一个Android壳简单实现

又一年要过完了，没有什么提高，总结一下学习过程的中的笔记，顺便锻炼一下写文档的能力。

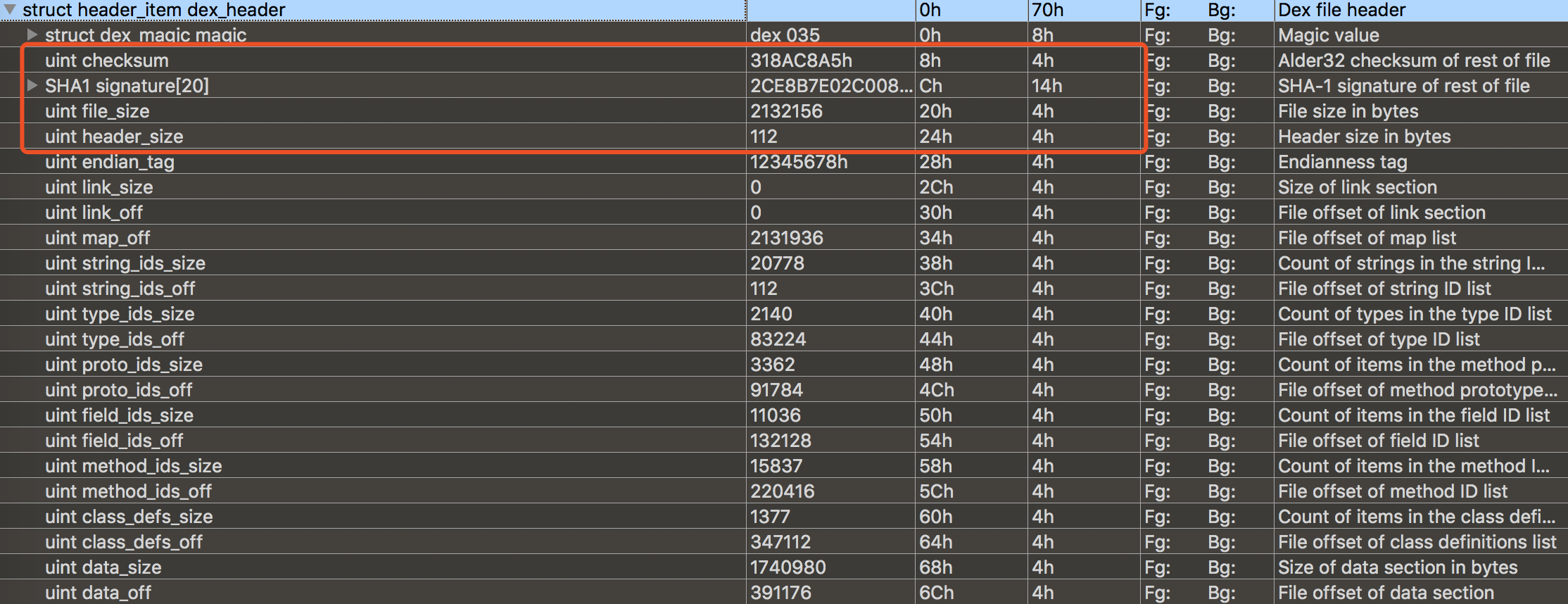
过时的技术，各位大佬请轻喷。

加壳程序工作流程：加密源程序APK文件，把加密后数据写入壳APK的Dex文件尾部并添加数据长度，修改Dex文件头部的checksum，signature，file\_size字段，使用源程序AndroidMainfest.xml替换壳程序的AndroidMainfest.xml文件内容。

脱壳程序工作流程：读取Dex文件末尾数据以及长度信息，解密数据保存文件，动态加载源APK。

加壳程序：

DEX文件头部结构如下：



修复CheckSum，CheckSum使用alder32算法校验除去magic，checksum以外的部分。

int FixCheckSum(unsigned char \*buffer, size\_t nLen)

{

long uCheckSum = adler32(buffer + 12,nLen - 12);

printf("new CheckSum is %02x\r\n",uCheckSum);

memcpy(buffer + 8,&uCheckSum,4);

return 0;

}

修复Signature，Signature使用sha1计算除去 magic ,checksum 和 signature以外的部分。

int FixSHA1Header(unsigned char \*buffer, int nLen)

{

unsigned char digest[SHA\_DIGEST\_LENGTH];

SHA\_CTX ctx;

SHA1\_Init(&ctx);

printf("%d\r\n",nLen - 32);

SHA1\_Update(&ctx,(buffer + 32),nLen - 32);

SHA1\_Final(digest,&ctx);

char result[SHA\_DIGEST\_LENGTH \* 2 + 1] = {0};

for (int i = 0; i < SHA\_DIGEST\_LENGTH; ++i)

{

sprintf(&result[i \* 2],"%02x",digest[i]);

}

memcpy(buffer + 12,digest,SHA\_DIGEST\_LENGTH);

printf("new SHA1 is %s\r\n",result);

return 0;

}

修改添加加密数据后的Dex文件大小

int FixDexHeader(unsigned char \*buffer, int nLen)

{

int nDexSize = nLen;

memcpy(buffer + 32,&nDexSize,4);

}

int Encrypt(char \*in,int nSize)

{

for (int i = 0; i < nSize; ++i)

{

in[i] = in[i] ^ 0xff;

}

return 0;

}

修改Dex头部信息时需要先修改filesize，在计算signature，最后计算checksum的值否则会提示DEX文件无效。

到这里加壳程序就完成了很简单，只是做加密数据，合并内容，修改dex header字段，生成新dex文件。

脱壳程序：

首先需要创建一个MyApplication继承Application，实现attachBaseContext()和onCreate方法，因为程序启动后会先调用Application，之后在调用Activity。

在attachBaseContext()里解密数据释放源APK文件，使用反射加载APK然后替换ClassLoader，代码如下。

@Override

protected void attachBaseContext(Context base) {

super.attachBaseContext(base);

File odex = this.getDir("test\_odex", MODE\_PRIVATE);

File libs = this.getDir("test\_lib", MODE\_PRIVATE);

odexPath = odex.getAbsolutePath();

libPath = libs.getAbsolutePath();

apkPath = odexPath + "/test.apk";

Log.i("smallsun", "odexPath" + odexPath + "\r\n");

Log.i("smallsun", "libPath" + libPath + "\r\n");

Log.i("smallsun", "apkPath" + apkPath + "\r\n");

File apkFile = new File(apkPath);

if (apkFile.exists()) {

Log.i("smallsun", "file is exists");

return;

}

try {

//创建APK

apkFile.createNewFile();

byte[] dexData = readDexFromAPK();

Log.i("smallsun", "decryptDexFile");

decryptDexFile(dexData);

//配置动态加载环境，获取主线程对象

Object currentActivityThread = RefInvoke.invokeStaticMethod("android.app.ActivityThread","currentActivityThread",new Class[]{},new Object[]{});

String packageName = this.getPackageName();

ArrayMap mPackage = (ArrayMap) RefInvoke.getFieldOjbect("android.app.ActivityThread",currentActivityThread,"mPackages");

WeakReference wr = (WeakReference) mPackage.get(packageName);

//创建被加壳APK的DexClassLoader对象 加载apk内的类和本地代码（c/c++代码）

DexClassLoader dexClassLoader = new DexClassLoader(apkPath,odexPath,libPath, (ClassLoader) RefInvoke.getFieldOjbect("android.app.LoadedApk", wr.get(), "mClassLoader"));

//把当前进程的DexClassLoader 设置成了被加壳apk的DexClassLoader

RefInvoke.setFieldOjbect("android.app.LoadedApk","mClassLoader",wr.get(),dexClassLoader);

Log.i("smallsun","ClassLoader:" + dexClassLoader);

try {

Object object = dexClassLoader.loadClass("com.apktest.MainActivity");

Log.i("smallsun", "object : " + object);

}catch (Exception e) {

Log.i("smallsun", "activity:" + Log.getStackTraceString(e));

}

} catch (IOException e) {

throw new RuntimeException(e);

}

}

到这里为止已经实现了一个简单的壳了，但是一些APP会有Application类，这时还需要替换源程序的Application，代码如下。

super.onCreate();

Log.i("smallsun", "onCreate");

String appClassName = null;

try {

ApplicationInfo ai = this.getPackageManager().getApplicationInfo(this.getPackageName(), PackageManager.GET\_META\_DATA);

Bundle bundle = ai.metaData;

if (bundle != null && bundle.containsKey("APPLICATION\_CLASS\_NAME")) {

appClassName = bundle.getString("APPLICATION\_CLASS\_NAME");

} else {

return;

}

} catch (PackageManager.NameNotFoundException e) {

e.printStackTrace();

}

//有值的话调用该Applicaiton

Object currentActivityThread = RefInvoke.invokeStaticMethod("android.app.ActivityThread", "currentActivityThread", new Class[]{}, new Object[]{});

Object mBoundApplication = RefInvoke.getFieldOjbect("android.app.ActivityThread", currentActivityThread, "mBoundApplication");

Object loadedApkInfo = RefInvoke.getFieldOjbect("android.app.ActivityThread$AppBindData", mBoundApplication, "info");

RefInvoke.setFieldOjbect("android.app.LoadedApk", "mApplication", loadedApkInfo, null);

Object oldApplication = RefInvoke.getFieldOjbect("android.app.ActivityThread", currentActivityThread, "mInitialApplication");

ArrayList<Application> mAllApplications = (ArrayList<Application>) RefInvoke.getFieldOjbect("android.app.ActivityThread", currentActivityThread, "mAllApplications");

mAllApplications.remove(oldApplication);

ApplicationInfo appinfo\_In\_LoadedApk = (ApplicationInfo) RefInvoke.getFieldOjbect("android.app.LoadedApk", loadedApkInfo, "mApplicationInfo");

ApplicationInfo appinfo\_In\_AppBindData = (ApplicationInfo) RefInvoke.getFieldOjbect("android.app.ActivityThread$AppBindData", mBoundApplication, "appInfo");

appinfo\_In\_LoadedApk.className = appClassName;

appinfo\_In\_AppBindData.className = appClassName;

Application app = (Application) RefInvoke.invokeMethod("android.app.LoadedApk", "makeApplication", loadedApkInfo, new Class[]{boolean.class, Instrumentation.class}, new Object[]{false, null});

RefInvoke.setFieldOjbect("android.app.ActivityThread", "mInitialApplication", currentActivityThread, app);

ArrayMap mProviderMap = (ArrayMap) RefInvoke.getFieldOjbect("android.app.ActivityThread", currentActivityThread, "mProviderMap");

Iterator it = mProviderMap.values().iterator();

while (it.hasNext()) {

Object providerClientRecord = it.next();

Object localProvider = RefInvoke.getFieldOjbect("android.app.ActivityThread$ProviderClientRecord", providerClientRecord, "mLocalProvider");

RefInvoke.setFieldOjbect("android.content.ContentProvider", "mContext", localProvider, app);

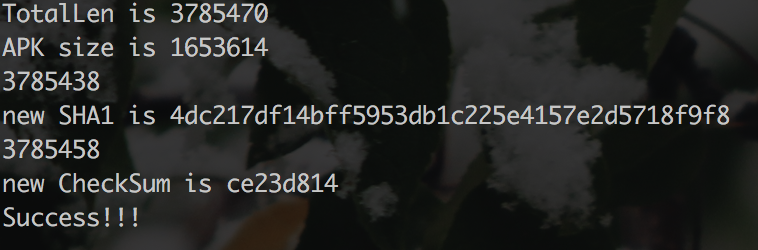
}

app.onCreate();

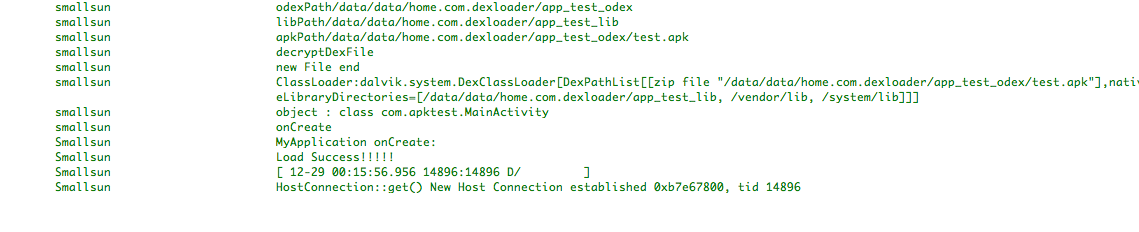
}

至此实现了一个壳的雏形，测试一下效果。

加密源APK生成新dex文件



把新dex文件放入壳Apk中，重新签名，安装程序,启动程序，通过log可以看见执行了源APK中的代码。



再说一下实现过程中发现的一些问题，在java层进行解密及动态加载过程很容易被反编译，反编译后的伪代码与源码几乎没差，可以清楚的看出壳的运行逻辑，同时还会释放出源apk文件存放在本地。这时需要把关键代码移植到Native层，同时在内存中加载文件，这里就不多说了可以看代码，为了偷懒部分代码还是在java实现，完全可以把attachBaseContext和onCreate用native实现。

So加固

1. 通过加密节的方式加密

解密程序流程：通过\_\_attribute\_\_((section(".mytext")))属性将要加密的函数定义在.mytext节中，实现解密函数，添加\_\_attribute\_\_((constructor))属性，将代码定义在.init\_array段。

　解密函数的实现很简单，这里我们首先在getLibAddr函数中通过/proc/<pid>/maps文件获得加载的so文件路径和基址，通过ehdr->e\_entry这个变量获取到被加密节的大小，ehdr->e\_shoff获得加密节的地址偏移解密数据。

代码如下：

extern "C" JNIEXPORT

//替换成自定义section的名字

jint JNICALL Java\_home\_com\_sotest\_MainActivity\_Add(JNIEnv \*env, jobject obj) \_\_attribute\_\_((section(".mytext")));

jint JNICALL Java\_home\_com\_sotest\_MainActivity\_Add(JNIEnv \*env, jobject obj)

{

printf("Call Add Func!\r\n");

return (jint)(1 + 1);

}

unsigned int GetLibAddr()

{

char \*szName = "libnative-lib.so";

int nPid = getpid();

unsigned int nBase = 0;

char buffer[1024] = {0};

sprintf(buffer,"/proc/%d/maps",nPid);

FILE \*fp = fopen(buffer,"r");

if (fp != nullptr)

{

while(fgets(buffer,sizeof(buffer),fp))

{

if(strstr(buffer,szName))

{

//分割字符串 返回-之前内容

char \*temp;

temp = strtok(buffer,"-");

nBase = strtoul(temp,nullptr,16);

LOGD("BASE IS 0x%x\r\n",nBase);

break;

}

}

}

fclose(fp);

return nBase;

}

void Init() \_\_attribute\_\_((constructor)); //使用”attribute((constructor))”将函数放到”.init\_array”段

void Init()

{

LOGD("Call Init");

unsigned int nBase = GetLibAddr();

Elf32\_Ehdr \*pEhdr = (Elf32\_Ehdr \*)nBase;

//获取加密节地址

unsigned int mytextBase = pEhdr->e\_shoff + nBase;

unsigned int nBlock = (pEhdr->e\_entry) >> 16;//加密节的大小

unsigned int nSize = (pEhdr->e\_entry) & 0xffff;//加密节的大小

LOGD("nBlock = %d,nSize = %d", nBlock,nSize);

LOGD("mytextBase = 0x%x", mytextBase);

//修改内存属性

if(mprotect((void\*)(mytextBase / PAGE\_SIZE \* PAGE\_SIZE),4096 \* nSize,PROT\_READ | PROT\_EXEC | PROT\_WRITE) != 0)

{

LOGD("mem privilege change failed");

}

//解密

for(int i = 0;i < nBlock; i++)

{

unsigned char \*addr = (unsigned char\*)(mytextBase + i);

\*addr = ~(\*addr);

}

if(mprotect((void \*) (mytextBase / PAGE\_SIZE \* PAGE\_SIZE), 4096 \* nSize, PROT\_READ | PROT\_EXEC) != 0)

{

LOGD("mem privilege change failed");

}

LOGD("Decrypt success!!!");

}

加密程序流程：解析elf结构，从文件头读取section偏移shoff，shnum和shstrtab，读取shstrtab中的字符串，从shoff读取section header，通过pShdr->sh\_name读取节表名比较是否自定义节表名，通过pShdr->sh\_offset pShdr->sh\_size读取节表大小以及内容进行加密，修改section字段中的e\_shoff为pShdr->sh\_addr，修改e\_entry为pShdr->sh\_size，写入文件。

作为动态链接库，e\_entry入口地址是无意义的，因为程序被加载时，设定的跳转地址是动态连接器的地址，这个字段是可以被作为数据填充的。

so装载时，与链接视图没有关系，即e\_shoff、e\_shentsize、e\_shnum和e\_shstrndx这些字段是可以任意修改。

代码如下：

int FormatElf(unsigned char \*buffer)

{

//自定义节名

char \*szSection = ".mytext";

//ELF头地址

Elf32\_Ehdr \*pEhdr = (Elf32\_Ehdr\*)buffer;

if (!((pEhdr->e\_ident[0] == 0x7F) &&

(pEhdr->e\_ident[1] == 'E') &&

(pEhdr->e\_ident[2] == 'L') &&

(pEhdr->e\_ident[3] == 'F')))

{

printf("无效的文件\r\n");

return 0;

}

//定位到节表

Elf32\_Shdr \*pShdr = (Elf32\_Shdr\*)(buffer + pEhdr->e\_shoff);

//定位字符串表

char \*str = nullptr;

Elf32\_Shdr \*pShdrstr = (Elf32\_Shdr \*)(buffer + pEhdr->e\_shoff + sizeof(Elf32\_Shdr) \* pEhdr->e\_shstrndx);

str = (char\*)(pShdrstr->sh\_offset + buffer);

int nOffset = 0;

int nSize = 0;

for (int i = 0; i < pEhdr->e\_shnum; i++)

{

if (strcmp(str + pShdr->sh\_name,szSection) == 0)

{

//获取偏移和大小

nOffset = pShdr->sh\_offset;

nSize = pShdr->sh\_size;

printf("find section %s offset is 0x%x size is 0x%x\n",szSection,nOffset,nSize);

break;

}

pShdr++;

}

unsigned char \*sectionBuf = (unsigned char \*)(buffer + nOffset);

//计算section占用内存几页，一个页大小4096

int nPage = nSize / 4096 + (nSize % 4096 == 0 ? 0 :1);

printf("nOffset = 0x%x, nSize = 0x%x\n", nOffset, nSize);

printf("nPage = %d\n",nPage);

pEhdr->e\_entry = (nSize << 16) + nPage;

pEhdr->e\_shoff = nOffset;

//加密

printf("sectionBuf is %x",sectionBuf);

for (int j = 0; j < nSize; j++)

{

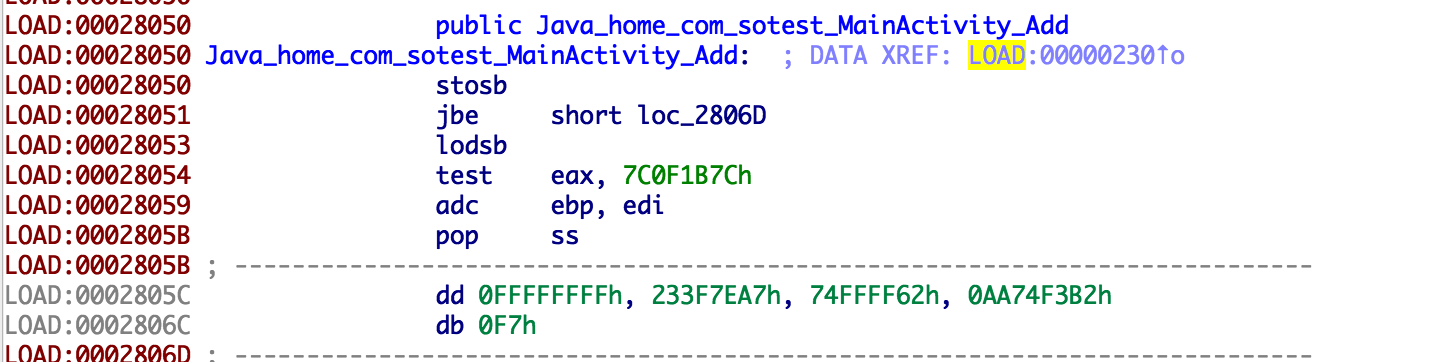
sectionBuf[j] = ~sectionBuf[j];

}

return 1;

}

加密后Add函数如下



1. 加密SO中指定函数

加密程序流程：解析elf文件，读取文件头，获取e\_phoff、e\_phentsize和e\_phnum信息，通过Elf32\_Phdr中的p\_type字段，找到DYNAMIC，遍历.dynamic，找到.dynsym、.dynstr、.hash section文件中的偏移和.dynstr的大小，根据函数名称，计算hash，根据hash值，找到下标hash % nbuckets的bucket；根据bucket中的值，读取.dynsym中的对应索引的Elf32\_Sym符号；从符号的st\_name所以找到在.dynstr中对应的字符串与函数名进行比较。若不等，则根据chain[hash % nbuckets]找下一个Elf32\_Sym符号，直到找到或者chain终止为止，找到函数对应的Elf32\_Sym符号后，即可根据st\_value和st\_size字段找到函数的位置和大小，加密数据写入文件。

代码如下：

static unsigned elfhash(const char \*\_name)

{

const unsigned char \*name = (const unsigned char \*) \_name;

unsigned h = 0, g;

while(\*name) {

h = (h << 4) + \*name++;

g = h & 0xf0000000;

h ^= g;

h ^= g >> 24;

}

return h;

}

void GetTargetFuncInfo(unsigned char\* buffer,char \*funcName,funcInfo \*info)

{

Elf32\_Ehdr \*pEhdr = (Elf32\_Ehdr \*) buffer;

Elf32\_Phdr \*pHdr = (Elf32\_Phdr \*) (buffer + pEhdr->e\_phoff);

Elf32\_Word DynSize = 0;

Elf32\_Word DynStrsz = 0;

Elf32\_Off DynOffset = 0;

Elf32\_Addr DynSymtab, DynStrtab, DynHash;

for (int i = 0; i < pEhdr->e\_phnum; i++)

{

if (pHdr->p\_type == PT\_DYNAMIC)

{

DynSize = pHdr->p\_filesz;

DynOffset = pHdr->p\_offset;

printf("Find Section %s size = %d offser = %02x\n", "PT\_DYNAMIC", DynSize, DynOffset);

}

pHdr++;

}

Elf32\_Dyn \*pDyn = (Elf32\_Dyn \*) (buffer + DynOffset);

for (int j = 0; j < DynSize / sizeof(Elf32\_Dyn); j++)

{

//符号表位置

if (pDyn->d\_tag == DT\_SYMTAB)

{

DynSymtab = pDyn->d\_un.d\_ptr;

printf("Find .dynsym, addr = 0x%x, val = 0x%x\n", DynSymtab, pDyn->d\_un.d\_val);

}

//hash位置

if (pDyn->d\_tag == DT\_HASH)

{

DynHash = pDyn->d\_un.d\_ptr;

printf("Find .hash, addr = 0x%x\n", DynHash);

}

//字符串位置

if (pDyn->d\_tag == DT\_STRTAB)

{

DynStrtab = pDyn->d\_un.d\_ptr;

printf("Find .dynstr, addr = 0x%x\n", DynStrtab);

}

//字符串大小

if (pDyn->d\_tag == DT\_STRSZ)

{

DynStrsz = pDyn->d\_un.d\_val;

printf("Find .dynstr size, size = 0x%x\n", DynStrsz);

}

pDyn++;

}

//定位字符串表

char \*dynStr = (char \*) malloc(DynStrsz);

memcpy(dynStr, buffer + DynStrtab, DynStrsz);

/\* nbucket

\*-----------------

\* nchain

\*------------------

\* bucket[0]

\* ...

\* bucket[nbucket-1]

\* ------------------

\* chain[0]

\* ...

\* chain[nchain-1]

\*/

//计算函数名称经过hash运行后的值

unsigned funHash = elfhash(funcName);

printf("Function %s hashVal = 0x%x\n", funcName, funHash);

//获取nbucket的值

int nNbucket = \*(int \*) (buffer + DynHash);

printf("nbucket = %d\n", nNbucket);

//获取nchain

int nNchain = \*(int \*) (buffer + DynHash + 4);

printf("nchain = %d\n", nNchain);

funHash = funHash % nNbucket; //bucket[X%nbucket]给出了一个索引y，该索引可用于符号表，也可用于chain表

printf("funHash mod nbucket = %d \n", funHash);

int nFunIndex = \*(int \*) (buffer + DynHash + 8 + funHash \* 4);//y = bucket[X%nbucket]返回的索引y

printf("funcIndex:%d\n", nFunIndex);

Elf32\_Sym \*pSym = (Elf32\_Sym \*) (buffer + DynSymtab + nFunIndex \* sizeof(Elf32\_Sym));//该索引对应的符号表

//如果索引y对应的符号表不是所需要的,那么chain[y]则给出了具有相同哈希值的下一个符号表项

if (strcmp(dynStr + pSym->st\_name, funcName) != 0)

{

while (1)

{

printf("hash:%x,nbucket:%d,funIndex:%d\n", DynHash, nNbucket, nFunIndex);

nFunIndex = \*(int \*) (buffer + DynHash + 4 \* (2 + nNbucket + nFunIndex)); //搜索chain链

printf("funcIndex:%d\n", nFunIndex);

if (nFunIndex == 0) {

puts("Cannot find funtion!\n");

return;

}

pSym = (Elf32\_Sym \*) (buffer + DynSymtab + nFunIndex \* sizeof(Elf32\_Sym)); //chain[]中对应的符号表

if (strcmp(dynStr + pSym->st\_name, funcName) == 0)

{

break;

}

}

}

printf("Find: %s, offset = 0x%x, size = 0x%x\n", funcName, pSym->st\_value, pSym->st\_size);

info->st\_value = pSym->st\_value;

info->st\_size = pSym->st\_size;

free(dynStr);

return;

}

解密程序流程：解密流程为加密逆过程，区别是找到PT\_DYNAMIC后需取p\_vaddr和p\_filesz字段而不是 p\_offset字段。

代码如下：

void GetTargetFuncInfo(unsigned int nBase,const char \*funcName,funcInfo \*info)

{

Elf32\_Ehdr \*pEhdr = (Elf32\_Ehdr \*) nBase;

Elf32\_Phdr \*pHdr = (Elf32\_Phdr \*) (nBase + pEhdr->e\_phoff);

LOGD("phdr = 0x%p, size = 0x%x\n", pEhdr, pEhdr->e\_phnum);

for (int i = 0; i < pEhdr->e\_phnum; ++i)

{

LOGD("phdr = 0x%p\n", pHdr);

//获得动态链接节

if(pHdr->p\_type == PT\_DYNAMIC)

{

LOGD("Find .dynamic segment");

break;

}

pHdr ++;

}

Elf32\_Off dyn\_vaddr = pHdr->p\_vaddr + nBase;

Elf32\_Word dyn\_size = pHdr->p\_filesz;

Elf32\_Dyn \*dyn;

Elf32\_Addr dyn\_symtab, dyn\_strtab, dyn\_hash;

Elf32\_Word dyn\_strsz;

for (int i = 0; i < dyn\_size / sizeof(Elf32\_Dyn); i++)

{

dyn = (Elf32\_Dyn \*)(dyn\_vaddr + i \* sizeof(Elf32\_Dyn));

//符号表位置

if(dyn->d\_tag == DT\_SYMTAB)

{

dyn\_symtab = (dyn->d\_un).d\_ptr;

LOGD("Find .dynsym section, addr = 0x%x\n", dyn\_symtab);

}

//获得hash段

if(dyn->d\_tag == DT\_HASH)

{

dyn\_hash = (dyn->d\_un).d\_ptr;

LOGD("Find .hash section, addr = 0x%x\n", dyn\_hash);

}

//保存函数字符串的位置

if(dyn->d\_tag == DT\_STRTAB)

{

dyn\_strtab = (dyn->d\_un).d\_ptr;

LOGD("Find .dynstr section, addr = 0x%x\n", dyn\_strtab);

}

//字符串长度

if(dyn->d\_tag == DT\_STRSZ)

{

dyn\_strsz = (dyn->d\_un).d\_val;

LOGD("Find strsz size = 0x%x\n", dyn\_strsz);

}

}

dyn\_symtab += nBase;

dyn\_hash += nBase;

dyn\_strtab += nBase;

dyn\_strsz += nBase;

/\* nbucket

\*-----------------

\* nchain

\*------------------

\* bucket[0]

\* ...

\* bucket[nbucket-1]

\* ------------------

\* chain[0]

\* ...

\* chain[nchain-1]

\*/

unsigned funHash = elfhash(funcName);//获得函数名称经过hash运行后的值

Elf32\_Sym \*funSym = (Elf32\_Sym \*) dyn\_symtab;

char \*dynstr = (char\*) dyn\_strtab;

unsigned nbucket = \*((int \*) dyn\_hash);//获得nbucket的值

int \*bucket = (int \*)(dyn\_hash + 8);//bucket链

unsigned int \*chain = (unsigned int \*)(dyn\_hash + 4 \* (2 + nbucket));//越过bucket链，到达chain链

LOGD("hash = 0x%x, nbucket = 0x%x\n", funHash, nbucket);

//bucket[X%nbucket]给出了一个索引y，该索引可用于符号表，也可用于chain表

int mod = (funHash % nbucket);

LOGD("mod = %d\n", mod);

LOGD("i = 0x%d\n", bucket[mod]);

//i = mod = bucket[funHash%nbucket]，通过遍历i = chain[i]表，找到funSym对应的符号表

int i;

for(i = bucket[funHash % nbucket]; i != 0; i = chain[i])

{

if(strcmp(dynstr + (funSym + i)->st\_name, funcName) == 0)

{

LOGD("Find %s\n", funcName);

break;

}

}

info->st\_value = (funSym + i)->st\_value;//函数对应符号表中保存函数的地址

info->st\_size = (funSym + i)->st\_size;//函数符号表中保存函数的大小

LOGD("st\_value = %d,st\_size = %d",info->st\_value,info->st\_size);

}

void Init() \_\_attribute\_\_((constructor)); //使用”attribute((constructor))”将函数放到”.init\_array”段

void Init()

{

LOGD("Call Init");

unsigned int nBase = GetLibAddr();

Elf32\_Ehdr \*pEhdr = (Elf32\_Ehdr \*)nBase;

funcInfo info;

const char funcName[] = "Java\_home\_com\_sotest\_MainActivity\_Add";

GetTargetFuncInfo(nBase,funcName,&info);

unsigned int nPage = info.st\_size / PAGE\_SIZE + ((info.st\_size % PAGE\_SIZE == 0) ? 0 : 1);

LOGD("nPage = %02x", nPage);

LOGD("nPage = %02x", PAGE\_SIZE);

if(mprotect((void \*) ((nBase + info.st\_value) / PAGE\_SIZE \* PAGE\_SIZE), 4096 \* nPage, PROT\_READ | PROT\_EXEC | PROT\_WRITE) != 0)

{

LOGD("mem privilege change failed");

}

for(int i = 0;i < info.st\_size - 1; i++)

{

char \*addr = (char\*)(nBase + info.st\_value - 1 + i);

\*addr = ~(\*addr);

}

if(mprotect((void \*) ((nBase + info.st\_value) / PAGE\_SIZE \* PAGE\_SIZE), 4096 \* nPage, PROT\_READ | PROT\_EXEC) != 0)

{

LOGD("mem privilege change failed");

}

}

参考各位大佬帖子地址

<https://bbs.pediy.com/thread-191649.htm>

https://bbs.pediy.com/thread-191649.htm

<https://bbs.pediy.com/thread-216119.htm>

https://bbs.pediy.com/thread-225303.htm

还有四哥的安卓逆向大黄书

代码：https://github.com/smallsun107/AndroidShell

到这写不下去了，收回要锻炼写文档的那句话，贴了代码后不知道说些什么了，dex和so动态加载部分可以看看大佬写的文章，这篇文章属于拿来主义吧，参考了许多大佬写的东西，但实践过程中也学到了不少姿势，再次熟悉了dex和elf文件格式等等。