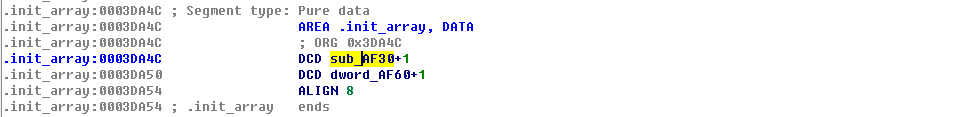
本人菜鸟一枚，刚刚接触android逆向不久，对逆向不是很了解，但对加固稍微有一点点了解，研究了下梆梆的加固包，发现其加固包目录下存在个加密的secData0.jar，现在市场上，加固厂商采用的加固方式一般是启动代理dex，然后通过DexClassLoader动态加载jar或dex的方式，将要保护的类加载到内存中去，但这种方式只能传参文件的路径，无法传参内存地址，不知道梆梆是怎么做到的，猜想是hook方式。本人也用过hook，但是兼容性和稳定性不太理想。接下来开始逆向分析，第一次发帖，写的有点乱，有些的不对的地方望大神指正。

1. 先将代理dex拖到jeb中看看其大致流程，代理dex是各大厂商普遍采用的，手动new要保护的Application类，调用native方法attach，将new出来的Application设置到应用程序上下文context中，系统调用代理类的onCreate时，再在其中调用手动new的Application的onCreate方法，至此，代理dex完成了启动目标dex。

SecAppWrapper中，有一静态变量 realApplication,在attachBaseContext函数中通过，根据，Helper.APPNAME，手动加载并实例化一个对象，Helper.APPNAME为null，应该是在so中通过反射赋值的，在调用attachBaseContext函数的时候，this.getClassLoader中已经关联到了要保护的dex，所以在此之前，一定做了解密dex，加载dex，并设置loader的操作。比Application更早提前执行的就是类的静态块代码，再看看静态块代码，只是加载了libSecShell.so，加载so的流程大致说下，java层通过System.loadLibrary函数，调用到c层的dvmLoadNativeCode->dlopen,dlopen把so加载到内存中后，会调用so中的init，init\_array，接下来dvmLoadNativeCode会通过dlsym查找到so中的JNI\_OnLoad函数，如果存在去执行。所以基本上可以确定梆梆对dex的解密、加载操作就是在so的这三个地方。

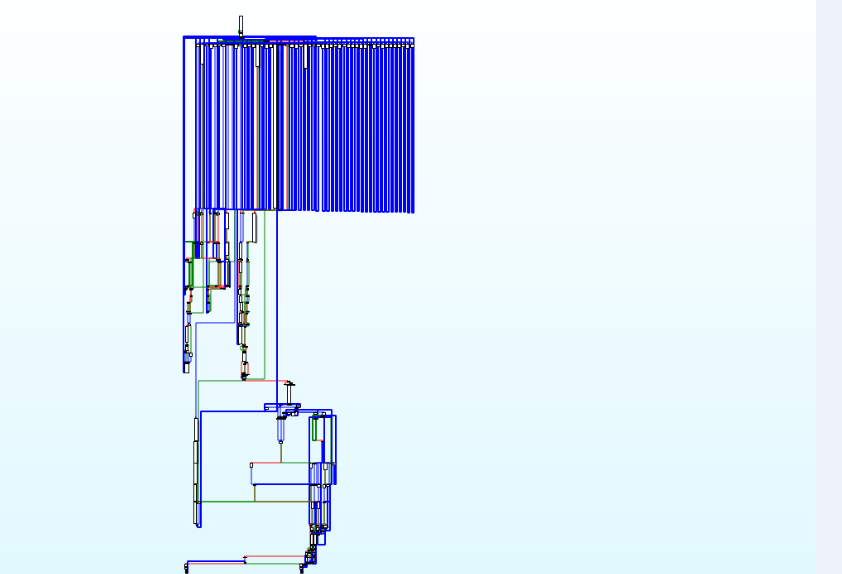


1. 将梆梆加固用到的libSecShell.so，拖到ida里，先静态，大致能不能看出点什么东西来，还好ida能打开，看看init，finit，JNI\_OnLoad，没找的init，finit也为空，判断所以操作在JNI\_OnLoad中进行。



3、JNI\_Onload分析

看看流程图，这流程服了，一般程序不会这么写的，肯定是llvm混淆过的，刚开始看确实头疼，不过慢慢看还是可以分析的，以下是我的分析过程



r3=15

if(r3==52){

Jni\_Onload退出

}else if(r3>52){

无效分支

}else{ //r3<52

进入switch{

case 15: GetEnv

case 28:

sub\_CF10：dlsym 查找符号（mmprotect，mmap unmmap）放到全局变量中，com/secshell/shellwrapper/Helper

解密字符串“Ljava/lang/String;“ “PKGNAME”

env-> env->GetStaticFieldID(clazz, "PKGNAME", "Ljava/lang/String;"));

获取PKGNAME = "com.example.test8"，

case 29:

动态解密解密用到的字符串

case 7 :

新建.cache目录，sub\_D570:写data/data/com.example.test8/.cache/classes.dve 24个字节，不知道有什么用。

找到/data/dalvik-cache/data@app@com.example.test8-1.apk@classes.dex = dexPtr，字符串

open(dexPtr)

case 32:主要的逻辑包括加载jar，解密jar，从jar中释放dex，反射调用install，hook函数，反调试函数等等。

{

jclass jc = findClass("com/secshell/shellwrapper/Helper");

env->RegisterNatives(jc,{"attach"," (Landroid/app/Application;Landroid/content/Context;)V }”);

p6325BD8519FF3EAD9668F36987CD0110（）//在4.4.4dalvik手机上没用到

{ dlopen(libdvm.so,0)

dlsym(\_Z22dvmRawDexFileOpenArray),sym1=dlsym(dvmRawDexFileOpen),

p4B2441F65675A731D2FEFF5CC2166CE2(findClass(dalvik/system/DexFile);env->getMethodId("<init>","(Ljava/lang/String;)V")

}

pB35C255E59C8408F082D5490EB26F32C（）

{

dlopen(libc.so)；

p845C09B79D87F284CE0F33FBC24DD952（）

{

进行inline hook函数，hook了好多libc.so中的函数

包括read、write 、close、 munmap 、msync、

\_\_openat、 pread64、 \_\_mmap2

}

}

pC0E901BB7A6D1B669B72D78E6861439F(/data/com.example.test8/.cache/classes.dex)

{

判断file指向内存前三个字节是否为“dex”or“dey”

}

pFBF8EA28AB5406DC5CFADBC7CE32467F()//读取so最后几个字节，存储起来

{

fopen( /data/data/com.example.test8/lib/libSecShell.so);

fseek(fd,-8,SEEK\_END);

fread(buf,1,8,fd);

fseek(fd,-10,SEEK\_END);

fclose(fd)

}

pD87C3778018C8497DE25DC3140A39FA6()

//打开/data/app/com.example.test8-1.app/从中取出secData0.jar

{

p64068FFF75D5FF726D395AD2CF88C6F7()

{

fd = open("/data/app/com.example.test8-1.app",0x0,0x0);

}

p5A0228D84B11FF138D5616E546386E2A()

{

strlen("assets/secData0.jar")

return 0x00002722

}

p8D083BC566F0CE8A42363E0F1CBA1CD9()

pDB44B5F00E6156543E0CAE1D01C88736()

{

从apk中抽取加密的secData0.jar到内存中

}

p34D946B85C4E13BE6E95110517F61C41(addr,len) //在这个函数中下断可以dump jar

{

解密SecData0.jar

sub\_195FC(0,addr,len)//解密算法所在的函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

{

p3CBBD6F30D91F38FCD0A378BE7E54877()

{

malloc分配内存地址存放到p5E7BF0B62C098453447B32884992D488，

p5E7BF0B62C098453447B32884992D488🡺存放明文dex地址

pDB44B5F00E6156543E0CAE1D01C88736()//在这个函数中下断可以dump dex

{从解密的jar中释放dex文件到p5E7BF0B62C098453447B32884992D488

inflateInit2\_();

inflate();

write(fd,buf,size);//write被hook，写到.cache上的为加密dex

inflateEnd();}

}

unk\_D268()

{ jni层

NewStringUTF(/data/data/com.example.test8/.cache/classes.jar);

NewStringUTF(/data/data/com.example.test8/.cache/classes.dex);

FindClass(com/secshell/shellwrapper/DexInstall);

FindClass(java/lang/Class);

GetMethodID(0x1D400071,"getClassLoader","()Ljava/lang/ClassLoader;");

CallObjectMethod(DexInstall, getClassLoad\_ID);

GetStaticMethodID(DexInstall,"install",

"(Ljava/lang/ClassLoader;Ljava/lang/String;)V");

CallStaticVoidMethod(DexInstall, install\_ID, loader, dexPath);

Java层：

makeDexElements-> openDexFileNative->dvmRawDexFileOpen(dex,dex)

梆梆hook了dvmRawDexFileOpen函数，会调用到

pB7F20650D654BF17487B377A15C6F5FF

pB7F20650D654BF17487B377A15C6F5FF()构建RawDexFile->cookie返回java层

{

case 7：

0xBEEAE44C = “/sclass.dex”

strcmp(/sclass.dex,/data/data/com.example.test8/.cache/classes.dex);

case 10:

strstr(/data/data/com.example.test8/.cache/classes.dex,

./cache/classes.dex)

case 9:

case 4:

dvmRawDexFileOpenArray(明文dex，len，

RawDexFile\*\* ppRawDexFile/\*0xBEEAE48C\*/)；

case 1:

case 8:

}

}

至此完成了加密jar的解密，释放dex，加载dex，设置loader，然后在代理壳的onCreate调用realApplication.onCreate，应用程序就运行起来了。

///////////////////////////////////////////////////////////////////

p071ADBC73D8008F1BE158FD0441DC741(//创建线程(kill -9))

{

case 8:

access("/data/app/com.example.test8-1.app");

case 9:

malloc 存储字符串"/data/app/com.example.test8-1.app"

case 0:

case 10:

0x0003148C pthread\_thread\_create（ p2CDCBA17913F0B54DE1DBA053AFBD7EB(kill -9)）；

}

p7E7056598F77DFCC42AE68DF7F0151CA\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 反调试

{

case 5：

prtcl(PR\_SET\_DUMPABLE,1,0,0,0);

getpid()

case 3:

case 6:

case 11:

pipe(sp+0x58);

case 8:

pid = fork();

if(pid >0)

r3=9;

else

r3=4;

case 9://父进程

设置管道为写端

case 9->case 10:

write(fd,1,);

case 4：

p845C09B79D87F284CE0F33FBC24DD952(libc.so,ptrace){

}

调用反调试函数，附加父进程，调用扫描函数

anti\_thread\_of\_process\_debug

p78D3797A85ACABF62A884C5574655B5A//启动线程 pEB4046F8D020AD7E1F60BA0D1D8F989B

r3=1;

case 1:

read(fd,buf,1) = '';

r3=5;

case 5:

栈检查

case 38：

}

p794BC17E009571800343687071A57359 //不清楚功能

///////////////////////////////////////////////////////////////////

case 38:

case 43：打开/data/app/com.example.test8-1.apk

fgets读取 /proc/pid/status

打开/data/data/com.example.test8/lib/libSecShell.so

打开/data/app/com.example.test8-1.apk

}

}

总结，梆梆官网免费版加固大致的流程就是这样的，语言组织能力太差，写的太粗糙，哪里有分析错误的地方，还请大神来指正，共同进步，不过还有两点没有搞明白

1. 为什么反调试线程会在，jar全部解密加载完成后再启动????????奇了怪
2. 上面完成之后，梆梆还调用了好多函数，比如fork，fork，两次fork之后然后在

子进程中调用：

fork\_execute\_dex2opt –>execute\_dex2opt-> dvmPrepForDexOpt-> getenv->

->dvmContinueOptimization不知道这个流程有什么用？？？