处使用时仅从156 - 159四个字节进行了修改

```
      }
      if ( v32 != 1351089396 )
        break;
      *(shm + 156) = 18;
      *(shm + 157) = 19;
      *(shm + 158) = 15;
      *(shm + 159) = 3;
      v32 = 1991675850;
    }
    if ( v32 != 1381819862 )
      break;
    ++v37;
    v32 = 1280132146;
  }
```

- 第三处是赋0值，而且在大循环里

```
      v32 = v9;
    }
    if ( v32 != -1233480869 )
      break;
    *(shm + v37) = 0;
    v32 = -2067267860;
  }
  if ( v32 != -951427033 )
    break;
  v32 = -45173303;
}
```

重点关注156 - 159四字节，18 19 15 3 ,其实就是test方法的insn指令的最后四字节==12 13 0F 03==



尝试将赋值数组填回dex,刚好0x80字节

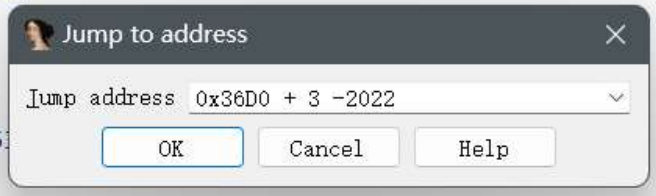
```
12 17 12 01 6E 10 27 00 08 00 0C 02 12 01 21 24 D8 04 04 FF 35 41 10 00 48 04 02
01 D8 05 01 01 48 05 02 05 B7 54 B7 14 8D 44 4F 04 02 01 D8 01 01 01 28 EE 22 00
1F 00 70 30 31 00 70 02 12 04 1F 04 20 00 22 05 1C 00 70 10 2C 00 05 00 1A 06 00
00 6E 20 2E 00 65 00 0C 05 21 26 DA 06 06 02 6E 20 2D 00 65 00 0C 05 1A 06 8D 00
6E 20 2E 00 65 00 0C 05 6E 10 2F 00 05 00 0C 05 23 76 24 00 12 07 4D 00 06 07 71
30 26 00 54 06 0C 04 6E 10 2B 00 04 00 0C 03 71 10 12 00 03 00 0A 04 0F 04
```

使用JEB解析修改后的dex，果然原形毕露，就是将输入经过简单异或后转为hex调用了check函数

```
    }

    public static boolean test(String input) {
        byte[] inputBytes = input.getBytes();
        int v1;
        for(v1 = 0; v1 < inputBytes.length - 1; ++v1) {
            inputBytes[v1] = (byte)(inputBytes[v1] ^ inputBytes[v1 + 1] ^ v1);
        }

        return Utils.check(String.format(null, "%0" + inputBytes.length * 2 + "x", new BigInteger(1, inputBytes)).toLowerCase());
    }
}
```

跳转到check函数的位置，发现存在大量虚假控制流

```
      case 181510861:
        v15[0] = "nativeCheck";
        v15[1] = "(Ljava/lang/String;)Z";
        v15[2] = &off_29BC + 3;
        v15[3] = "check";
        v15[4] = "(Ljava/lang/String;)Z";
        v15[5] = &loc_36D0 + 3;
        v5 = RegisterNatives(env, "com/mrctf/android2022/Utils", v15, 2);
        v6 = -1845948980;
        if ( v5 )
          v6 = -195494424;
        v10 = v6;
        break;
      case 289099020:
        v12 = -1;
        v10 = -167796475
        break;
      case 1331580579:
        v10 = 17669451;
```




```
  qmemcpy(v33, &unk_186D0, sizeof(v33));
  s = GetStringUTFChars(a1, a3, 0);
  v26 = strlen(s);
  size = v26 - (v26 % 8 - 8);
  if ( !(v26 % 8) )
    size = v26 - v26 % 8;
  if ( size != 80 )
  {
    v32 = 0;
    goto LABEL_52;
  }
  v3 = 0;
  v4 = (x_39 * (x_39 - 1)) << 31 == 0;
  if ( y_40 < 10 )
    v3 = 1;
  if ( (v4 & v3) == 0 && !((v4 ^ v3) << 31) )
    goto LABEL_57;
  while ( 1 )
  {
    v24 = malloc(0x50u);
    v5 = 0;
    v28 = 0;
    if ( y_40 > 9 )
      v5 = 1;
    if ( (((x_39 * (x_39 - 1)) & 1) == 0) != y_40 < 10 || ((v5 | (x_39 * (x_39 - 1)) & 1) ^ 1) << 31 )
      break;
LABEL_57:
    malloc(0x50u);
  }
  while ( 1 )
  {
```

先将.bss和.data段设为只读，之后使用IDA脚本设置x.和y.开头变量值为0

```
import ida_bytes
import idautils
for addr,name in idautils.Names():
    if name.startswith('x.') or name.startswith('y.'):
        ida_bytes.patch_dword(addr,0)
```

修改后效果

```
10
11  qmemcpy(cmp, &unk_186D0, sizeof(cmp));
12  s = GetStringUTFChars(a1, a3, 0);
13  v6 = strlen(s);
14  size = v6 - (v6 % 8 - 8);
15  if ( !(v6 % 8) )
16    size = v6 - v6 % 8;
17  if ( size == 80 )
18  {
19    enc = malloc(0x50u);
20    for ( i = 0; (i < 80) << 31; ++i )
21    {
22      if ( i >= v6 )
23        enc[i] = 0;
24      else
25        enc[i] = s[i];
26    }
27    sub_203C(enc, 80);
28    for ( j = 0; j <= 79; ++j )
29    {
30      if ( (((enc[j] & 0xA3 | ~enc[j] & 0xC5B9B45C) ^ (dword_1B384[j] & 0xA3 | ~dword_1B384[j] & 0xC5B9B45C)) != cmp[j]) << 31 )
31        return 0;
32    }
33    return 1;
34  }
35  else
36  {
37    return 0;
38  }
39}
```

```
unsigned int __fastcall sub_203C(char *a1, int a2)
{
  unsigned int result; // r0
  int v3; // r3
  unsigned int v4; // r0
  unsigned int v5; // r0
  int i; // [sp+8h] [bp-2Ch]
  unsigned int v7; // [sp+Ch] [bp-28h]
  int v8; // [sp+18h] [bp-1Ch]
  unsigned int v9; // [sp+1Ch] [bp-18h]
  unsigned int v10; // [sp+1Ch] [bp-18h]

  v7 = 0;
  for ( i = 0; ; i += 8 )
  {
    result = v7;
    if ( v7 >= a2 / 8 )
      break;
    v3 = (a1[i] << 24) | (a1[i + 1] << 16);
    v4 = ((a1[-(-2 - i)] << 8) & 0x5200 | ~(a1[-(-2 - i)] << 8) & 0x68C9ADDA) ^ (v3 & 0x97360000 | ~v3 & 0x68C9ADDA);
    v9 = (v4 & 0x709B9B00 | ~v4 & 0x8F6464FC) ^ (~a1[i + 3] & 0x8F6464FC | a1[i + 3] & 3) | ~(~v4 | ~a1[i + 3]);
    v5 = _byteswap_ulong(*&a1[i + 4]);
    v10 = (v9 & 0x99D9082E | ~v9 & 0x6626F7D1) ^ 0x4604D7F3;
    v8 = v5 & ~v10 | v10 & ~v5;
    a1[i] = HIBYTE(v10);
    a1[i + 1] = BYTE2(v10);
    a1[i + 2] = BYTE1(v10);
    a1[-(-3 - i)] = v10;
    a1[i + 4] = HIBYTE(v8);
    a1[i + 5] = BYTE2(v8);
    a1[-(-6 - i)] = BYTE1(v8);
    a1[-(-7 - i)] = v8;
    ++v7;
  }
  return result;
}
```

逻辑就是对传入的字符串计算长度并进行8位padding，即不足8位补0，如果补完后长度不足80直接返回0，否则传入sub_203c，将其转为伪码就是

```
    int encryptCount = 0;

    int left;
    int right;
    int key = 0x20222022;
    int sum = 0;
    unsigned int i, j;

    encryptCount = length / 8;

    for (i = 0, j = 0; i < encryptCount; i++, j += 8) {
        sum = 0;

        left = data[j] << 24 | data[j + 1] << 16 | data[j + 2] << 8 | data[j +
3];
        right = data[j + 4] << 24 | data[j + 5] << 16 | data[j + 6] << 8 |
data[j + 7];
```

```
        left = left ^ key;
        right = right ^ left;

        data[j] = (left >> 24) & 0xff;
        data[j + 1] = (left >> 16) & 0xff;
        data[j + 2] = (left >> 8) & 0xff;
        data[j + 3] = left & 0xff;
        data[j + 4] = (right >> 24) & 0xff;
        data[j + 5] = (right >> 16) & 0xff;
        data[j + 6] = (right >> 8) & 0xff;
        data[j + 7] = right & 0xff;
    }
```

返回check函数，这里的dword_1B384可以根据索引找到发现是sign的值

```
    }
    if ( v7 == -1077710331 )
      break;
    switch ( v7 )
    {
      case -489556048:
        v7 = 480912003;
        break;
      case 412269416:
        dword_1B384[i] = sign_global[i];
        v7 = -1467821456;
        break;
      case 451480161:
        v5 = 412269416;
        if ( v29 )
          v5 = 1876572082;
        v7 = v5;
        break;
```

整个逻辑转为伪码便是

```
    for(i=0;i<80;i++){
        if( (enc[i] ^ sign[i]) != cmp[i]) return 0;
    }
```

那解题思路应该就明确了，先解出sign，然后解出enc，通过enc反推传入check的字符串，得到后就能得到hex编码，之后就可以得到flag

重新梳理抽取流程便是：在libstub中hook LoadMethod获取到dex在内存中的基址，然后在调用 libnative 的 nativeCheck时还原函数内容，执行结束后填充回去，防止执行后被dump，而这个地址的传输使用的是shm文件的读写

EXP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned char signByte[1023] = {

 0x33,0x31,0x3a,0x31,0x34,0x36,0x34,0x61,0x3b,0x39,0x32,0x39,0x3c,0x3f,0x3f,0x38
,0x71,0x21,0x22,0x20,0x24,0x27,0x26,
0x26,0x28,0x2b,0x2a,0x29,0x2c,0x29,0x29,0x2d,0x44,0x43,0x1b,0x47,0x16,0x15,0x15,
0x17,0x18,0x4d,0x1a,0x36,0x31,0x3b,0x31,0x65,0x3d,0x30,0x33,0x30,0x31,0x3c,0x6d,
0x3b,0x3d,0x6a,0x3f,0x21,
```

```
0x21,0x23,0x23,0x76,0x25,0x23,0x27,0x28,0x2a,0x2a,0x2f,0x24,0x2e,0x2f,0x2f,0x41,
0x12,0x12,0x13,0x1c,0x15,0x10,0x17,0x1b,0x1c,0x1f,0x30,0x35,0x32,0x35,0x35,0x36,
0x36,0x36,0x3f,0x3a,0x39,0x3a,0x3c,0x6c,0x3d,0x3f,0x20,0x29,0x22,0x25,0x24,0x26,
0x23,0x22,0x28,0x2d,0x2a,0x23,0x2d,0x2e,0x2e,0x2e,0x17,0x12,0x11,0x12,0x14,0x44,
0x15,0x17,0x18,0x11,0x1a,0x36,0x31,0x31,0x36,0x31,0x35,0x32,0x37,0x3f,0x38,0x39,
0x3b,0x3d,0x3a,0x3d,0x3c,0x21,0x21,0x73,0x20,0x24,0x25,0x2e,0x27,0x2e,0x29,0x29,
0x2e,0x29,0x2d,0x2a,0x2f,0x41,0x10,0x11,0x13,0x15,0x12,0x15,0x14,0x19,0x19,0x4b,
0x33,0x31,0x32,0x3b,0x34,0x33,0x36,0x34,0x3d,0x3c,0x3a,0x3f,0x3c,0x6f,0x3f,0x3c,
0x20,0x20,0x25,0x20,0x27,0x24,0x26,0x76,0x2b,0x29,0x2a,0x23,0x2c,0x2b,
0x2e,0x2c,0x15,0x14,0x12,0x17,0x14,0x16,0x17,0x14,0x18,0x18,0x1d,0x33,0x32,0x32,
0x32,0x61,0x34,0x31,0x37,0x6c,0x3a,0x38,0x38,0x3d,0x3e,0x3e,0x3c,0x25,0x22,0x23,
0x20,0x27,0x26,0x26,0x24,0x2a,0x2a,0x2b,0x28,0x25,0x2e,0x2d,0x2c,0x10,0x14,0x43,
0x12,0x13,0x15,0x42,0x14,0x1c,0x1a,0x1c,0x33,0x31,0x31,0x36,0x37,0x35,0x35,0x30,
0x3b,0x39,0x39,0x39,0x3f,0x3c,0x3d,0x36,0x23,0x22,0x21,0x23,0x21,0x74,0x25,0x27,
0x2c,0x21,0x29,0x2a,0x2c,0x7c,0x2d,0x2f,0x10,0x19,0x12,
0x15,0x14,0x16,0x13,0x12,0x18,0x1d,0x1a,0x36,0x30,0x31,0x33,0x35,0x32,0x35,0x34,
0x39,0x39,0x6b,0x38,0x3c,0x3d,0x36,0x3f,0x26,0x21,0x21,0x26,0x21,0x25,0x22,0x27,
0x20,0x28,0x29,0x2b,0x2d,0x2a,0x2d,0x2c,0x11,0x11,0x43,0x10,0x14,0x15,0x1e,0x17,
0x1e,0x19,
0x19,0x35,0x34,0x32,0x37,0x34,0x32,0x37,0x34,0x38,0x38,0x3d,0x38,0x3f,0x3c,0x3e,
0x6e,0x23,0x21,0x22,0x2b,0x24,0x23,0x26,0x24,0x2d,0x2c,0x2a,0x2f,0x2c,0x7c,0x2f,
0x2c,0x10,0x10,0x15,0x10,0x17,0x14,0x16,0x46,0x1b,0x19,0x1a,0x38,0x31,0x34,0x33,
0x37,0x30,0x33,0x37,0x3c,0x39,0x68,0x3a,0x3f,0x3d,0x3f,0x38,0x23,0x22,0x23,0x23,
0x75,0x26,0x26,0x27,0x20,0x29,0x2c,0x2b,0x2f,0x28,0x2b,0x2f,0x14,0x11,0x11,0x12,
0x17,0x15,0x17,0x10,0x1b,0x1a,0x1a,0x38,0x33,0x32,0x32,0x36,0x37,0x35,0x37,0x38,
0x6d,0x3a,0x3d,0x3c,0x34,0x3c,0x6e,0x28,0x27,0x26,0x2b,0x2c,0x23,0x70,0x20,0x28,
0x7d,0x2a,0x2a,0x2c,0x2c,0x2e,
0x2e,0x10,0x14,0x12,0x13,0x14,0x16,0x1e,0x15,0x18,0x18,0x1a,0x66,0x31,0x32,0x30,
0x34,0x3d,0x34,0x37,0x39,0x39,0x6b,0x3b,0x3e,0x35,0x3c,0x3f,0x21,0x21,0x23,0x23,
0x24,0x2d,0x27,0x73,0x7b,0x7c,0x2a,0x2e,0x2a,0x2d,0x27,0x7a,0x17,0x44,0x12,0x47,
0x14,0x16,0x13,0x1f,0x4b,0x1a,0x18,0x38,0x33,0x31,0x65,0x60,0x67,0x67,0x35,0x38,
0x3e,0x39,0x39,0x39,0x3d,0x6f,0x39,0x27,0x77,0x71,0x71,0x2c,0x71,0x74,0x2f,0x21,
0x7f,0x2f,0x79,0x28,0x7b,0x2c,0x79,0x41,0x47,0x1b,0x13,
0x42,0x16,0x14,0x13,0x19,0x10,0x48,0x37,0x31,0x3a,0x30,0x62,0x30,0x30,0x36,0x31,
0x6c,0x6c,0x39,0x3e,0x39,0x3d,0x69,0x22,0x25,0x2a,0x2a,0x71,0x23,0x2f,0x26,0x21,
0x2b,0x2b,0x2c,0x7e,0x25,0x2c,0x7d,0x46,0x10,0x15,0x15,0x10,0x47,0x42,0x1e,0x4c,
0x11,0x18,0x39,0x63,0x36,0x36,0x34,0x34,0x31,0x66,0x6a,0x3e,0x32,0x68,0x6a,0x3e,
0x6c,0x3b,0x29,0x23,0x24,0x70,0x23,0x70,0x72,0x75,0x7e,0x2c,0x7e,0x2f,0x2b,0x29,
0x29,0x29,0x41,0x13,0x16,0x45,0x15,0x47,0x1e,0x17,0x4b,0x11,0x1c,0x65,0x30,0x37,
0x34,0x32,0x37,0x32,0x64,0x6c,0x6f,0x6b,0x39,0x6f,0x3a,0x3b,0x3b,0x23,0x25,0x73,
0x22,0x20,0x2d,0x26,0x75,0x2e,0x2d,0x28,0x29,0x2b,0x29,0x2e,0x2a,0x16,0x44,0x43,
0x41,0x15,0x41,0x42,0x44,0x19,0x4f,0x13,0x61,0x31,0x31,0x36,0x33,0x34,0x32,0x64,
0x6c,0x3d,0x33,0x6e,0x6e,0x3b,0x3e,0x3a,0x27,0x26,0x24,0x25,0x70,0x70,0x72,0x23,
0x2b,0x2c,0x7e,0x2d,0x25,0x7f,0x2d,0x7c,0x44,0x16,0x10,0x17,0x47,0x12,0x1f,0x1f,
0x1b,0x4c,0x18,0x38,0x38,0x63,
0x32,0x31,0x64,0x35,0x33,0x3c,0x3b,0x39,0x6e,0x6d,0x3f,0x3b,0x69,0x76,0x20,0x2a,
0x23,0x75,0x77,0x24,0x23,0x2d,0x7f,0x28,0x7a,0x2a,0x2b,0x2f,0x7d,0x18,0x18,0x41,
0x47,0x10,0x11,0x16,0x1e,0x1f,0x4c,0x19,0x64,0x36,0x30,0x61,0x60,0x34,0x60,0x31,
0x30,0x6a,0x3a,0x33,0x38,0x6c,0x3d,0x6e,0x26,0x28,0x71,0x21,0x2d,0x76,0x2f,0x73,
0x29,0x2b,0x78,0x7d,0x7f,0x7e,0x27,0x79,0x11,0x17,0x12,0x17,0x15,0x43,0x13,0x13,
0x18,0x4c,0x18,0x39,0x30,0x61,0x65,0x30,0x30,0x60,0x64,
0x31,0x38,0x3d,0x6f,0x3c,0x6b,0x6b,0x3e,0x21,0x21,0x23,0x22,0x2d,0x73,0x20,0x20,
0x7e,0x7d,0x2c,0x79,0x2d,0x2e,0x28,0x2d,0x41,0x19,0x40,0x13,0x45,0x12,0x15,0x17,
0x11,0x4d,0x49,0x34,0x32,0x34,0x34,0x3d,0x35,0x63,0x33,0x3e,0x6c,0x3c,0x3c,0x6f,
0x3f,0x3d,0x3d,0x27,0x70,0x77,0x20,0x26,0x23,0x72,0x74,0x2c,0x21,0x7b,0x23,0x29,
```

```
0x2d,0x7f,0x2e,0x12,0x13,0x16,0x16,0x47,0x43,0x1f,0x1f,0x4a,0x1c,0x1b,0x36,0x32,
0x34,0x36,0x34,0x60,0x30,0x64,0x6a,0x31,0x33,0x69,0x38,0x3e,0x3a,0x3e,0x27,0x73,
0x25,0x20,0x70,0x23,0x26,0x27,0x2c,0x2d,0x2e,0x2d,0x7f,0x2f,0x26,0x7a,0x18,0x17,
0x47,0x47,0x13,0x11,0x47,0x10,0x10,0x1a,0x48,0x64,0x62,0x67,0x66,0x3d,0x64,0x34,
0x3e,0x3e,0x39,0x38,0x3f,0x34,0x3e,0x3a,0x39,0x73,0x26,0x20,0x76,0x71,
0x70,0x73,0x74,0x7d,0x2a,0x23,0x2b,0x79,0x7b,0x29,0x26,0x13,0x19,0x46,0x46,0x41,
0x1d,0x11,0x17,0x1e,0x4a,0x48,0x34,0x31,0x33,0x60,0x67,0x32,0x33,0x32,0x3a,0x3e,
0x3d,0x3d,0x3c,0x39,0x6d,0x39,0x27,0x21,0x20,0x23,0x27,0x25,0x27,0x27,0x28,0x29,
0x2b,0x7a,0x2f,0x2f,0x2f,0x2c,0x10,0x10
};

void getSign(){
    int i = 0;
    for(i = 0; i < 1023; i++){
        signByte[i] = signByte[i] ^ (i%43);
    }
}

unsigned char getIndex(unsigned char x){
    int i = 0;
    unsigned char charset[17] = "0123456789abcdef" ;
    for(i = 0; i < 16; i++){
        if(charset[i] == x){
            return i;
        }
    }
    return -1;
}

void dec(unsigned char *data, int length)
{
    int encryptCount = 0;

    int left;
    int right;
    int key = 0x20222022;
    int sum = 0;
    unsigned int i, j;

    encryptCount = length / 8;

    for (i = 0, j = 0; i < encryptCount; i++, j += 8)
    {
        sum = 0;

        left = data[j] << 24 | data[j + 1] << 16 | data[j + 2] << 8 | data[j +
3];
        right = data[j + 4] << 24 | data[j + 5] << 16 | data[j + 6] << 8 |
data[j + 7];

        right = right ^ left;
        left = left ^ key;

        data[j] = (left >> 24) & 0xff;
        data[j + 1] = (left >> 16) & 0xff;
        data[j + 2] = (left >> 8) & 0xff;
        data[j + 3] = left & 0xff;
        data[j + 4] = (right >> 24) & 0xff;
```

```
        data[j + 5] = (right >> 16) & 0xff;
        data[j + 6] = (right >> 8) & 0xff;
        data[j + 7] = right & 0xff;
    }
}

int main(){
    int i = 0,j = 0,count = 0;
    char res[80] = {0};
    unsigned char tmp;

    unsigned char cmp[80] =
    {
            0x26, 0x2B, 0x2C, 0x73, 0x14, 0x11, 0x16, 0x13, 0x20, 0x77,
            0x2A, 0x29, 0x13, 0x44, 0x13, 0x1A, 0x75, 0x70, 0x26, 0x21,
            0x12, 0x43, 0x13, 0x12, 0x20, 0x26, 0x23, 0x26, 0x13, 0x45,
            0x11, 0x17, 0x75, 0x70, 0x2C, 0x70, 0x13, 0x1B, 0x14, 0x15,
            0x27, 0x20, 0x25, 0x27, 0x17, 0x4C, 0x15, 0x14, 0x2F, 0x20,
            0x20, 0x78, 0x1F, 0x18, 0x43, 0x46, 0x23, 0x27, 0x22, 0x27,
            0x17, 0x4A, 0x17, 0x10, 0x20, 0x73, 0x20, 0x25, 0x15, 0x14,
            0x12, 0x1B, 0x24, 0x26, 0x10, 0x43, 0x24, 0x27, 0x10, 0x1A
    };
    getSign();
    for(i = 0; i < 80; i++){
        cmp[i] = cmp[i] ^ signByte[i];
    }
    dec(cmp, 80);

    for (count = 0; count < 80; count++)
    {
        if(cmp[count] == 0){
            break;
        }
    }

    for(i = 0; i < count - 1; i += 2){
        res[j++] = getIndex(cmp[i]) << 4 | getIndex(cmp[i+1]);
    }

    for(i = j-2; i >= 0; i--){
        res[i] = res[i] ^ res[i + 1] ^ (i);
    }
    printf("%s", res);
    return 0;
}
```

3. 参考

https://developer.android.com/reference/dalvik/system/InMemoryDexClassLoader

https://blog.csdn.net/shulianghan/article/details/122017822

https://hanshuliang.blog.csdn.net/article/details/121950834#oat_file_assistantccGenerateOatFileNoChecks__385

https://www.jianshu.com/p/ae66be381e6f

http://aospxref.com/android-8.0.0_r36/xref/art/runtime/oat_file_assistant.cc?fi=MakeUpToDate#GenerateOatFileNoChecks

4. 相关开源项目

Allows you to partly emulate an Android native library.
https://github.com/AeonLucid/AndroidNativeEmu

This is a personal improved version of AndroidNativeEmu
https://github.com/maiyao1988/ExAndroidNativeEmu

孤挺花（Armariris） -- 由上海交通大学密码与计算机安全实验室维护的LLVM混淆框架
https://github.com/GoSSIP-SJTU/Armariris

绕过 Android阻止应用动态链接非公开NDK库限制 进行dlopen和dlsym
https://github.com/lizhangqu/dlfcn_compat

thumb16 thumb32 arm32 inlineHook框架
https://github.com/ele7enxxh/Android-Inline-Hook

函数代码抽空解决方案
https://github.com/luoyesiqiu/dpt-shell

ollvm4.0
https://github.com/obfuscator-llvm/obfuscator/tree/llvm-4.0