[原创]百度加固免费版分析及VMP修复 精

卧勤个槽 1

1天前 1006

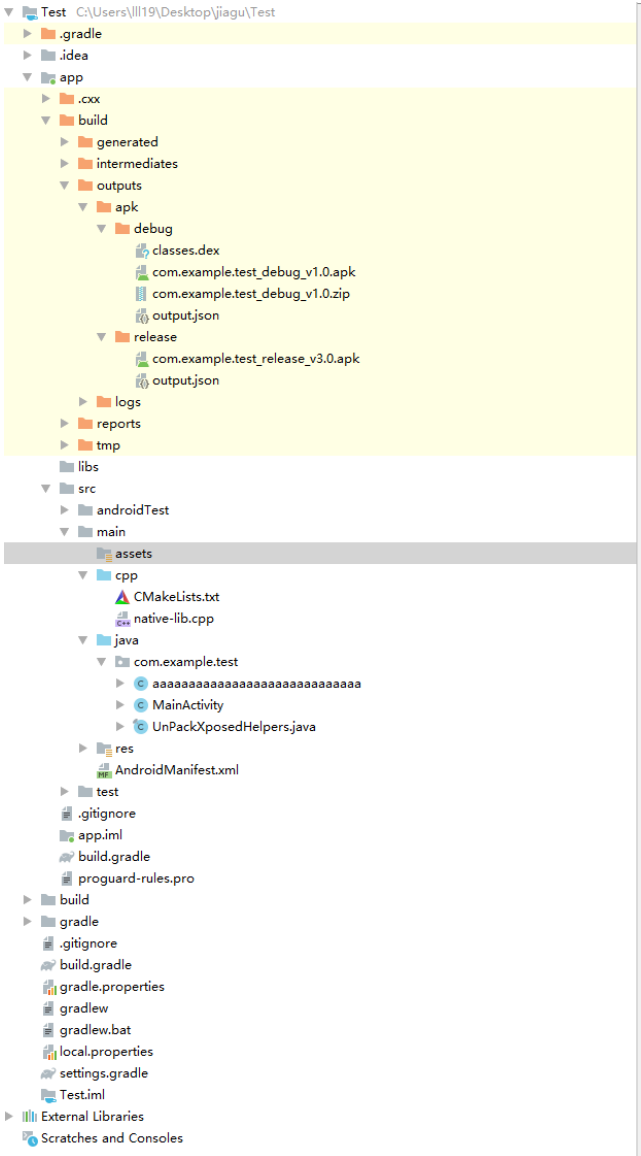
背景

本来计划年后跑路的，不知道是我太菜，还是疫情原因，投简历都没人搭理我。现在又不能出门，只好自己找点事干了。

本文基于Android8.1分析。如果不想看分析过程，可以直接跳到最后的总结。

加固和简单分析

自己随便写个app，上传到百度开发者平台去加固。

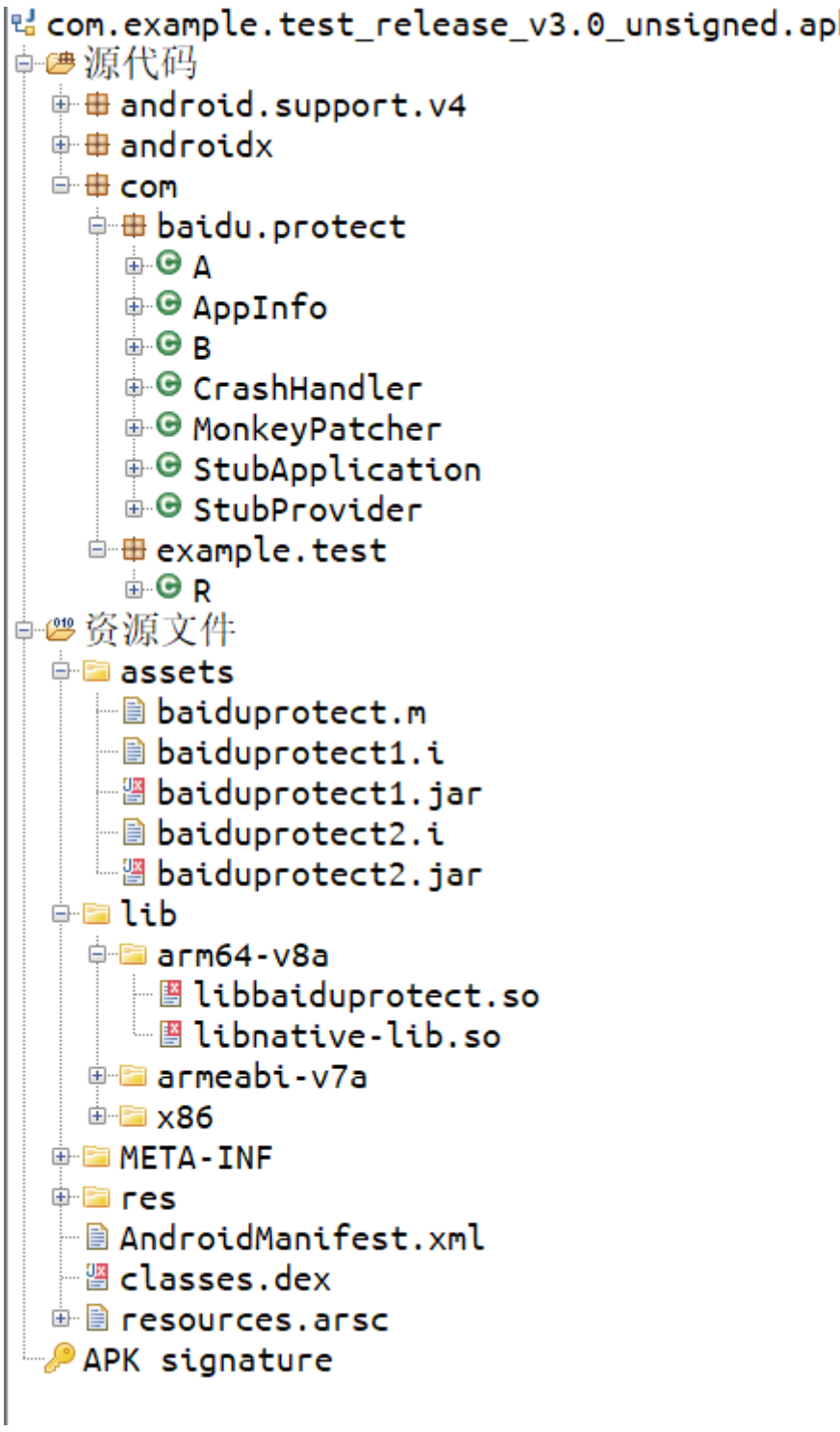



```
@RequiresApi(api = Build.VERSION_CODES.N)
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    // Example of a call to a native method
    TextView tv = findViewById(R.id.sample_text);
    String s = stringFromJNI();
    tv.setText(s);

    try
    {
        Signature[] signatures = getPackageManager().getPackageInfo("com.example.test", GET_SIGNATURES).signatures;
        for (Signature signature : signatures)
        {
            byte[] bytes = signature.toByteArray();
            Log.v(MainActivity.logTag, Class.forName("[Ljava.lang.String;").toString());
        }

        switch (s.length())
        {
            case 0:
                Log.v(MainActivity.logTag, "test_switch_0");
                break;
            case 1:
                Log.v(MainActivity.logTag, "test_switch_1");
                break;
            case 2:
                Log.v(MainActivity.logTag, "test_switch_2");
                break;
            case 3:
                Log.v(MainActivity.logTag, "test_switch_3");
                break;
            case 4:
                Log.v(MainActivity.logTag, "test_switch_4");
                break;
            case 5:
                Log.v(MainActivity.logTag, "test_switch_5");
                break;
            case 6:
                Log.v(MainActivity.logTag, "test_switch_6");
                break;
            case 7:
                Log.v(MainActivity.logTag, "test_switch_7");
                break;
            case 8:
                Log.v(MainActivity.logTag, "test_switch_8");
                break;
            case 9:
                Log.v(MainActivity.logTag, "test_switch_9");
                break;
            case 10:
                Log.v(MainActivity.logTag, "test_switch_10");
                break;
            default:
                Log.v(MainActivity.logTag, "test");
        }
    }
    catch (Throwable e)
    {
        e.printStackTrace();
        Log.v(MainActivity.logTag, "ex:" + e);
    }
}
```

加固后反编译看下。包名com.example.test下原来的类都没了，多了个com.baidu.protect，assets下面多了几个文件，lib下面多了一个so。猜测是通过libbaiduprotect.so将assets下的文件解密出dex，然后加载。



详细分析

通过AndroidManifest.xml知道最先执行的class为com.baidu.protect.StubApplication

找到attachBaseContext方法，可以看到先通过Debug.isDebuggerConnected检测是否被调试，如果被调试就不会加载so，直接进行application替换，这样肯定是不行的，因为原来的class都没有被加载进来，程序直接就会崩溃，所以要调试的话需要把这里过掉。

```
/* access modifiers changed from: protected */
public void attachBaseContext(Context context) {
    mContext = context;
    mBootStrapApplication = this;
    AppInfo.APKPATH = context.getApplicationInfo().sourceDir;
    AppInfo.DATAPATH = getDataFolder(context.getApplicationInfo().dataDir);
    if (!Debug.isDebuggerConnected()) {
        loadLibrary();
        A.n001(AppInfo.PKGNAME, AppInfo.APPNAME, AppInfo.APKPATH, AppInfo.DATAPATH,
    }
    if (AppInfo.APPNAME != null && AppInfo.APPNAME.length() > 0) {
        mRealApplication = MonkeyPatcher.createRealApplication(AppInfo.APPNAME);
    }
    super.attachBaseContext(context);
    if (mRealApplication != null) {
        MonkeyPatcher.attachBaseContext(context, mRealApplication);
    }
}
```

```
    private static void loadLibrary() {
        if ((AppInfo.SUPPORT_ARCH == null || AppInfo.SUPPORT_ARCH.length() < 3) && isX86ABI()) {
            loadX86Library();
        } else {
            System.loadLibrary(AppInfo.LIBNAME);
        }
    }
```

```
/* compiled from: BS544 */
public class AppInfo {
    public static String APKPATH = "";
    public static String APPNAME = null;
    public static String APP_VERSION = "3.0";
    public static String DATAPATH = "";
    public static String LIBNAME = "baiduprotect";
    public static String PKGNAME = "com.example.test";
    public static boolean RELOAD_SP = false;
    public static boolean REPORT_CRASH = true;
    public static String SUPPORT_ARCH = "armeabi-v7a";
    public static String VERSION = "ARA5.6.2";
    public static String sdkVersion = "19";
    public static String targetSdkVersion = "28";
}
```

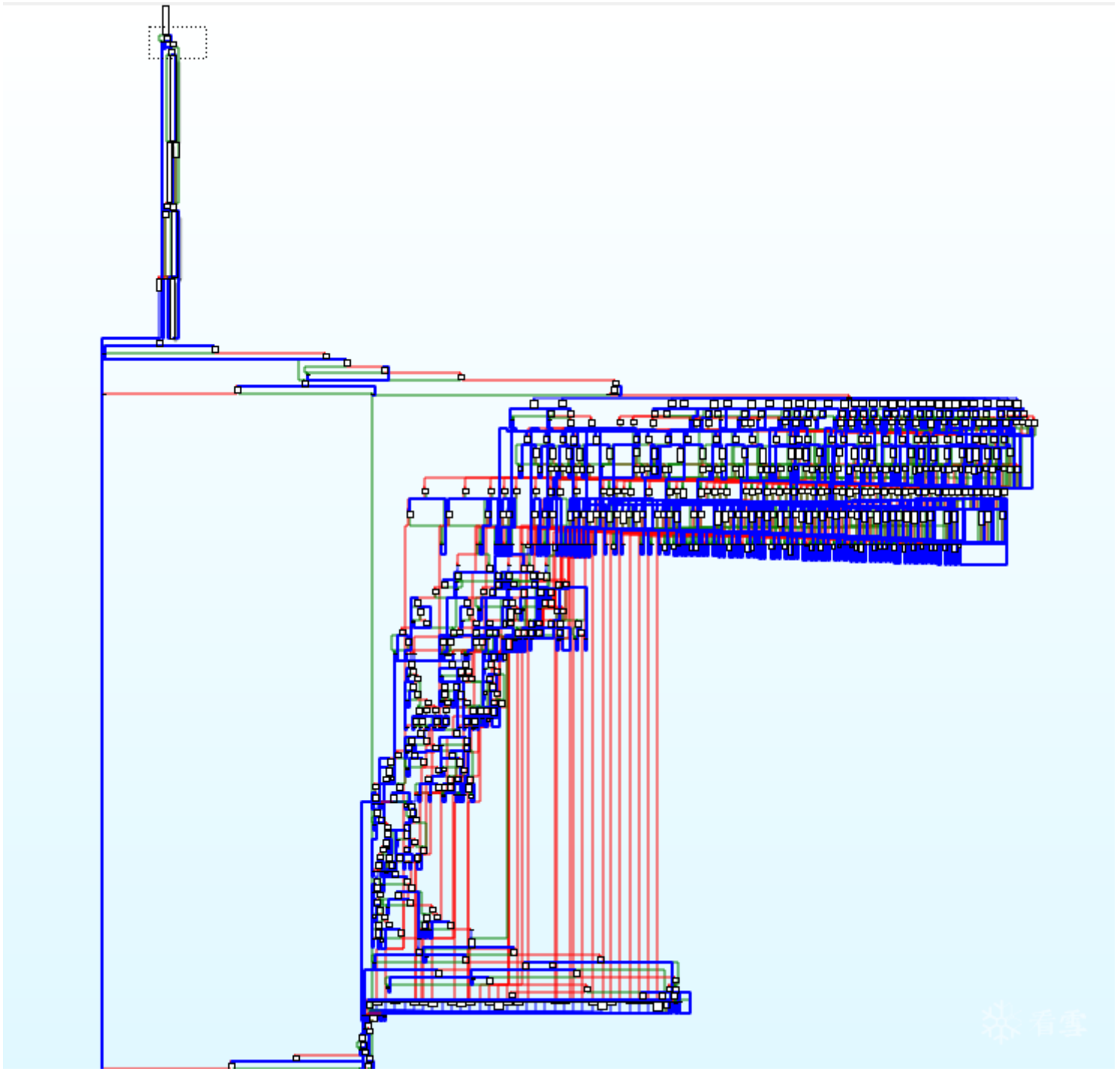
将libbaiduprotect.so用ida打开。可以看到有.init_array，这个数组包含好几个函数，而其他很多函数，包括JNI_OnLoad都被加密了。

Name	Start	End	R	W	X	D	L	Align	Bas	Type	Class	AD	T	DS
LOAD	0000000000000000	0000000000006A40	R	.	X	.	L	byte	01	public	CODE	64	00	0E
.plt	0000000000006A40	0000000000006F90	R	.	X	.	L	para	03	public	CODE	64	00	0E
.text	0000000000006F90	0000000000088030	R	.	X	.	L	para	04	public	CODE	64	00	0E
.gnu.text	0000000000088030	00000000000959A8	R	.	X	.	L	dword	05	public	CODE	64	00	0E
LOAD	00000000000959A8	00000000000959B0	R	.	X	.	L	byte	01	public	CODE	64	00	0E
.rodata	00000000000959B0	000000000009CB78	R	.	.	.	L	para	06	public	CONST	64	00	0E
.eh_frame_hdr	000000000009CB78	000000000009F2C4	R	.	.	.	L	dword	07	public	CONST	64	00	0E
LOAD	000000000009F2C4	000000000009F2C8	R	.	X	.	L	byte	01	public	CODE	64	00	0E
.eh_frame	000000000009F2C8	00000000000AA938	R	.	.	.	L	qword	08	public	CONST	64	00	0E
.gcc_except_table	00000000000AA938	00000000000AABA4	R	.	.	.	L	dword	09	public	CONST	64	00	0E
.init_array	00000000000BB560	00000000000BB5C0	R	W	.	.	L	qword	0A	public	DATA	64	00	0E
.fini_array	00000000000BB5C0	00000000000BB5D0	R	W	.	.	L	qword	0B	public	DATA	64	00	0E
.data.rel.ro	00000000000BB5D0	00000000000BC800	R	W	.	.	L	para	0C	public	DATA	64	00	0E
LOAD	00000000000BC800	00000000000BC9F0	R	W	.	.	L	byte	02	public	DATA	64	00	0E
.got	00000000000BC9F0	00000000000BCFF8	R	W	.	.	L	qword	0D	public	DATA	64	00	0E
LOAD	00000000000BCFF8	00000000000BD000	R	W	.	.	L	byte	02	public	DATA	64	00	0E
.data	00000000000BD000	00000000000BE1F8	R	W	.	.	L	para	0E	public	DATA	64	00	0E
LOAD	00000000000BE1F8	00000000000BE200	R	W	.	.	L	byte	02	public	DATA	64	00	0E
.bss	00000000000BE200	00000000000C0148	R	W	.	.	L	para	0F	public	BSS	64	00	0E
.prgend	00000000000C0148	00000000000C0149	?	?	?	.	L	byte	10	public		64	00	10
extern	00000000000C0150	00000000000C0400	?	?	?	.	L	qword	11	public		64	00	11

```
nit_array:00000000000BB560 ; Segment type: Pure data
nit_array:00000000000BB560                 AREA .init_array, DATA, ALI(
nit_array:00000000000BB560                 ; ORG 0xBB560
nit_array:00000000000BB560 off_BB560       DCQ sub_88060           ; DA
nit_array:00000000000BB560                                         ; L(
nit_array:00000000000BB568                 DCQ sub_6FC4
nit_array:00000000000BB570                 DCQ sub_705C
nit_array:00000000000BB578                 DCQ sub_70F4
nit_array:00000000000BB580                 DCQ sub_718C
nit_array:00000000000BB588                 DCQ sub_7300
nit_array:00000000000BB590                 DCQ sub_7398
nit_array:00000000000BB598                 DCQ sub_74B8
nit_array:00000000000BB5A0                 DCQ sub_7578
nit_array:00000000000BB5A8                 DCQ sub_76BC
nit_array:00000000000BB5B0                 DCQ sub_7744
nit_array:00000000000BB5B8                 ALIGN 0x20
nit_array:00000000000BB5B8 ; .init_array   ends
text:0000000000007C00 ; } // starts at 7BC4
text:0000000000007C04 ; ---------------------------------------------------------------------------
text:0000000000007C04
text:0000000000007C04                 EXPORT JNI_OnLoad
text:0000000000007C04 JNI_OnLoad                          ; DATA XREF: LOAD:0000000000000628↑o
text:0000000000007C04 ; __unwind {
text:0000000000007C04                 STLXRB          W0, W18, [X2]
text:0000000000007C08                 ORR             W20, W0, W0
text:0000000000007C0C                 SUB             W20, W2, W0
text:0000000000007C10                 FMADD           D17, D3, D1, D0
text:0000000000007C10 ; ---------------------------------------------------------------------------
text:0000000000007C14                 DCD 0xE81F40B9
text:0000000000007C18                 DCQ 0x87D0453E9A30329, 0xE82740B909008052, 0xA8C35EB8E82700B9
text:0000000000007C18                 DCQ 0x227D40D300013FD6, 0xE01740F9A1035FF8, 0x80140F9A9C35EB8
text:0000000000007C18                 DCQ 0x82947F9E01700F9, 0x13FD6A80200F0, 0xA9C35EB8207D40D3
text:0000000000007C18                 DCQ 0x8C547F9080140F9, 0x12000014A80200F0, 0xE8031C32E82700B9
text:0000000000007C18                 DCQ 0x4105805200013FD6, 0x80010A8B227D40D3, 0x4A7D40D329010B6B
text:0000000000007C18                 DCQ 0xABC35EB8EC1740F9, 0x80140F9AAC35EB8, 0x8F546F9E9031C32
text:0000000000007C18                 DCQ 0x13FD6A80200F0, 0xA1035FF8227D40D3, 0xA9C35EB8E01740F9
text:0000000000007C18                 DCQ 0xE01700F9080140F9, 0xA80200F0082947F9, 0xE0031C3200013FD6
```

现在详细分析.init_array中的每一个函数。

先看第一个函数sub_88060，一看就是被混淆过的。



先跳过第一个函数，看下后面的。进到函数sub_6FC4，看下c代码。没有被加密，也没有被混淆，但是其中调用的loc_3E73C被加密了。

```
__int64 sub_6FC4()
{
  __int64 v0; // x0
  __int64 v1; // x19

  v0 = ((__int64 (__fastcall *)(const char *))qword_28C28[0])("710E34DFB3D273975E0936FBB4CF62BE541F3CC4A8");
  ((void (__fastcall *)(__int64 *, __int64, signed __int64))qword_3E590[0])(&qword_BE448, v0, 1LL);
  v1 = qword_BE7B8;
  qword_BE448 = (__int64)&off_BB5E0;
  if ( !qword_BE7B8 )
  {
    v1 = sub_82254(336LL);
    ((void (*)(void))loc_3E5A8)();
    qword_BE7B8 = v1;
  }
  ((void (__fastcall *)(__int64, __int64 *))loc_3E73C)(v1, &qword_BE448);
  return __cxa_atexit(qword_B3A0, &qword_BE448, &unk_BD000);
}
```

```
.text:000000000003E73C
.text:000000000003E73C loc_3E73C                       ; CODE XREF: sub_6FC4+74↑p
.text:000000000003E73C                                 ; sub_705C+74↑p ...
.text:000000000003E73C ; __unwind {
.text:000000000003E73C                 STXRB           W1, W11, [X30]
.text:000000000003E73C ; ---------------------------------------------------------------------------
.text:000000000003E740                 DCQ 0x91D4038E9FFFF35, 0xF30300AA680600D1, 0xFD7B01A9FD430091
.text:000000000003E740                 DCQ 0xC0035FD6F44FBEA9, 0xFD7B41A9F44FC2A8, 0x8090091E9FDFF35
.text:000000000003E740                 DCQ 0x9014039CE050011, 0x2900094AE9150038, 0x2901120B2901110B
.text:000000000003E740                 DCQ 0xA1C96E380912090B, 0xDF210071EE038E1A, 0x3FEA007192318B1A
.text:000000000003E740                 DCQ 0x1FEA0071F0338A1A, 0x301D001211F15F38, 0xAD4D1991EF0300AA
.text:000000000003E740                 DCQ 0xC1A80524D0300B0, 0xA1280522B198052, 0xEE031F2A680A0091
.text:000000000003E740                 DCQ 0x6902403969030034, 0xEE5DFF971F683438, 0x14FD41D380060091
.text:000000000003E740                 DCQ 0xE9FFFF35080113CB, 0x680600D1091D4038, 0xFD430091F30300AA
.text:000000000003E740                 DCQ 0xF44FBEA9FD7B01A9, 0xF44FC2A8C0035FD6, 0xE9FDFF35FD7B41A9
.text:000000000003E740                 DCQ 0xCE05001108090091, 0xE915003809014039, 0x2901110B2900094A
.text:000000000003E740                 DCQ 0x912090B2901120B, 0xEE038E1AA1C96E38, 0x92318B1ADF210071
.text:000000000003E740                 DCQ 0xF0338A1A3FEA0071, 0x11F15F381FEA0071, 0xEF0300AA301D0012
.text:000000000003E740                 DCQ 0x4D0300B0AD2D1991, 0x2B1980520C1A8052, 0x680A00910A128052
.text:000000000003E740                 DCQ 0x69030034EE031F2A, 0x1F68343869024039, 0x80060091195EFF97
.text:000000000003E740                 DCQ 0x80113CB14FD41D3, 0x91D4038E9FFFF35, 0xF30300AA680600D1
.text:000000000003E740                 DCQ 0xFD7B01A9FD430091, 0xC0035FD6F44FBEA9, 0xFD7B41A9F44FC2A8
.text:000000000003E740                 DCQ 0x8090091E9FDFF35, 0x9014039CE050011, 0x2900094AE9150038
.text:000000000003E740                 DCQ 0x2901120B2901110B, 0xA1C96E380912090B, 0xDF210071EE038E1A
.text:000000000003E740                 DCQ 0x3FEA007192318B1A, 0x1FEA0071F0338A1A, 0x301D001211F15F38
.text:000000000003E740                 DCQ 0xAD0D1991EF0300AA, 0xC1A80524D0300B0, 0xA1280522B198052
.text:000000000003E740                 DCQ 0xEE031F2A680A0091, 0x6902403969030034, 0x445EFF971F683438
.text:000000000003E740                 DCQ 0x14FD41D380060091, 0xE9FFFF35080113CB, 0x680600D1091D4038
```

现在只能回去分析第一个函数sub_88060了，把垃圾代码删除之后，分析出调用流程，发现全是一些字符操作，应该是在解密，没有发现有反调试的地方。

```
case 1://->24
case 2://->28
case 3://->27
case 4://->18
case 5://->25
case 6://->17/29
case 7://->4
case 8://->30
case 9://->6
case 10://->26
case 11://->19
case 12://->5
case 13://->15
case 14://->9
case 15://->8/return
case 16://->6
case 17://->10
case 18://->22
case 19://->3
case 20://->2
case 21://->13
case 22://->12
case 23://->7
case 24://->11
case 25://->20
case 26://->16
case 27://->23
case 28://->14
case 29://->31
case 30://->1
case 31://->21
```

```
init=13
                                                                ->29->31->21->13...
13->15->8->30->1->24->11->19->3->27->23->7->4->18->22->12->5->25->20->2->28->14->9->6->17->10->26->16->6...
     ->return
```

所以直接动态调试，当第一个函数sub_88060执行完后，dump解密后的so。将其用ida打开，可以看到函数都已经被解密了。

```
.text:0000000000007C04
.text:0000000000007C04 ; Attributes: bp-based frame
.text:0000000000007C04
.text:0000000000007C04                 EXPORT JNI_OnLoad
.text:0000000000007C04 JNI_OnLoad                              ; DATA XREF: LOAD:0000000000000628↑o
.text:0000000000007C04
.text:0000000000007C04 var_40          = -0x40
.text:0000000000007C04 var_30          = -0x30
.text:0000000000007C04 var_20          = -0x20
.text:0000000000007C04 var_10          = -0x10
.text:0000000000007C04 var_s0          =  0
.text:0000000000007C04
.text:0000000000007C04 ; __unwind {
.text:0000000000007C04                 STP             X26, X25, [SP,#-0x10+var_40]!
.text:0000000000007C08                 STP             X24, X23, [SP,#0x40+var_30]
.text:0000000000007C0C                 STP             X22, X21, [SP,#0x40+var_20]
.text:0000000000007C10                 STP             X20, X19, [SP,#0x40+var_10]
.text:0000000000007C14                 STP             X29, X30, [SP,#0x40+var_s0]
.text:0000000000007C18                 ADD             X29, SP, #0x40
.text:0000000000007C1C                 MOV             X21, X1
.text:0000000000007C20                 MOV             X22, X0
.text:0000000000007C24                 ADRP            X8, #dword_C0124@PAGE
.text:0000000000007C28                 ADRP            X9, #dword_C011C@PAGE
.text:0000000000007C2C                 MOV             W10, #1
.text:0000000000007C30                 ADD             X9, X9, #dword_C011C@PAGEOFF
.text:0000000000007C34                 LDR             W9, [X9]
.text:0000000000007C38                 SUBS            W10, W9, W10
.text:0000000000007C3C                 MADD            W9, W9, W10, WZR
.text:0000000000007C40                 MOV             W10, #1
.text:0000000000007C44                 ADD             X8, X8, #dword_C0124@PAGEOFF
.text:0000000000007C48                 LDR             W8, [X8]
.text:0000000000007C4C                 AND             W9, W9, W10
.text:0000000000007C50                 CMP             W9, #0
.text:0000000000007C54                 CSET            W9, EQ
.text:0000000000007C58                 CMP             W8, #0xA
.text:0000000000007C5C                 CSET            W8, LT
.text:0000000000007C60                 ORR             W8, W9, W8
.text:0000000000007C64                 CMP             W8, #0
.text:0000000000007C68                 B.EQ            loc_8F80
.text:0000000000007C6C
.text:0000000000007C6C loc_7C6C                                ; CODE XREF: JNI_OnLoad:loc_8F80↓j
.text:0000000000007C6C                 ADRP            X8, #dword_C0124@PAGE
```

继续分析.init_array中的函数，通过分析，从sub_6FC4到sub_7578这8个函数都在做一件事情，就是向一个列表添加函数，每个函数添加的时候会指定一个数字(索引)，将其通过索引排序插入列表。后面的过程中会按顺序调用列表中所有函数。

这8个函数这里就分析其中的sub_6FC4，其余的都类似，有兴趣的可以自己去看。

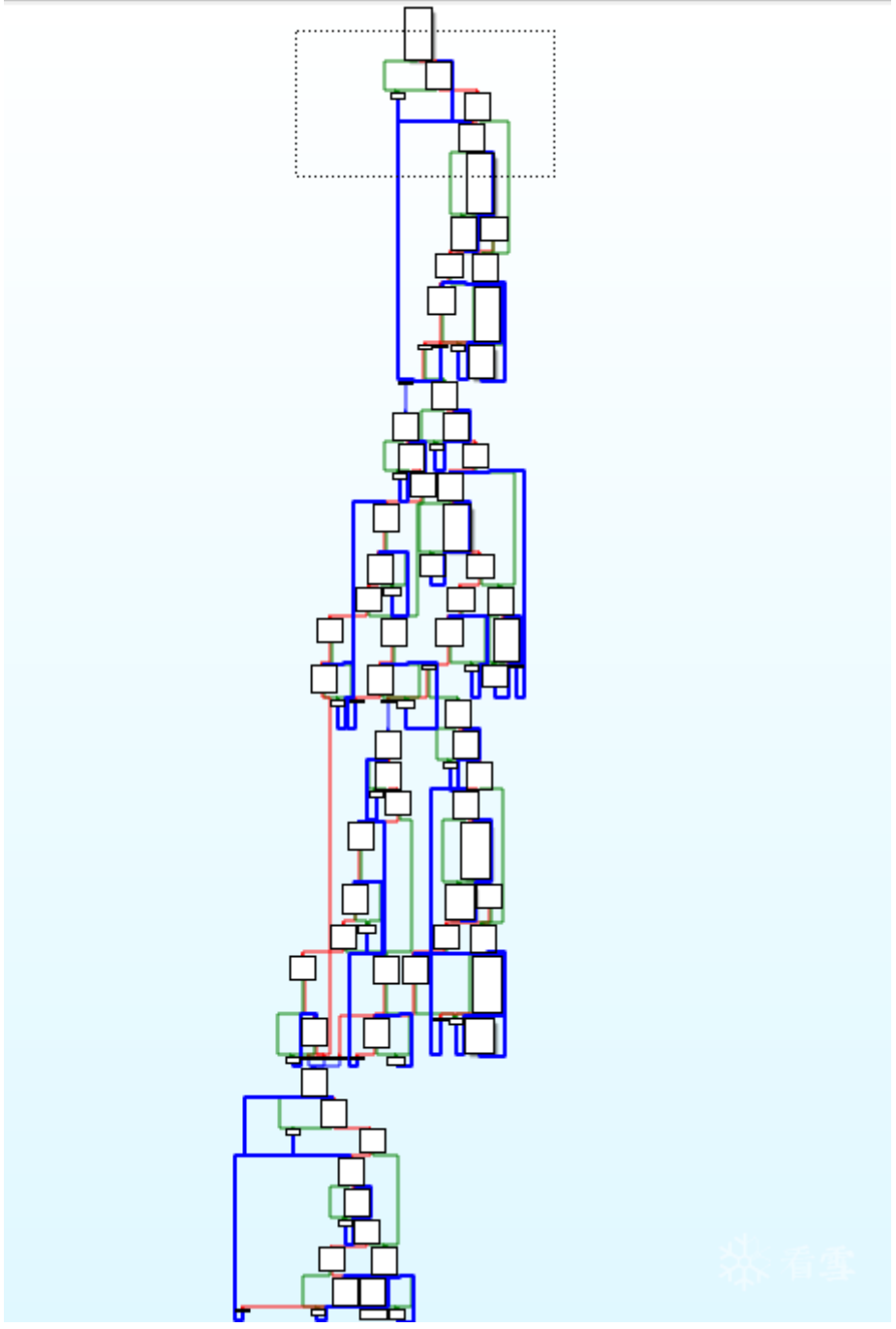
为了防止静态分析，so中所有字符串都被加密的，每个字符串的解密函数都不一样，但是原理都是一样的，只是解密用的key不一样。这里sub_28C28就是一个将字符串解密的函数，然后通过函数sub_3E590初始化一个调用对象，这里的第三个参数(1)就是索引，off_BB5E0是一个指针，里面存着调用的函数地址，qword_BE7B8就是函数列表，函数sub_3E73C用于将刚刚构造的调用对象添加到列表中。

```
1  __int64 sub_6FC4()
2  {
3    __int64 v0; // x0
4    __int64 v1; // x19
5
6    v0 = sub_28C28("710E34DFB3D273975E0936FBB4CF62BE541F3CC4A8");
7    sub_3E590((__int64)&qword_BE448, v0, 1);
8    v1 = qword_BE7B8;
9    qword_BE448 = (__int64)off_BB5E0;
10   if ( !qword_BE7B8 )
11   {
12     v1 = sub_82254(336LL);
13     sub_3E5A8();
14     qword_BE7B8 = v1;
15   }
16   sub_3E73C(v1, (__int64)&qword_BE448);
17   return __cxa_atexit(sub_B3A0, &qword_BE448, &unk_BD000);
18 }
```

最后分析得到一个索引（即调用顺序）和函数的映射关系。

```
1   sub_B3B4
3   sub_3E29C
6   sub_40CF8
7   sub_3DFC4
8   sub_11F5C
9   sub_45964
10  sub_3E96C
13  sub_42388
```

.init_array分析完了，就该JNI_OnLoad了。可以看到它也被混淆了。



将垃圾代码删除后，得到真实流程。开始我以为会通过时间来判断是否被调试，其实这里获取时间只是统计信息上报。

```
    *(&v18 - 2) = 0LL;
    gettimeofday(&stru_BE208, 0LL);
    v7 = (unsigned __int64)sub_91D8(v18, &v18 - 2, 65540LL) != 0;//GetEnv(vm);

    if ( !v7 )
    {
        qword_BE568 = v18;
        v8 = sub_7BC4();//get_call_back_queue
        sub_3E628(v8, 1LL, *v5, 0LL, 0LL, 0LL);//call_back_queue
        v9 = (unsigned __int64)sub_91E4(*v5) != 0;//FindClass_and_RegisterNatives
        if ( v9 )
        {
            *(_DWORD *)v4 = -1;
        }
        else
        {
            v10 = sub_7BC4();
            sub_3E628(v10, 2LL, *v5, 0LL, 0LL, 0LL);
            gettimeofday(&v18 - 2, 0LL);
            qword_BE218 = 1000000 * (*(&v18 - 2) - qword_BE208) + *(&v18 - 1) - qword_BE210;//消耗的微秒
            *(_DWORD *)v4 = 65540;
        }
    }
    result = *(unsigned int *)v4;
    return result;
```

这里的函数sub_3E628就是调用队列中的所有函数，它的第2个参数会被作为参数传到函数中，判断函数是否该执行。

```
__int64 __fastcall sub_3E628(__int64 result, int a2)
{
  int v2; // w10
  signed __int64 v3; // x11
  signed __int64 v4; // x24
  __int64 (****v5)(void); // x25
  __int64 (*****v6)(void); // x24
  __int64 (****v7)(void); // t1
  int v8; // w27
  __int64 (***v9)(void); // t1
  signed int v10; // w8
  __int64 v11; // [xsp+8h] [xbp-68h]
  signed __int64 v12; // [xsp+10h] [xbp-60h]
  int v13; // [xsp+1Ch] [xbp-54h]

  v11 = result;
  v2 = 10000000 * a2 + 101000;
  v3 = 1LL;
  do
  {
    v12 = v3;
    v13 = v2;
    v4 = v11 + 24 * v3;
    v5 = *(__int64 (*****)(void))(v11 + 24 * v3);
    v7 = *(__int64 (*****)(void))(v4 + 8);
    v6 = (__int64 (*****)(void))(v4 + 8);
    v8 = v2;
    if ( v5 != v7 )
    {
      do
      {
        dword_BE588 = v8;
        v9 = *v5;
        ++v5;
        result = (**v9)();
        if ( result & 1 )
          v10 = 2;
        else
          v10 = 1;
        dword_BE588 = v10 | v8;
        v8 += 1000;
      }
      while ( v5 != *v6 );
    }
    v3 = v12 + 1;
    v2 = v13 + 100000;
  }
  while ( v12 != 13 );
  return result;
}
```

所以，首先用1作为参数调用队列中的所有函数，通过分析只有sub_B3B4会执行。该函数通过dlsym将libc中的符号获取保存下来，之后调用这些函数都通过指针调用。

```

  if ( a2 != 1 )
    return 0LL;
  v2 = sub_28AD0("F48398ED31542B0E");           // libc.so
  v3 = dlopen(v2, 1LL);
  if ( !v3 )
    exit(1LL);
  v4 = sub_25E78("6E2DCEF43F47B331");           // _exit
  *(_QWORD *)exit = dlsym(v3, v4);
  v5 = sub_25F24("04C0441C5939526D");           // exit
  *(_QWORD *)exit_0 = dlsym(v3, v5);
  v6 = sub_25FD0("171DD59574B03063041BD88665B4");// pthread_create
  pthread_create = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD))dlsym(v3, v6);
  v7 = sub_2607C("D4906DE875BD569CCE8B6CF4");   // pthread_join
  pthread_joinpthread_join = dlsym(v3, v7);
  v8 = sub_26128("6C9860A90F4D93A7");           // memcpy
  memcpy_0 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD))dlsym(v3, v8);
  v9 = sub_261D4("2D5E4DE7FD827D7E");           // malloc
  *(_QWORD *)malloc_0 = dlsym(v3, v9);
  v10 = sub_26280("070C098F58629CFF");          // calloc
  *(_QWORD *)calloc_0 = dlsym(v3, v10);
  v11 = sub_2632C("3D78BB624F4927F7");          // memset
  memset_0 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD))dlsym(v3, v11);
  v12 = sub_263D8("C1633EFFD2BBA4ED");          // fopen
  fopen_0 = dlsym(v3, v12);
  v13 = sub_26484("ED617095DDFEE33B");          // fclose
  fclose_0 = (__int64 (__fastcall *)(_QWORD))dlsym(v3, v13);
  v14 = sub_26530("E24E68C72ED3BB4E");          // fgets
  fgets_0 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD))dlsym(v3, v14);
  v15 = sub_265DC("4EFE6FC425A325B6");          // strtoul
  *(_QWORD *)strtoul = dlsym(v3, v15);
  v16 = sub_26688("30640DB4A5A47CCE");          // strtoull
  *(_QWORD *)strtoull = dlsym(v3, v16);
  v17 = sub_26734("59CA0B1C240EE666");          // strstr
  *(_QWORD *)strstr_0 = dlsym(v3, v17);
  v18 = sub_267E0("B3783A25593967C1");          // ptrace
  ptrace = dlsym(v3, v18);
  v19 = sub_2688C("715D268CE678C6E2");          // mprotect
  mprotect = dlsym(v3, v19);
  v20 = sub_26938("678568CC1272D9F4");          // strlen
```

继续回到JNI_OnLoad，执行完函数队列中的函数，就该执行函数sub_91E4了，该函数是注册com.baidu.protect.A中的部分本地函数，n001,n002,n003,分别对应的函数为sub_9318，sub_94E4，sub_9564

```
__int64 __fastcall FindClass_and_RegisterNatives(JNIEnv *a1)
{
  int v1; // w0
  jclass clazz; // [xsp+10h] [xbp-60h]
  JNINativeMethod methods[3]; // [xsp+18h] [xbp-58h]
  JNIEnv *v5; // [xsp+60h] [xbp-10h]
  unsigned int v6; // [xsp+6Ch] [xbp-4h]

  v5 = a1;
  methods[0].name = sub_25FD0("09598DD611D1543C");// void n001(String str, String str2, String str3, String str4, int i, boolean z);
  methods[0].signature = sub_2607C(
                           "8CA86FFB66BD1DAFC58A62B543A840AACA833ED67ABD44A28B8864F477F361B7D68D6BFD2B9058A2D2852AF671B25"
                           "5ECF79077F37EBB098FCE8573FB3FB053ADC3CB56EE62B55CA49FAD5FB346");
  methods[0].fnPtr = n001;
  methods[1].name = (const char *)sub_26128("6FCD3DF87F3493A7");
  methods[1].signature = (const char *)sub_261D4("687340E5F6931217241042E4FC951810341062E4FC951806340408DD");// void n002(Context context);
  methods[1].fnPtr = n002;
  methods[2].name = sub_26280("0A5D55D037019CFF");
  methods[2].signature = sub_2632C("783480112A3D27F7");// void n003();
  methods[2].fnPtr = n003;
  clazz = (jclass)FindClass(v5, (const char *)str_class_name_com_baidu_protect_A);// "com/baidu/protect/A"
  if ( clazz )
  {
    v1 = RegisterNatives(v5, clazz, methods, 3);
    if ( v1 < 0 )
      v6 = -1;
    else
      v6 = 0;
  }
  else
  {
    v6 = -1;
  }
  return v6;
}
```

```
package com.baidu.protect;

import android.content.Context;

public class A {
    public static final String BAIDUPROTECT_TAG = "baiduprotect5.0.1";

    public static native byte B(int i, Object obj, Object... objArr);

    public static native char C(int i, Object obj, Object... objArr);

    public static native double D(int i, Object obj, Object... objArr);

    public static native float F(int i, Object obj, Object... objArr);

    public static native int I(int i, Object obj, Object... objArr);

    public static native long J(int i, Object obj, Object... objArr);

    public static native Object L(int i, Object obj, Object... objArr);

    public static native short S(int i, Object obj, Object... objArr);

    public static native void V(int i, Object obj, Object... objArr);

    public static native boolean Z(int i, Object obj, Object... objArr);

    public static native void n001(String str, String str2, String str3, String

    public static native void n002(Context context);

    public static native void n003();
}
```

继续回到JNI_OnLoad，注册函数后，失败就直接返回了，成功则以2作为参数再次调用队列中的所有函数。通过分析，只有sub_3E29C会执行。可以看到，该函数可以接受2和3，现在我们只看参数为2的情况。

```
signed __int64 __fastcall sub_3E29C(__int64 a1, int a2, __int64 a3)
{
  __int64 v3; // x19
  char v5; // [xsp+8h] [xbp-38h]

  v3 = a3;
  if ( a2 == 3 )
  {
    sub_13880(a3, *((_QWORD *)&xmmword_BE5F0 + 1), &v5);
    sub_66064(&qword_BE690, v3, &v5, 16LL);
  }
  else if ( a2 == 2 )
  {
    sub_3E36C(0LL, a3);
    return 1LL;
  }
  return 0LL;
}
```

现在进入sub_3E36C，其主要就是调用sub_3E3F0，而sub_3E3F0就是通过读取/proc/self/maps文件，通过加载的虚拟机文件，判断虚拟机类型。

```
// 通过maps文件获取vm类型
baidu_enum_vm_type __cdecl get_vm_type()
{
  _BYTE *v0; // x19
  _BYTE *v1; // x0
  __int64 v2; // x0
  __int64 v3; // x19
  __int64 v4; // x0
  signed __int64 v5; // x25
  __int64 v6; // x0
  __int64 v7; // x0
  _BYTE *v8; // x0
  _BYTE *v9; // x0
  baidu_enum_vm_type v10; // w20
  char v12; // [xsp+8h] [xbp-158h]
  __int64 v13; // [xsp+108h] [xbp-58h]

  v0 = sub_25FD0("4819CF8872FE27590B0F928A70A127");// /proc/self/maps
  v1 = sub_2607C("D6E4059A10DC32C3");           // "r"
  v2 = fopen(v0, v1);
  v3 = v2;
  if ( v2 )
  {
    if ( fgets(&v12, 256LL, v2) )
    {
      while ( 1 )
      {
        v4 = strrchr(&v12, '/');
        if ( v4 )
        {
          v5 = v4 + 1;
          v6 = sub_26128("6D946FAE0959BDD46E"); // libdvm.so
          if ( !(unsigned int)strncmp(v6, v5, 9LL) )
          {
            v10 = 1;
            goto LABEL_15;
          }
          v7 = sub_261D4("2C5643EAE095530D2F"); // libart.so
          if ( !(unsigned int)strncmp(v7, v5, 9LL) )
          {
            v10 = 2;
            goto LABEL_15;
          }
          v8 = sub_26280("080407955A6AF59B3B01008E4273B28C0B");// libvmkid_lemur.so
          if ( !(unsigned int)strncmp(v8, v5, 17LL) )
          {
            v10 = 3;
            goto LABEL_15;
          }
          v9 = sub_2632C("3C74B470455E09843F"); // libaoc.so
          if ( !(unsigned int)strncmp(v9, v5, 9LL) )
            break;
        }
        if ( !fgets(&v12, 256LL, v3) )
          goto LABEL_9;
      }
      v10 = 4;
    }
    else
    {
LABEL_9:
      v10 = 0;
    }
LABEL_15:
    LODWORD(v2) = fclose(v3);
  }
  else
  {
    v10 = 0;
  }
  if ( _stack_chk_guard == v13 )
    LODWORD(v2) = v10;
  return (signed int)v2;
```

至此，libbaiduprotect.so的加载流程就执行完了。

现在回到java层，调用com.baidu.protect.A.n001方法，通过前面分析可以，该方法对应的本地函数为sub_9318

```
public void attachBaseContext(Context context) {
    mContext = context;
    mBootStrapApplication = this;
    AppInfo.APKPATH = context.getApplicationInfo().sourceDir;
    AppInfo.DATAPATH = getDataFolder(context.getApplicationInfo(
    if (!Debug.isDebuggerConnected()) {
        loadLibrary();
        A.n001(AppInfo.PKGNAME, AppInfo.APPNAME, AppInfo.APKPATH
    }
    if (AppInfo.APPNAME != null && AppInfo.APPNAME.length() > 0)
        mRealApplication = MonkeyPatcher.createRealApplication(A
    }
    super.attachBaseContext(context);
    if (mRealApplication != null) {
        MonkeyPatcher.attachBaseContext(context, mRealApplicatic
    }
}
```

sub_9318中将参数保存起来，然后最主要就是调用了sub_781C，

```
__int64 __fastcall n001(JNIEnv *env, jclass a2, jstring PKGNAME, jstring APPNAME, jstring APK
{
  char *v8; // x0
  char *v9; // x0
  char *v10; // x0
  const char *v11; // x20
  __int64 v12; // x0
  baidu_struc_6 a1; // [xsp+18h] [xbp-278h]
  __int64 (__fastcall *v15)(__int64); // [xsp+28h] [xbp-268h]
  jboolean report_crash; // [xsp+33h] [xbp-25Dh]
  jint sdk_int; // [xsp+34h] [xbp-25Ch]
  jstring data_path; // [xsp+38h] [xbp-258h]
  jstring apk_path; // [xsp+40h] [xbp-250h]
  jobject app_name; // [xsp+48h] [xbp-248h]
  jstring pkg_name; // [xsp+50h] [xbp-240h]
  jclass v22; // [xsp+58h] [xbp-238h]
  JNIEnv *env_1; // [xsp+60h] [xbp-230h]
  char a2a; // [xsp+68h] [xbp-228h]

  v22 = a2;
  env_1 = env;
  app_name = APPNAME;
  pkg_name = PKGNAME;
  data_path = DATAPATH;
  apk_path = APKPATH;
  sdk_int = SDK_INT;
  report_crash = REPORT_CRASH;
  sdk_version = SDK_INT;
  b_REPORT_CRASH = REPORT_CRASH != 0;
  v8 = jstring_to_utf_chars(env, &a2a, (__int64)PKGNAME, 512LL);
  string_assign_0(&::PKGNAME, v8);
  if ( app_name )
    gref_APPNAME = NewGlobalRef(env_1, app_name);
  v9 = jstring_to_utf_chars(env_1, &a2a, (__int64)apk_path, 512LL);
  string_assign_0(&::APKPATH, v9);
  v10 = jstring_to_utf_chars(env_1, &a2a, (__int64)data_path, 512LL);
  string_assign_0(&::DATAPATH, v10);
  v11 = sub_25F24("44CB02463B5D3E0202D3");      // "%s/.bdlock"
  v12 = string_c_str(&::DATAPATH);
  snprintf(&a2a, 0x200u, v11, v12);             // "/data/user/0/com.example.test/.bdlock"
  v15 = sub_781C;
  open_and_lock_file(&a1, &a2a);
  v15((__int64)env_1);
  return flock_and_close(&a1);
}
```

sub_781C也是被混淆的，但是代码很少，稍微看下，就知道只做了一件事，就是用3作为参数调用函数队列中的所有函数。

```
__int64 __fastcall sub_781C(__int64 a1)
{
  __int64 v1; // x19
  __int64 v2; // x0
  __int64 v3; // x2
  __int64 v5; // x0
  __int64 v6; // x0
  __int64 v7; // x2
  __int64 v8; // x0
  __int64 v9; // x2
  __int64 v10; // [xsp+0h] [xbp-10h]
  __int64 vars0; // [xsp+10h] [xbp+0h]

  v1 = a1;
  while ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )
    ;
  if ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )
    goto LABEL_9;
  while ( 1 )
  {
    if ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )
      goto LABEL_13;
    while ( 1 )
    {
      *(&v10 - 2) = v1;
      v2 = sub_7BC4(a1);
      v3 = *(&v10 - 2);
      a1 = sub_3E628(v2, 3);
      if ( (dword_C0118 * (dword_C0118 - 1) & 1) == 0 || dword_C0120 < 10 )
        break;
LABEL_13:
      *(&v10 - 2) = v1;
      v6 = sub_7BC4(a1);
      v7 = *(&v10 - 2);
      a1 = sub_3E628(v6, 3);
    }
    if ( (dword_C0118 * (dword_C0118 - 1) & 1) == 0 || dword_C0120 < 10 )
      break;
LABEL_9:
    if ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )
      goto LABEL_15;
    while ( 1 )
    {
      vars0 = v1;
      v5 = sub_7BC4(a1);
      a1 = sub_3E628(v5, 3);
      if ( (dword_C0118 * (dword_C0118 - 1) & 1) == 0 || dword_C0120 < 10 )
        break;
LABEL_15:
      *(&v10 - 2) = v1;
      v8 = sub_7BC4(a1);
      v9 = *(&v10 - 2);
      a1 = sub_3E628(v8, 3);
    }
  }
  while ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )
    ;
  return 0LL;
}
```

通过分析，会有sub_3E29C、sub_40CF8、sub_3DFC4、sub_11F5C、sub_45964这几个函数执行。

先看sub_3E29C，这个函数之前执行过参数为2的部分，现在来看参数为3的部分。先执行sub_13880，通过分析，该函数是获取apk包的签名，然后计算签名的MD5。然后sub_66064将签名的MD5值进行扩展，变为176字节。然后将签名的MD5和扩展后的内容存放在qword_BE690。

```
if ( a2 == 3 )
{
  sub_13880(a3, *((__int64 *)&xmmword_BE5F0 + 1), (__int64)&v5);
  sub_66064(&qword_BE690, v3, &v5, 16LL);
}
```

再看sub_40CF8，该函数可以接受3和4作为参数，现在先看3，

调用sub_409E0，它先注册com.baidu.protect.CrashHandler的本地函数，然后调用init方法。然后用sigaction设置信号4、6、7、8、11的回调sub_40BD0。

```
  env_1 = env;
  if ( a2 == 4 )
  {
    if ( b_REPORT_CRASH && !(byte_BE800 & 1) )
    {
      v4 = str_class_name_com_baidu_protect_CrashHandler;
      v5 = (_BYTE *)sub_28D80("75FC229422DB36AF");// asynRun
      v6 = (_BYTE *)sub_28E2C("22460BBD4F767F20");// ()V
LABEL_9:
      v8 = v6;
      v9 = ((__int64 (__fastcall *)(JNIEnv *, __int64))(*env_1)->FindClass)(env_1, v4);
      v10 = ((__int64 (__fastcall *)(JNIEnv *, __int64, _BYTE *, _BYTE *))(*env_1)->GetStaticMethodID)(
              env_1,
              v9,
              v5,
              v8);
      if ( ((unsigned __int8 (__fastcall *)(JNIEnv *))(*env_1)->ExceptionCheck)(env_1) == 1 )
        ((void (__fastcall *)(JNIEnv *))(*env_1)->ExceptionClear)(env_1);
      else
        callStaticMethod_0((__int64)env_1, v9, v10, v11, v12, v13, v14, v15, v16, v17, v18, v19, v20);
      ((void (__fastcall *)(JNIEnv *, __int64))(*env_1)->DeleteLocalRef)(env_1, v9);
      return 1LL;
    }
  }
  else
  {
    if ( a2 != 3 )
      return 0LL;
    if ( b_REPORT_CRASH )
    {
      RegisterNatives_and_call_init_and_sig_call_back(a1, (__int64)env);
      read_uuid(env_1, v21);
      result = 1LL;
      if ( *(_DWORD *)&v21[80] < 0x12D )
        return result;
      byte_BE800 = 1;
      v4 = str_class_name_com_baidu_protect_CrashHandler;
      v5 = sub_28ED8("90E927A0D46BB8D0");       // asynRun
      v6 = sub_28F84("21F36BB163FCC4D6");       // ()V
      goto LABEL_9;
    }
  }
  return 1LL;
```

再看sub_3DFC4,这个函数最主要就是sub_3D6AC，通过解析apk中assets，生成各种路径，然后通过qword_BE2F0生成目录，qword_BE2F0就是之前从libc中获取的mkdir的指针。

```
signed __int64 __fastcall sub_3DFC4(__int64 a1, int a2, __int64 a3)
{
  __int64 v3; // x19
  __int64 v4; // x20
  int v5; // w23
  __int64 *v6; // x24
  __int64 v7; // t1
  __int64 v8; // x20

  v3 = a3;
  if ( a2 != 3 )
    return 0LL;
  v4 = qword_BE548;
  if ( !qword_BE548 )
  {
    v4 = sub_82254(81536LL);
    sub_3D1F4();
    qword_BE548 = v4;
  }
  dword_BE574 = 100;
  if ( sub_3D6AC(v4, v3) & 1 && *(_DWORD *)(v4 + 7300) >= 1 )
  {
    v5 = 0;
    v6 = (__int64 *)(v4 + 7440);
    do
    {
      v7 = *v6;
      v6 += 6;
      qword_BE2F0(v7, 448LL);
      ++v5;
    }
    while ( v5 < *(_DWORD *)(v4 + 7300) );
  }
  if ( dword_BE570 > 25 )
  {
    dword_BE574 = 101;
  }
  else
  {
    v8 = qword_BE548;
    if ( !qword_BE548 )
    {
      v8 = sub_82254(81536LL);
      sub_3D1F4();
      qword_BE548 = v8;
    }
    sub_3C2A4(v8, v3);
  }
  return 1LL;
}
```

再看sub_11F5C，该函数只是调用了sub_BC60，sub_BC60也被混淆了，删除垃圾代码后，流程如下。因为我用的手机是8.1的，所以只看了sdk大于26的，

```
__int64 __fastcall sub_BC60(__int64 a1)
{
    __int64 *v1; // x19
    __int64 result; // x0
    __int64 v3; // [xsp+0h] [xbp-10h]
    __int64 vars0; // [xsp+10h] [xbp+0h]
    //sdk_version=27
    v1 = &v3 - 2;
    *(&v3 - 2) = a1;

    if ( sdk_version >= 26 )
    {
        result = sub_188AC();//修改libart.so内存页属性，hook __android_log_print(nop)和mmap
    }
    else if ( sdk_version >= 21 )
    {
        result = sub_14468();
    }
    else
    {
        result = sub_1B284();
    }

    if ( sdk_version >= 26 )
    {
        result = sub_11AB0(*v1);//将assets下所有baiduprotect{n}.jar解密成dex，然后通过InMemo
    }
    else if(sdk_version >= 19)
    {
        result = sub_11490(*v1);
    }
    else if(sdk_version >= 9)
    {
        result = sub_11004(*v1);
    }
    else
    {
        result = sub_10A70(*v1);
    }

    if ( sdk_version >= 26 )
    {
        result = sub_1B154(result);//取消mmap的hook

    }
    else if(sdk_version >= 21 )
    {
        result = sub_18710(result);
    }
    else
    {
        result = sub_1B18C(result);
    }

    byte_BE200 = 1;
    return result;
}
```

先看sub_188AC，该函数也是被混淆的，删除垃圾代码后流程如下，

```
if ( vm_type_enum == 4 )
{
    *v0 = sub_25E78("5D21D4FC24249D425E");//libaoc.so
}
else if ( dword_BE570 < 29 )
{
    *v0 = (__int64)sub_25FD0("0B00DF8663A57A4F08");//libart.so
}
else
{
*v0 = (__int64)sub_25F24("0DD14F092B4D300C12DD031B36");//libartbase.so
}

v1 = (unsigned __int64)sub_4029C(algn_BE518, *v0) != 0;
if ( !v1 )
{
    v7 = sub_2607C("FBBB64F474AE5DAAC0BB69F5778342B1CD8A71");//__android_log_print
    sub_3FF9C(algn_BE518, v7, sub_1B044);
    v8 = sub_261D4("2D5240FB92E17D7E");//mmap
    result = sub_3FF9C(algn_BE518, v8, sub_1B070);
}
else
{
    dword_BE574 = 421;
    v2 = (void (__fastcall *)(void *, signed __int64, __int64, const char *, signed __
    v3 = sub_25E78("6A6DC5A76E32EE0B143B");
    v4 = (unsigned int *)__errno();
    v5 = strerror(*v4);
    v2(&unk_BE58C, 64LL, v3, "art26_hook_funcs", 86LL, v5);
    result = sub_407CC(421LL);
}
```

sub_4029C读取/proc/self/maps文件，查找对应的so，修改内存页属性

```
  vm_so_name_1 = vm_so_name;
  open = (__int64 (__fastcall *)(_BYTE *, _BYTE *))fopen_0;
  so_start_addr = so_start_addr_1;
  v5 = sub_25E78("1E38C4F22868C0545D2E99F02A37C0");// /proc/self/maps
  v6 = sub_25F24("13B82D685939526D");           // r
  v7 = open(v5, v6);
  if ( !v7 )
    return 0xFFFFFFFFLL;
  memset_0(&v27, 0LL, 64LL);
  v23 = sub_25FD0("1544909711D1543C");          // r--p
  v8 = sub_2607C("819725BF63FC17E9D7C420B063FC17E9D7C420E9");// %s %s %*s %*s %*s %s
  v9 = (const char *)sub_26128("73D075BA7F3493A7");// r-xp
  if ( !fgets_0(&v28, 400LL, v7) )
  {
    v21 = 0;
    v20 = 1;
    goto LABEL_19;
  }
  not_first = 0;
  while ( 1 )
  {
    if ( !strstr_0(&v28, vm_so_name_1) )        // 查找libart.so
      goto LABEL_6;
    sscanf(&v28, v8, &v27, &v25, &v26);         // 起讫地址、属性、偏移地址、主从设备号、inode编号 路径
    v10 = strchr_0(&v27, '-');                  // 7ef9dd2000-7efa3cc000 r-xp 00000000 103:0b 1553 /system/lib64/libart.so
    v11 = v10;
    if ( !v10 )
      goto LABEL_6;
    *v10 = 0;
    if ( !not_first )
    {
      *so_start_addr = strtoull(&v27, 0LL, 16);
      not_first = 1;
    }
    if ( strstr_0(&v28, v23) )
    {
      v12 = v11 + 1;
      segment_start_addr = strtoull(&v27, 0LL, 16);
      segment_end_addr = strtoull(v12, 0LL, 16);
      mprotect = (unsigned int (__fastcall *)(unsigned __int64, unsigned __int64, signed __int64))::mprotect;
      segment_len = segment_end_addr - segment_start_addr;
      mods = 3LL;
      goto LABEL_14;
    }
    if ( strstr_0(&v28, v9) )
      break;
LABEL_6:
    if ( !fgets_0(&v28, 400LL, v7) )
    {
      v20 = 1;
      goto LABEL_16;
    }
  }
  v18 = v11 + 1;
  segment_start_addr = strtoull(&v27, 0LL, 16);
  v19 = strtoull(v18, 0LL, 16);
  mprotect = (unsigned int (__fastcall *)(unsigned __int64, unsigned __int64, signed __int64))::mprotect;
  segment_len = v19 - segment_start_addr;
  mods = 7LL;
LABEL_14:
  if ( !mprotect(segment_start_addr, segment_len, mods) )
    goto LABEL_6;
  v20 = 0;
LABEL_16:
  v21 = not_first;
LABEL_19:
  fclose_0(v7);
  return (unsigned int)(((v21 == 0) | ~v20) << 31 >> 31);
}
```

sub_3FF9C是对函数进行hook，通过动态节，找到重定位节和got，然后替换指定标签的地址。这里分别hook了__android_log_print和mmap，__android_log_print被替换为sub_1B044，这是个空函数，禁用log。mmap被替换为sub_1B070，在加载dex的时候有用，稍后分析。

再看sub_11AB0，最主要的是下面的这个循环，将assets下所有的jar解密为dex，然后通过InMemoryDexClassLoader加载，然后提取DexPathList$Element添加到源classloader中。

```
                                        Ljava_nio_ByteBuffer_Ljava_lang_ClassLoader__V);
for ( i = 0; ; ++i )
{
  v50 = i;
  v51 = get_baidu_struct_1();
  if ( v50 >= (signed int)get_baiduprotect_jar_number(v51) )
    break;
  dex = 0LL;
  dex_size = 0LL;
  if ( (unsigned __int8)ExceptionCheck(env) == 1 )
    ExceptionClear(env);
  v52 = get_baidu_struct_1();
  decode_dex_i(v52, i, &dex, &dex_size);
  dex_buf = NewDirectByteBuffer(env, dex, dex_size);
  set_dex_size(dex_size);
  object_InMemoryDexClassLoader = NewObjectV(
                                    env,
                                    (__int64)class_InMemoryDexClassLoader,
                                    InMemoryDexClassLoader_Constructor,
                                    dex_buf,
                                    classLoader,
                                    v54,
                                    v55,
                                    v56,
                                    v57,
                                    v58,
                                    v59,
                                    v60,
                                    v66);
  merge_element(env, (__int64)obj_mClassLoader, object_InMemoryDexClassLoader);// 将Element合并后重新设置到原类加载器上
  dex_ptr = (char *)get_mmap_result__from_hook();
  baidu_struc_1 = get_baidu_struct_1();
  update_dex_info_to_baidu_struct_1(baidu_struc_1, i, dex_ptr, dex_size);
  DeleteLocalRef(env);
  DeleteLocalRef(env);
  free(dex);
}
```

对于每个assets/baiduprotect*.jar，它由以下几部分组成，

原dex （前0x1000字节加密） （中间某部分内容被抹掉） （class_data_off_被抹掉）	附加数据1 （用于恢复中间被抹掉的部分）	附加数据2 （用于恢复class_data_off_， 大小为所有class*4）	附加数据3 （用于执行vmp方法）	附加数据的头 （sizeof=0x118） +4 附加数据1的大小 +8 中间被抹掉内容的偏移 +12 附加数据3的大小

先看sub_3BA90，该函数用于解密dex，其中sub_9B2C用于获取apk包中的所有文件目录，sub_A104用于检查apk包中是否存在assets/baiduprotect*.jar，sub_A23C用于获取该文件的信息(压缩前后大小，时间，crc等)，sub_A60C用于将文件解压出来，sub_65980用之前的签名信息将前0x1000字节解密，sub_1C43C将附加数据1复制到附加数据头中指定的位置，用签名信息解密，再用附加数据2修复class_data_off_，然后重新计算校验和。

由此可知，如果重新打包，签名不正确的话就会解密失败。

dex脱壳：当sub_3BA90执行返回就可以dump解密的dex了，如果不想动态调试手动搞，也可以写个xposed模块，在后面一步InMemoryDexClassLoader加载dex的时候获取。

```
v48 = 0LL;
v7 = a2;
if ( (unsigned int)sub_9B2C(&v41, *(_QWORD *)(a1 + 19776), 0xFFFFFFFFLL) )// parse_zip_center_dir
{
  v8 = (void (__fastcall *)(void *, signed __int64, __int64, const char *, signed __int64, __int64
  dword_BE574 = 418;
  v9 = sub_25E78("6A6DC5A76E32EE0B143B");
  v10 = (unsigned int *)__errno();
  v11 = strerror(*v10);
  v8(&unk_BE58C, 64LL, v9, "GetDexData", 28LL, v11);
  sub_407CC(418LL);
}
else if ( *(_DWORD *)(v6 + 7300) > v7 )
{
  v12 = (void (__fastcall *)(char *, signed __int64, __int64, _QWORD))qword_BE320;
  v13 = sub_25E78("503BC5F83F349C535021D2E83B35DC45542BC2B82F69D95043");// assets/baiduprotect%d.j
  v12(&v49, 306LL, v13, (unsigned int)(v7 + 1));
  v14 = sub_A104(&v41, &v49);                   // lookup_string
  if ( v14 )
  {
    v40 = 0LL;
    if ( sub_A23C(&v41, v14, 0LL, &v40, 0LL, 0LL, 0LL, 0LL) & 1 )// get_compressed_file_info
    {
      v15 = qword_BE260(v40);
      if ( v15 )
      {
        if ( sub_A60C(&v41, v14, v15) & 1 )   // unzip_stream
        {
          if ( v40 >= 0x1000 )
            v16 = 4096LL;
          else
            v16 = (unsigned int)v40;
          sub_65980(&qword_BE690, v15, v16, v15);// use_signature_info_decode
          v17 = sub_1C43C(v15, v40, 0LL);
          if ( v17 )
          {
```

sub_18880保存当前dex的大小，后面加载dex的时候要用到。

sub_12100用InMemoryDexClassLoader类调用NewObjectV加载dex，InMemoryDexClassLoader内部会使用mmap分配内存存放dex，通过前面分析，我们知道mmap被hook替换为sub_1B070，所以现在来看下sub_1B070，可以看到，先将原始方法调用了一遍，然后检查是否为刚才加载dex所需的那块内存，是则将其保存。

```
__int64 __fastcall hook_mmap(__int64 a1, unsigned __int64 a2)
{
  unsigned __int64 v2; // x19
  __int64 result; // x0
  void (__fastcall *v4)(void *, signed __int64, _BYTE *, const char *, signed __int64, __int64); // x20
  _BYTE *v5; // x19
  unsigned int *v6; // x0
  __int64 v7; // x0

  v2 = a2;
  result = mmap_0(a1);
  if ( is_set_dex_size_wait_mmap_use == 1 && dex_size__hook_mmap_use <= v2 && STRING_MAX + dex_size__hook_mmap_use >= v2 )
  {
    if ( result == -1 )
    {
      v4 = (void (__fastcall *)(void *, signed __int64, _BYTE *, const char *, signed __int64, __int64))snprintf_0;
      init_0___sdk_greater_25_101 = 405;
      v5 = sub_25E78("6A6DC5A76E32EE0B143B");
      v6 = (unsigned int *)__errno();
      v7 = strerror(*v6);
      v4(&unk_BE58C, 64LL, v5, "art_mmap", 33LL, v7);
      sub_407CC(405LL);
      result = -1LL;
    }
    else
    {
      mmap_result = result;
      is_set_dex_size_wait_mmap_use = 0;
    }
  }
  return result;
}
```

回到刚才加载dex的调用之后，通过sub_1217C分配一个新的DexPathList$Element数组，将原来系统的类加载器和刚才的InMemoryDexClassLoader中的classLoader.pathList.dexElements合并成一个数组，然后替换原来系统中的类加载器的dexElements。sub_1889C获取刚才mmap的hook函数保存的dex地址。sub_3DDC8解析出dex中的各种数组指针

最后一个函数sub_45964，主要就两步。

```
signed __int64 __fastcall sub_45964(__int64 a1, int a2, __int64 a3)
{
  __int64 v3; // x19
  __int64 v4; // x20
  __int64 v5; // x21
  __int64 v6; // x0
  __int64 v7; // x22
  __int64 v8; // x0
  __int64 v9; // x4

  v3 = a3;
  if ( a2 != 3 )
    return 0LL;
  if ( !qword_BED10 )
  {
    a1 = sub_82254(1LL);
    qword_BED10 = a1;
  }
  sub_42C08(a1, v3, qword_BE790);
  if ( *(_DWORD *)(sub_3D48C() + 7300) >= 1 )
  {
    v4 = 0LL;
    do
    {
      v5 = qword_BED10;
      if ( !qword_BED10 )
      {
        v5 = sub_82254(1LL);
        qword_BED10 = v5;
      }
      v6 = sub_3D48C();
      if ( *(_DWORD *)(v6 + 7300) <= (signed int)v4 )
        v7 = 0LL;
      else
        v7 = *(_QWORD *)(v6 + 8 * v4);
      v8 = sub_3D48C();
      if ( *(_DWORD *)(v8 + 7300) <= (signed int)v4 )
        v9 = 0LL;
      else
        v9 = *(unsigned int *)(v8 + 4 * v4 + 2048);
      sub_42D8C(v5, v3, (unsigned int)v4++, v7, v9, 2LL, (unsigned int)dword_BE570);
    }
    while ( (signed int)v4 < *(_DWORD *)(sub_3D48C() + 7300) );
  }
  return 1LL;
}
```

第一步调用sub_42C08注册com.baidu.protect.A中剩余的本地函数，用于vmp代理，可以看到，一共10个代理，每个对应一个返回值类型。每个函数对应的本地函数都是一样的，均为sub_42598。

```
  v3 = a3;
  v4 = a2;
  v7 = sub_26530("D2290DB35DD3BB4E");           // V
  v8 = method_prototype_V;
  v9 = sub_42598;
  v10 = sub_265DC("678A1DB04AD649B6");          // Z
  v11 = method_prototype_Z;
  v12 = sub_42598;
  v13 = sub_26688("01107FC0CAD110A2");          // B
  v14 = method_prototype_B;
  v15 = sub_42598;
  v16 = sub_26734("69BE796F507CE666");          // C
  v17 = method_prototype_C;
  v18 = sub_42598;
  v19 = sub_267E0("900C48443A5C67C1");          // S
  v20 = method_prototype_S;
  v21 = sub_42598;
  v22 = sub_2688C("552D54E3921DA596");          // I
  v23 = method_prototype_I;
  v24 = sub_42598;
  v25 = sub_26938("5EF11AA0771CD9F4");          // J
  v26 = method_prototype_J;
  v27 = sub_42598;
  v28 = sub_269E4("7640876B378F2B6B");          // F
  v29 = method_prototype_F;
  v30 = sub_42598;
  v31 = sub_26A90("02B1F3133EB607CE");          // D
  v32 = method_prototype_D;
  v33 = sub_42598;
  v34 = sub_26B3C("669B877B16770DB8");          // L
  v35 = method_prototype_Object;
  v36 = sub_42598;
  v5 = ((__int64 (__fastcall *)(JNIEnv *, __int64))(*v4)->FindClass)(v4, v3);
  if ( v5 )
    result = ((signed int (__fastcall *)(JNIEnv *, __int64, __int64 *, signed __int64))(*v4)->RegisterNatives)(
               v4,
               v5,
               &v7,
               10LL) >= 0;
  else
    result = 0LL;
  return result;
}
```

第二步就是对每个dex调用sub_42D8C，去解析附加数据3，通过分析，其数据结构如下，其中有用的字段为方法数组和指令替换表。

```
00000000
00000000 baidu_struc_12_append_data_3 struc ; (sizeof=0x153, mappedto_181)
00000000 field_0         DCB 6 dup(?)            ; string(C)
00000006 field_6         DCD ?
0000000A string_array_size DCD ?                 ; 字符串的个数
0000000E string_start_offset DCD ?               ; field_string_array字段相对于field_0的偏移
00000012 field_18        DCD ?
00000016 field_22        DCD ?
0000001A method_array_size DCD ?                 ; 方法的个数
0000001E method_start_offset DCD ?               ; method数组相对于field_0的偏移
00000022 op_map_start_offset DCD ?               ; 指令替换表相对于field_0的偏移
00000026 string_array    baidu_struc_11_string2 ? ; 字符串数组，前4字节为长度，后面紧跟着字符串
00000036 method_array    baidu_struc_12_append_data_3_method ? ; method数组
00000053 op_map_array    DCB 256 dup(?)          ; 指令替换表,当前表为it、函数初始表为ft1，正真使用的表ft2，生成ft2[it[i]]=ft1[
00000153 baidu_struc_12_append_data_3 ends
00000153
```

其中方法的数据结构如下，

```
00000000
00000000 baidu_struc_12_append_data_3_method struc ; (sizeof=0x1D, mappedto_183)
00000000                                         ; XREF: baidu_struc_12_append_data_3/r
00000000 field_1         DCD ?
00000004 field_2         DCW ?
00000006 field_3         DCD ?
0000000A field_4         DCW ?
0000000C outs_size       DCW ?
0000000E ins_size        DCW ?
00000010 registers_size  DCW ?
00000012 s_idx           DCW ?
00000014 insns_size      DCW ?                   ; 指令的条数
00000016 insns           DCW ?                   ; 一个数组，指令
00000018 tries_size      DCW ?                   ; try数量
0000001A try_catch_array_len DCW ?               ; 异常部分的长度
0000001C try_catch_array DCB ?                   ; 一个数组，异常部分的所有内容
0000001D baidu_struc_12_append_data_3_method ends
0000001D
```

至此，com.baidu.protect.A.n001方法的调用过程就分析完了。

现在，回到attachBaseContext，剩下的就只是替换程序原来的application了。

然后就是onCreate，它调用了com.baidu.protect.A.n002

```
public void onCreate() {
    patchApplication();
    super.onCreate();
    this.mOnCreateFinished = true;
    if (AppInfo.RELOAD_SP && Build.VERS
        reloadSP();
    }
    if (mRealApplication != null) {
        mRealApplication.onCreate();
    }
    A.n002(this);
}
```

通过前面分析可以，该方法对应的本地函数为sub_94E4，该函数主要就是用4作为参数调用函数队列中的所有函数。

```
1  __int64 __fastcall sub_94E4(__int64 a1, __int64 a2, __int64 a3)
2  {
3    __int64 v3; // x0
4    __int64 result; // x0
5    __int64 v5; // [xsp+8h] [xbp-28h]
6    __int64 v6; // [xsp+10h] [xbp-20h]
7    __int64 v7; // [xsp+18h] [xbp-18h]
8    __int64 v8; // [xsp+20h] [xbp-10h]
9    __int64 v9; // [xsp+28h] [xbp-8h]
10
11   v8 = a2;
12   v9 = a1;
13   v7 = a3;
14   v3 = sub_7BC4(a1);
15   sub_3E628(v3, 4);
16   result = gettimeofday(&v5, 0LL);
17   qword_BE220 = 1000000 * (v5 - qword_BE208) + v6 - qword_BE210;
18   return result;
19 }
```

通过分析，会有sub_ 40CF8、sub_ 3E96C、sub_ 42388这几个函数执行。这部分就不详细写了，通过分析知sub_ 40CF8调用CrashHandler.asynRun方法，向https://apkprotect.baidu.com/apklog发送统计信息。sub_ 3E96C assets/baiduprotect.m检查dex的完整性，该文件中存有加密的dex MD5。sub_ 42388注册com.baidu.xshield.jni.Asc和com.baidu.xshield.utility.KeyUtil的本地函数，调用com.baidu.xshield.ac.XH.init方法。

接下来看看dump出来的dex文件。可以看到onCreate方法被改了，调用了没有返回值那个代理函数。推测0xAB000000是其id，暂且称为vmp_method_id

```
    }

    public MainActivity() {
        super();
    }

    protected void onCreate(Bundle arg5) {
        A.V(0xAB000000, this, new Object[]{arg5});
    }

    public static native String[] parseClassNames(byte[] arg0) {
    }

    public native String stringFromJNI() {
    }
}
```

由前面分析可知，所有代理绑定的本地函数都为sub_42598，该函数也被混淆了，删除垃圾代码后如下，由此可知，vmp_method_id值的最高字节没有用，第16至24位为dex编号，通过该值找到对应dex解析后的信息。

```
sub_4A458(baidu_struc_14_array[(m_id & 0xFF0000u) >> 16], m_id&&0xffff, env, obj, args)
```

sub_4A458通过分析，可知，vmp_method_id的低16位为附加数据3中method的索引，通过该id找到对应的method结构，然后分配寄存器空间。将参数值放入寄存器中，然后检查指令对应的函数数组有没有初始化，没有初始化则通过附加数据3最后的指令替换表，将原始的指令数组映射。然后开始通过解释器执行指令。

随便找个vmp化后的方法指令，然后和未加固前的指令对比，可以看出只是将指令第一个字节替换了(第一个字节表示哪条指令，后面的都是操作数)。还有就是有些id因为重新编译后变了。所以我们只需要把指令根据替换表再改回来就行了。

```
加固前的指令
00    6f 00 e7 0d 21 00       invoke-super        AppCompatActivity->onCreate(Bundle)V, p0, p1
06    14 02 1c 00 0a 7f       const               p1, 0x7F0A001C
0c    6e 00 51 3b 21 00       invoke-virtual      MainActivity->setContentView(I)V, p0, p1
12    14 02 7e 00 07 7f       const               p1, 0x7F07007E
18    6e 00 4f 3b 21 00       invoke-virtual      MainActivity->findViewById(I)View, p0, p1
1e    0c 02                   move-result-object  p1
20    1f 02 00 02             check-cast          p1, TextView
24    6e 10 52 3b 01 00       invoke-virtual      MainActivity->stringFromJNI()String, p0
2a    0c 00                   move-result-object  v0
2c    6e 00 fe 0b 02 00       invoke-virtual      TextView->setText(CharSequence)V, p1, v0
32    0e 00                   return-void

vmp指令
00    B9 20 02 00 21 00
06    B3 02 1C 00 0A 7F
0c    B0 20 0A 00 21 00
12    B3 02 7E 00 07 7F
18    B0 20 08 00 21 00
1e    24 02
20    79 02 03 00
24    B0 10 0B 00 01 00
2a    24 00
2c    B0 20 00 00 02 00
32    57 00
```

指令替换函数未混淆前的垃圾代码如下

```
while ( 1 )
{
    if ( i >= 256 )
    {
        *(_BYTE *)v3 = 1;
        goto LABEL_51;
    }
    v4->func_array[v6->op_map_array[i]] = ori_func_array[i];
    ++*(_DWORD *)v5;
}
```

修复vmp方法

思路:

1.将附加数据3解析出来，构造成DexCode添加到dex文件的后面，然后将class_data中的code_off修改为新构造的DexCode。

2.因为code_off是uleb128类型的值，所以新的值和旧的值占用空间可能不一样，所以当空间占用相同的情况下，则在原来的地方直接修改，不相同的话还得重新构造一个class_data放在所有添加的DexCode之后，然后将class_def中的class_data_off更新。

3.如何判断dex中的方法和附加数据中方法的关系？通过dex中方法id（暂且称为dex_method_id）和调用vmp代理时使用的vmp_method_id进行关联，如下图所示。

4.当method_ids_map为空的时候，修复程序将所有方法和它的id输出到文件，然后手动在文件中去找到对应方法dex_method_id，添加到method_ids_map中，再次运行修复程序就会将map所有指定的方法修复

```
std::unordered_map<int, int> method_ids_map;
method_ids_map[0x00001879] = 0xAB000000;
method_ids_map[0x000018a4] = 0xAB000001;
method_ids_map[0x000018c2] = 0xAB000002;
method_ids_map[0x0000192d] = 0xAB000003;
method_ids_map[0x00001943] = 0xAB000004;
method_ids_map[0x0000194e] = 0xAB000005;
method_ids_map[0x0000195e] = 0xAB000006;
```

修复过程:

首先将dex读取进内存

```
std::unique_ptr<unsigned char[]> read_file(std::string &str_file_path, int &file_len)
{
    std::ifstream ifs(str_file_path, std::ifstream::binary);

    ifs.seekg(0, ifs.end);
    file_len = static_cast<int>(ifs.tellg());
    unsigned char* np_buf = new unsigned char[file_len];

    ifs.seekg(0, ifs.beg);
    ifs.read(reinterpret_cast<char *>(np_buf), file_len);

    std::unique_ptr<unsigned char[]> up_buf(np_buf);
    ifs.close();

    return up_buf;
}
```

当map为空的时候，将文件添加个后缀，然后将所有方法信息写入。

```
if (method_ids_map.size() == 0)
{
    std::ofstream ofs(str_dex_path + ".methods.txt");
    parse_class_methods(dex_buf.get(), ofs);
    ofs.close();
}
```

```
5 Lcom/example/test/MainActivity;
   direct_method_id:0x0000001f, code_off:0x0001c294 | <clinit>()V
   direct_method_id:0x00000020, code_off:0x0001c2b0 | <init>()V
   direct_method_id:0x00000024, code_off:0x00000000 | parseClassNames([B)[Ljava/lang/String;
   virtual_method_id:0x00000023, code_off:0x0001c2c8 | onCreate(Landroid/os/Bundle;)V
   virtual_method_id:0x00000026, code_off:0x00000000 | stringFromJNI()Ljava/lang/String;
6 Lcom/example/test/MemberUtils;
```

```
std::unordered_map<int, int> method_ids_map;
method_ids_map[0x00000023] = 0xAB000000;
```

再次执行，现在因为map不为空，开始修复。

首先，解析附加数据3构造出所有的DexCode。

```
//构造DexCode列表
std::vector<std::pair<int, std::unique_ptr<char[]>>>  parse_append_data(unsigned char* dex_buf, int dex_len)
{
    std::vector<std::pair<int, std::unique_ptr<char[]>>> v_dex_code;//first为当前code的起始偏移

    append_data_header* append_data_header_ptr = reinterpret_cast<append_data_header*>(dex_buf + dex_len - 0x118);
    unsigned char* method_info_ptr = reinterpret_cast<unsigned char*>(append_data_header_ptr) - append_data_header_ptr->size_2;
    method_info_header* method_info_header_ptr = reinterpret_cast<method_info_header*>(method_info_ptr);

    fill_op(method_info_ptr + method_info_header_ptr->op_replace_offset);//先用附加数据3中的指令替换表构造用于替换指令的数组
    dexCreateInstrWidthTable();//生成一个数组，含有每条指令的宽度

    baidu_code_item* baidu_code_item_ptr = reinterpret_cast<baidu_code_item*>(method_info_ptr + method_info_header_ptr->method_code_item_offset);
    auto dex_code_len = 0;
    for (int method_index = 0; method_index < method_info_header_ptr->method_num; method_index++)//遍历附加数据3中的方法
    {
        unsigned short *insns = reinterpret_cast<unsigned short*>(baidu_code_item_ptr + 1);
        int width, insns_index = 0;
        for (; insns_index < baidu_code_item_ptr->insns_size; insns_index += width)//遍历每一条指令
        {
            width = dexGetInstrOrTableWidthAbs(insns+ insns_index);//计算每条指令的同时，会通过替换表将指令替换回来
        }

        int tries_size = insns[insns_index];
        int tries_len = insns[insns_index+1];

        unsigned char* tries = reinterpret_cast<unsigned char* >(insns + insns_index + 2);
        int pad_len = baidu_code_item_ptr->insns_size % 2 ? 1 : 0;//附加数据中的指令是没有填充对齐的，需要自己计算下
        auto code_len = 0x10 + (baidu_code_item_ptr->insns_size+ pad_len) * 2 + tries_len;
        code_len = (code_len + 3) / 4 * 4;//添加的code是挨个存放的，为了方便计算，每个code长度都4字节对齐
        char*np_dex_code = new char[code_len];
        v_dex_code.push_back(std::make_pair(dex_code_len, std::unique_ptr<char[]>(np_dex_code)));
        dex_code_len += code_len;

        DexCode* code_ptr = reinterpret_cast<DexCode*>(np_dex_code);
        code_ptr->registersSize = baidu_code_item_ptr->registers_size;
        code_ptr->insSize = baidu_code_item_ptr->ins_size;
        code_ptr->outsSize= baidu_code_item_ptr->outs_size;
        code_ptr->triesSize= tries_size;
        code_ptr->debugInfoOff= 0;
        code_ptr->insnsSize = baidu_code_item_ptr->insns_size;
        auto tries_ptr =std::copy_n(insns, baidu_code_item_ptr->insns_size, &code_ptr->insns[0]);//复制指令
        if (pad_len)//指令长度对齐
        {
            *(tries_ptr++) = 0;
        }
        std::copy_n(tries, tries_len, reinterpret_cast<unsigned char*>(tries_ptr));//异常相关数据

        baidu_code_item_ptr = reinterpret_cast<baidu_code_item*>(tries+tries_len);//下一个方法
    }
    v_dex_code.push_back(std::make_pair(dex_code_len, nullptr));//最后添加一个空的，用于计算所有code的长度

    return v_dex_code;
}
```

然后遍历class的方法，修复code_off

```
std::vector<std::pair<int, std::unique_ptr<unsigned char[]>>> restore_method_code(unsigned char* dex_buf, int dex_len, std::unordered_map<int, i
{
    dex_len = (dex_len + 3) / 4 * 4;//原dex对齐，这样好计算偏移
    auto code_len = v_dex_code.back().first;//获取到构造的code的总长度

    DexHeader* dex_header_ptr = reinterpret_cast<DexHeader*>(dex_buf);
    DexStringId* string_ids = reinterpret_cast<DexStringId*>(dex_buf + dex_header_ptr->stringIdsOff);
    DexTypeId* type_ids = reinterpret_cast<DexTypeId*>(dex_buf + dex_header_ptr->typeIdsOff);
    DexProtoId* proto_ids = reinterpret_cast<DexProtoId*>(dex_buf + dex_header_ptr->protoIdsOff);
    DexMethodId* method_ids = reinterpret_cast<DexMethodId*>(dex_buf + dex_header_ptr->methodIdsOff);
    DexClassDef* class_defs = reinterpret_cast<DexClassDef*>(dex_buf + dex_header_ptr->classDefsOff);
    auto class_defs_size = dex_header_ptr->classDefsSize;

    std::unordered_set<int> type_ids_set;
    for (auto &method_id: method_ids_map)//遍历所有要修复的方法，找出他们所属的class缓存起来，后面遍历class中方法的时候，没有的直接跳过
    {
        type_ids_set.insert(method_ids[method_id.first].classIdx);
    }


    std::vector<std::pair<int, std::unique_ptr<unsigned char[]>>> v_dex_class_data;
    auto tot_new_class_data_len = 0;

    for (u4 class_index = 0; class_index < class_defs_size; class_index++)
    {
        if (type_ids_set.find(class_defs[class_index].classIdx)==type_ids_set.end())//没有的跳过
        {
            continue;
        }

        auto class_data_off = class_defs[class_index].classDataOff;
        if (class_data_off == 0)
        {
            continue;
        }

        unsigned char* class_data_ptr = dex_buf + class_data_off;
        DexClassDataHeader st_class_data_header;
        dexReadClassDataHeader(const_cast<const u1 **>(&class_data_ptr), &st_class_data_header);
        //跳过属性字段
        for (size_t i = 0; i < st_class_data_header.staticFieldsSize + st_class_data_header.instanceFieldsSize; i++)
        {
            readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));
            readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));
        }

        auto methods_start_ptr = class_data_ptr;//保存下方法起始的位置，如果要重新构造class_data的时候用得着

        auto parse_methods_array = [&](u4 methods_size)->std::pair<bool,int> {
            auto tot_diff = 0;
            auto realloc = false;
            auto method_id = 0;
            for (size_t methods_index = 0; methods_index < methods_size; methods_index++)
            {
                method_id += readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));
                readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));
                auto temp = class_data_ptr;//保存下code偏移开始的地方，重写偏移的时候用得着
                auto code_off = readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));

                if (method_ids_map.find(method_id)!= method_ids_map.end())
                {
                    auto new_code_off = dex_len + v_dex_code[method_ids_map[method_id] & 0xffff].first;//id只有低16位有用
                    auto diff = unsignedLeb128Size(new_code_off) - (class_data_ptr - temp);
                    if (diff==0)//长度相同，直接重写
                    {
                        writeUnsignedLeb128(temp, new_code_off);
                    }
                    else
                    {
                        realloc = true;//不相同
                        tot_diff += diff;//保存总的相差几字节
                    }

                }
            }
            return std::make_pair(realloc,tot_diff);
        };
        //分别遍历，因为方法起始id是分别计算的
        auto diff_1=parse_methods_array( st_class_data_header.directMethodsSize);
        auto diff_2 = parse_methods_array( st_class_data_header.virtualMethodsSize);
```

```
        if (!diff_1.first && !diff_2 .first)//检查是否需要重新构造class_data
        {
            continue;
        }

        //需要重新构造
        class_defs[class_index].classDataOff = dex_len + code_len + tot_new_class_data_len;//放在所有dexcode后面,
        auto new_class_data_len = class_data_ptr - (dex_buf + class_data_off) + diff_1.second + diff_2.second;
        unsigned char*np_new_class_data= new unsigned char[new_class_data_len];
        v_dex_class_data.push_back(std::make_pair(tot_new_class_data_len, std::unique_ptr<unsigned char[]>(np_new_class_data)));
        tot_new_class_data_len += new_class_data_len;

        //直接复制方法之前的内容
        np_new_class_data = std::copy(dex_buf + class_data_off, methods_start_ptr, np_new_class_data);
        class_data_ptr = methods_start_ptr;

        auto cp_methods_array = [&](u4 methods_size)->void {
            auto tot_diff = 0;
            auto realloc = false;
            auto method_id = 0;
            for (size_t methods_index = 0; methods_index < methods_size; methods_index++)//挨个复制
            {
                auto method_id_diff = readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));
                method_id += method_id_diff;
                auto access_flags =readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));
                auto code_off = readUnsignedLeb128(const_cast<const u1 **>(&class_data_ptr));

                np_new_class_data = writeUnsignedLeb128(np_new_class_data, method_id_diff);
                np_new_class_data = writeUnsignedLeb128(np_new_class_data, access_flags);
                if (method_ids_map.find(method_id) != method_ids_map.end())
                {
                    auto new_code_off = dex_len + v_dex_code[method_ids_map[method_id] & 0xffff].first;//计算新偏移
                    np_new_class_data = writeUnsignedLeb128(np_new_class_data, new_code_off);
                }
                else
                {
                    np_new_class_data = writeUnsignedLeb128(np_new_class_data, code_off);
                }
            }
        };

        cp_methods_array(st_class_data_header.directMethodsSize);
        cp_methods_array(st_class_data_header.virtualMethodsSize);
    }
    v_dex_class_data.push_back(std::make_pair(tot_new_class_data_len, nullptr));

    return v_dex_class_data;
}
```

将修复后的内容写入文件。

```
void write_new_dex(std::ofstream &ofs, unsigned char* dex_buf, int dex_len, std::vector<std::pair<int, std::unique_ptr<char[]>>> &v_dex_code, std::v
{
    auto align_dex_len = (dex_len + 3) / 4 * 4;
    auto code_len = v_dex_code.back().first;
    auto new_class_data_len = v_dex_class_data.back().first;
    reinterpret_cast<DexHeader*>(dex_buf)->fileSize = align_dex_len+ code_len + new_class_data_len;

    //checksum和signature不影响反编译就不修复了

    ofs.write(reinterpret_cast<char*>(dex_buf), dex_len);//写入原dex
    for (int i = 0; i < align_dex_len - dex_len; i++)//添加对齐字节
    {
        ofs.put(0);
    }

    //写入所有修复的DexCode
    for (size_t i = 0; i < v_dex_code.size()-1; i++)
    {
        ofs.write(v_dex_code[i].second.get(), v_dex_code[i+1].first- v_dex_code[i].first);
    }
    //写入所有重新构造的class_data
    for (size_t i = 0; i < v_dex_class_data.size() - 1; i++)
    {
        ofs.write(reinterpret_cast<char*>(v_dex_class_data[i].second.get()), v_dex_class_data[i + 1].first - v_dex_class_data[i].first);
    }

}
```

修复完成后，将dex反编译，可以看到已经能够看到原来的代码了。

```java
/* access modifiers changed from: protected */
public void onCreate(Bundle bundle) {
    MainActivity.super.onCreate(bundle);
    setContentView(R.layout.activity_main);
    String stringFromJNI = stringFromJNI();
    ((TextView) findViewById(R.id.sample_text)).setText(stringFromJNI);
    try {
        for (Signature byteArray : getPackageManager().getPackageInfo(BuildConfig.APPLICATION_ID, 64).signatures) {
            byteArray.toByteArray();
            Log.v(logTag, Class.forName("[Ljava.lang.String;").toString());
        }
        switch (stringFromJNI.length()) {
            case 0:
                Log.v(logTag, "test_switch_0");
                return;
            case 1:
                Log.v(logTag, "test_switch_1");
                return;
            case 2:
                Log.v(logTag, "test_switch_2");
                return;
            case 3:
                Log.v(logTag, "test_switch_3");
                return;
            case 4:
                Log.v(logTag, "test_switch_4");
                return;
            case 5:
                Log.v(logTag, "test_switch_5");
                return;
            case 6:
                Log.v(logTag, "test_switch_6");
                return;
            case 7:
                Log.v(logTag, "test_switch_7");
                return;
            case 8:
                Log.v(logTag, "test_switch_8");
                return;
            case 9:
                Log.v(logTag, "test_switch_9");
                return;
            case 10:
                Log.v(logTag, "test_switch_10");
                return;
            default:
                Log.v(logTag, "test");
                return;
        }
    } catch (Throwable th) {
        th.printStackTrace();
        Log.v(logTag, "ex:" + th);
    }
}
```

总结

1.直接在InMemoryDexClassLoader构造函数处获取dex。

2.反编译获取到的dex，找到所有vmp方法的vmp_method_id，如下图所示。

```java
protected void onCreate(Bundle arg5) {
    A.V(0xAB000000, this, new Object[]{arg5});
}
```

3.运行修复程序，将str_dex_path改为要修复的文件路径，method_ids_map内容置空，如下图所示。

```
//char * dex_path = "C:\\Users\\lll19\\Desktop\\jiagu\\com.example.test_release_v2.0\\baidu_v2\\140310.dex";
std::string str_dex_path = "C:\\Users\\lll19\\Desktop\\jiagu\\com.example.test_release_v3.0\\baidu_v3\\140565.dex";
//std::string str_dex_path = "C:\\Users\\lll19\\Desktop\\baidu_9zhi\\1609483.dex";

std::unordered_map<int, int> method_ids_map;
//method_ids_map[0x00000023] = 0xAB000000;
```

4.第一次运行完后，在dex同目录下，有一个同名.methods.txt后缀的文件，打开找到对应方法的dex_method_id，如下图所示 。

```
Lcom/example/test/MainActivity;
 direct_method_id:0x0000001f, code_off:0x0001c294 | <clinit>()V
 direct_method_id:0x00000020, code_off:0x0001c2b0 | <init>()V
 direct_method_id:0x00000024, code_off:0x00000000 | parseClassNames([B)[Ljava/lang/String;
 virtual_method_id:0x00000023, code_off:0x0001c2c8 | onCreate(Landroid/os/Bundle;)V
```

5.第二次运行修复程序，将方法对应的两个id添加到method_ids_map，如下图所示。注意顺序不要搞反了。

```
std::unordered_map<int, int> method_ids_map;
method_ids_map[0x00000023] = 0xAB000000;
```

6. 第二次运行完毕后，在dex同目录下，有一个同名.new.dex后缀的文件，反编译就能看到修复后的代码了。

说明

本文的数据结构和修复程序只对当前分析的版本有效。

附件

分析所用的apk和修复代码【baiduprotect.zip】

[招生]科锐逆向工程师培训(3月6日远程教学报名特惠, 第37期)

最后于 22小时前 被卧勒个槽编辑 ，原因： 附件

上传的附件：

baiduprotect.zip （5.82MB，63次下载）

19 收藏 | 8 赞 | 打赏 | 分享

最新回复 (20)

D-t 1天前 2 楼 0

不错

挤蹭菌衣 1天前 3 楼 0

支持楼主 学习了

D-t 1天前 4 楼 0

最后于 1天前 被D-t编辑 ，原因:

loveqiao 1天前 5 楼 0

牛

最新回复 (20)

tDasm 1天前 6 楼 0

学习。百度加固也vmp了。

Debug.isDebuggerConnected 怎么过？反编译修改然后再编译打包？

最后于 1天前 被tDasm编辑，原因:

你咋不上天呢 1天前 7 楼 0

学习了，感谢楼主

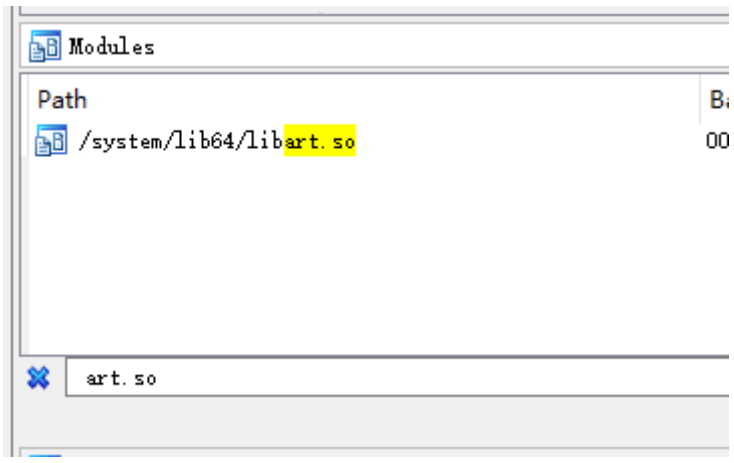
卧勒个槽 1 22小时前 8 楼 0

> tDasm 学习。百度加固也vmp了。 Debug.isDebuggerConnected 怎么过？反编译修改然后再编译打包？

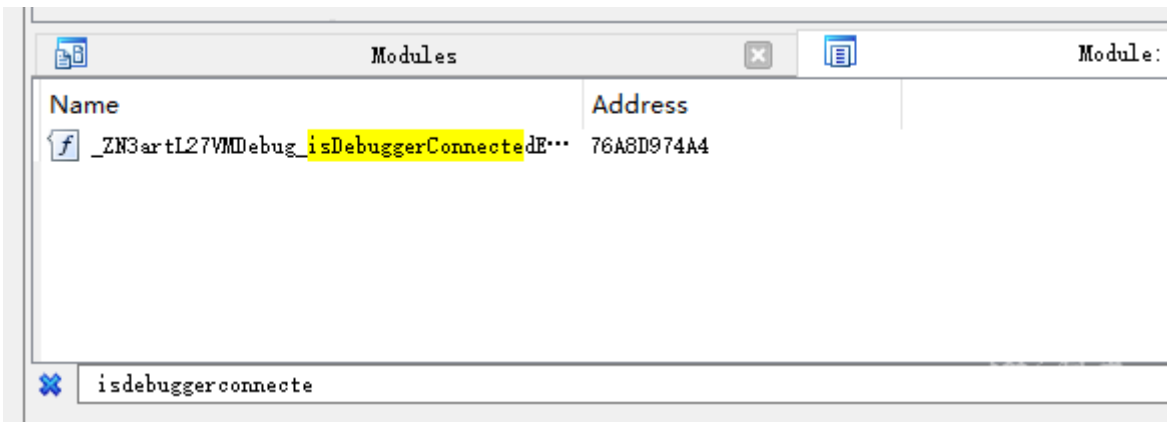
我是动态调试的，以调试模式启动app，

```
PS C:\Users\lllll9> adb shell am start -D -n com.example.test/.MainActivity
Starting: Intent { cmp=com.example.test/.MainActivity }
PS C:\Users\lllll9> adb forward tcp:23946 tcp:23946
```

ida附加上进程后，在modules窗口搜art.so，



双击找到的模块，在新窗口搜索isDebuggerConnected，



双击找到的函数，在返回处下断，

```
libart.so:00000076A8D974A4
libart.so:00000076A8D974A4 ; =============== S U B R O U T I N E =======================================
libart.so:00000076A8D974A4
libart.so:00000076A8D974A4
libart.so:00000076A8D974A4 _ZN3artL27VMDebug_isDebuggerConnectedEP7_JNIEnvP7_jclass
libart.so:00000076A8D974A4 ADRP            X8, #off_76A8FEF258@PAGE
libart.so:00000076A8D974A8 LDR             X8, [X8,#off_76A8FEF258@PAGEOFF]
libart.so:00000076A8D974AC LDRB            W0, [X8]
libart.so:00000076A8D974B0 RET
libart.so:00000076A8D974B0 ; End of function _ZN3artL27VMDebug_isDebuggerConnectedEP7_JNIEnvP7_jclass
libart.so:00000076A8D974B0
libart.so:00000076A8D974B0 ; ---------------------------------------------------------------------------
```

程序运行后会断在这里，查看返回值x0，当第二次返回1的时候将其改为0就行了。

上海刘一刀 2 21小时前 9 楼 0

大佬晚上好 文章中多次提到的 "把垃圾代码删除" 这个干掉 ollvm控制流平坦化的过程或者可以详细说一下。。。大佬写一下的话感觉又能多一个精华文章哪

最新回复 (20)

my1988 1 21小时前 10 楼 0

做个猜测，楼主是将混淆的代码f5之后直接复制到c++工程里面编译调试，通过这种方式删掉混淆的代码吗?

卧勒个槽 1 19小时前 11 楼 1

> 上海刘一刀 大佬晚上好 文章中多次提到的 "把垃圾代码删除" 这个干掉 ollvm控制流平坦化的过程或者可以详细说一下。。。大佬写一下的话感觉又能多一个精华文章哪

混淆后的代码，有很多判断条件是永远为真或永远为假的。
就像下面这个，一个变量x不需要知道它的值是多少，x*(x-1)永远为偶数，最后一位永远为0，将其和1按位与之后永远为0。
然后根据这些条件就能删掉很大一部分代码了。

```
if ( !v8 )
{
  while ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )/*false*/
    ;
  v17 = v120;
  if ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )/*false*/
    goto LABEL_290;
  while ( 1 )
  {
    if ( (dword_C0118 * (dword_C0118 - 1) & 1) != 0 && dword_C0120 >= 10 )/*false*/
      goto LABEL_403;
    while ( 1 )
    {
      *(_DWORD *)v9 = (_DWORD)v96;
      *(_DWORD *)v10 = 0;
      v18 = **(_QWORD **)v7 != 0LL;
      if ( (dword_C0118 * (dword_C0118 - 1) & 1) == 0 || dword_C0120 < 10 )/*true*/
        break;
LABEL_403:
      *(_DWORD *)v9 = (_DWORD)v96;
      *(_DWORD *)v10 = 0;
    }
    if ( (dword_C0118 * (dword_C0118 - 1) & 1) == 0 || dword_C0120 < 10 )/*true*/
    {
      v96 = &v95;
```

卧勒个槽 1 19小时前 12 楼 1

> my1988 做个猜测，楼主是将混淆的代码f5之后直接复制到c++工程里面编译调试，通过这种方式删掉混淆的代码吗?

当初还在实习的时候，第一次遇到混淆就是这么干的，一个函数五千行伪代码，调了一两天，😂

0x指纹 19小时前 13 楼 0

膜膜膜膜

my1988 1 19小时前 14 楼 0

> 卧勒个槽 当初还在实习的时候，第一次遇到混淆就是这么干的，一个函数五千行伪代码，调了一两天，[em_78]

我感觉这个方法挺实用的👍

上海刘一刀 2 18小时前 15 楼 0

> 卧勒个槽 混淆后的代码，有很多判断条件是永远为真或永远为假的。就像下面这个，一个变量x不需要知道它的值是多少，x*(x-1)永远为偶数，最后一位永远为0，将其和1按位与之后永远为0。然后根据这些条件就能删掉很大 ...

受教了

最新回复 (20)

pureGavin 18小时前

16 楼 0

mark，大家辛苦了

comewhere 1 13小时前

17 楼 0

mark 感谢分享

lyghost 10小时前

18 楼 0

膜拜大佬，分析的很详细

feikele 3小时前

19 楼 0

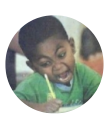
good

tDasm 2小时前

20 楼 0

谢谢。

如果用你修复的libbaiduprotect.so直接替换原来的so，程序能否正常运行?



卧勒个槽 1 1分钟前

21 楼 0

> tDasm 谢谢。 如果用你修复的libbaiduprotect.so直接替换原来的so，程序能否正常运行?

不能吧，

roysue

内容

回帖 表情 高级回复

删除 移动 精华 置顶 审核 关闭

返回