## 插件动态加载安全场景和危害

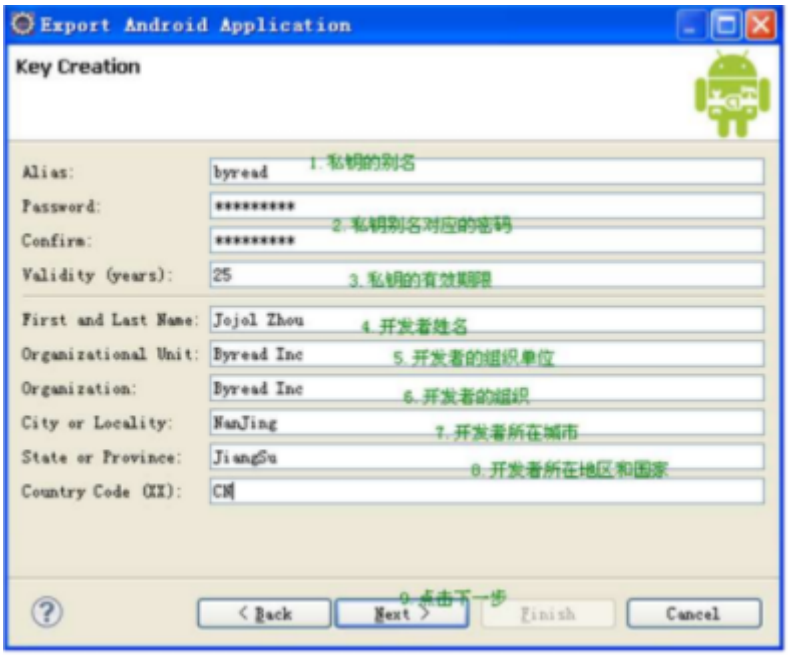
Android插件化机制具有模块解耦、可动态升级、按需加载，内存占用更低、节省升级流量等诸多便利，很多应用都使用了动态加载插件的技术，包括主流的APP使用的热补丁技术也是动态加载形式的一种，应用插件化是未来一个大的趋势，然而许多恶意软件为了绕过安全检测，也采用了在运行时下载并动态加载代码的方式。

## 插件动态加载安全漏洞分类

- APK签名校验原理和绕过
- DEX、JAR、SO绕过

## 加载绕过执行漏洞原理

DEX_APK_JAR等加载绕过执行漏洞指的是这几种文件在安装或者是动态加载的过程中被恶意的替换，并且绕过了签名校验或者是文件自校验或者是文件MD5等校验方式。




## APK签名校验原理和绕过

对于APK的验证一般使用的是签名校验或者是MD5校验，使用签名校验的方式较多。对于MD5校验的方式较为简单，

主要介绍签名校验是如何实现的。APK加载有两种方式，一种是校验签名实现安装，另外一种是校验签名实现动态加载。

绕过的问题一般是在实现动态加载的过程中，或者是破解实现二次打包的场景。

签名校验方式：

Android签名apk之后，会有一个META-INF文件夹，这里有三个文件：

- MANIFEST.MF

  存储的是每一个文件对应的SHA1（或者SHA256）消息摘要算法提取出该文件的摘要然后进行BASE64编码。

- CERT.SF

  计算这个MANIFEST.MF文件的整体SHA1值，再经过BASE64编码后，记录在CERT.SF主属性块（在文件头上）的 “SHA1-Digest-Manifest” 属性值值下。

  逐条计算MANIFEST.MF文件中每一个块的SHA1，并经过BASE64编码后，记录在CERT.SF中的同名块中，属性的名字是 “SHA1-Digest” 。

- CERT.RSA

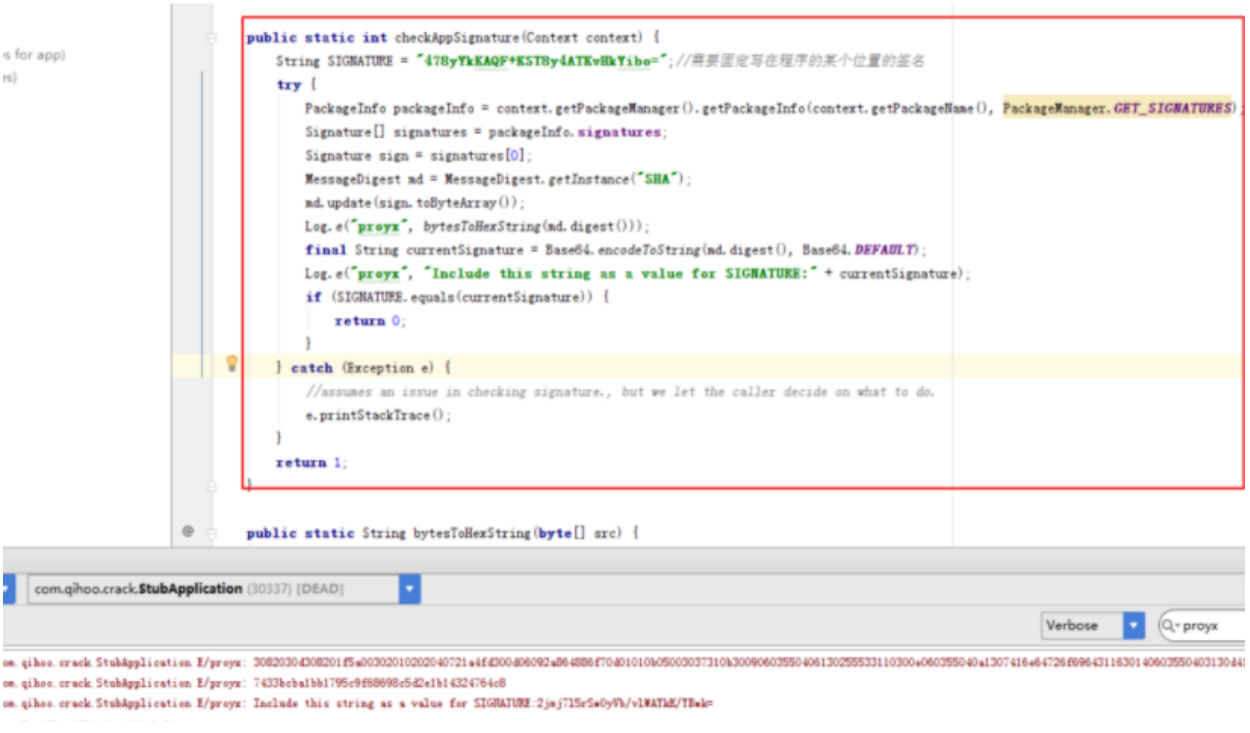
  会把之前生成的 CERT.SF文件， 用私钥计算出签名, 然后将签名以及包含公钥信息的数字证书一同写入 CERT.RSA 中保存。CERT.RSA是一个满足PKCS7格式的文件。

签名校验可以实现在Java层和native层。校验方式又分为两种一种是校验文件的DEX的MD5，另外一种是校验签名证书文件。

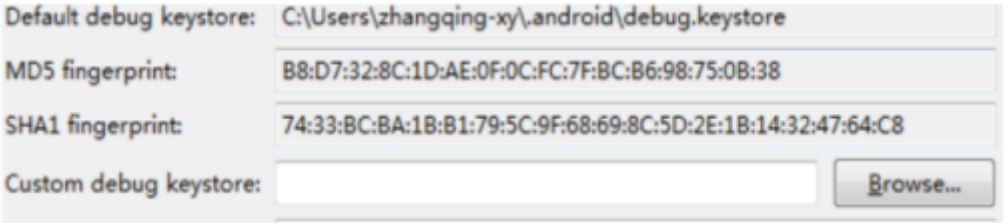
校验DEX的MD5校验方式有两种，一种是在so中去校验，另外一种是在DEX中实现在线签名校验，但是DEX校验的问题在于如果修改了so文件那么校验是无法检测到的，当然so文件也可以实现在线签名校验。

签名证书的校验验证方式比较完整，一般会把签名文件的MD5或者是SHA1值计算出来写死在so中或者是java代码中，在apk运行时候去检测当前apk的签名文件的MD5或者是SHA1值是否是与原有的值是一样的，不过MD5和SHA1的碰撞因子依靠现在的计算机是可以在秒单位的时间内寻找出来的。

## Java中计算签名的方式



会发现打印之后的SHA1与原签名文件的MD5一样




反编译之后的搜索Landroid/content/pm/PackageInfo;->signatures:[Landroid/content/pm/Signature位置：

```
invoke-interface {v8, v2}, Ljava/util/List;->add(Ljava/lang/Object;)Z

iget-object v8, v0, Landroid/content/pm/PackageInfo;->signatures:[Landroid/content/pm/Signature;

if-eqz v8, :cond_1

array-length v9, v8

if-lez v9, :cond_1

const/4 v9, 0x0

aget-object v8, v8, v9
```

## Native中校验签名方式

```
JNIEXPORT void JNICALL Java_com_example_testsign_JNIC_checkSign(JNIEnv * env, jclass cls, jobject context){
        //get PackageManager
        jclass contextClass = (*env)->GetObjectClass(env, context);
        jmethodID mid = (*env)->GetMethodID(env, contextClass, "getPackageManager", "()Landroid/content/pm/PackageManager;");
        jobject pm = (*env)->CallObjectMethod(env, context, mid);

        //get PackageName
        mid = (*env)->GetMethodID(env, contextClass, "getPackageName", "()Ljava/lang/String;");
        jstring packageName = (jstring) (*env)->CallObjectMethod(env, context, mid);

        // getPackageInfo     packageManager->getPackageInfo(packageName, GET_SIGNATURES);
        jclass the_class = (*env)->GetObjectClass(env, pm);
        mid = (*env)->GetMethodID(env, the_class, "getPackageInfo","(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;");
        jobject packageInfo = (*env)->CallObjectMethod(env, pm, mid, packageName,0x40); //GET_SIGNATURES = 64;

        //signatures[0]
        the_class = (*env)->GetObjectClass(env, packageInfo);
        jfieldID fid = (*env)->GetFieldID(env, the_class, "signatures", "[Landroid/content/pm/Signature;");
        jobjectArray signatures = (jobjectArray) (*env)->GetObjectField(env, packageInfo, fid);
        jobject sig = (*env)->GetObjectArrayElement(env, signatures, 0);

        //hashcode
        the_class = (*env)->GetObjectClass(env, sig);
        mid = (*env)->GetMethodID(env, the_class, "hashCode", "()I");
        int sig_value = (int) (*env)->CallIntMethod(env, sig, mid);
        LOGE("%d\n",sig_value);

        //finish 此处校验签名hashCode是否一致，如果不一致调用finish方法，因为finish没有返回值，所用调用的是CallVoidMethod
        /*
        jobject obj = (*env)->GetObjectClass(env, context);
        jmethodID mid = (*env)->GetMethodID(env, obj, "finish", "()V");
        (*env)->CallVoidMethod(env, context, mid);
        */
}
```

## 绕过姿势

APK的执行绕过的方式一般是修改APK内部的本地校验函数或者是修改在线校验函数或者是使用
hook的方式实现绕过。

**使用NDK实现签名校验的优缺点：**

优点:

NDK的好处是增加了程序的复杂度，因为so通过反汇编，得到的是ARM代码，而不是smali这种易
重修改、易理解的代码。修改so难度将大增。

**缺点:**

在so中实现签名校验功能，但在实现中将用于对比的签名HASH串明文存放在代码中，编译后，字符
串存放在.rodata段，还是可以修改并破解的。所以这个签名的hash传加密处理。

## DEX、JAR、SO绕过

一般需要动态加载的DEX文件会转换成JAR格式，其实本质还是一个DEX文件，在APK运行种很少有
J2SE生成的JAR文件。
DEX、JAR的校验方式有两种。
一种是校验文件的MD5的方式，存在本地和在线校验两种方式。区别是一种是把正确的MD5放在服
务器，另外一种是把正确的MD5固化在本地。
另外一种是校验DEX文件中的checksum和signature字段。
DEX、JAR的执行绕过的方式一般是修改APK内部的本地校验函数或者是修改在线校验函数或者是使
用hook的方式实现绕过。

## 影响和危害

DEX_APK_JAR等加载绕过执行的利用一般是发生在APK破解过程中比较多，APK破解的危害一般是广告、注入恶意的代码窃取用户隐私、欺骗、刷量、以及替换支付SDK等危害正常的开发者的利益。

## 加载绕过执行漏洞挖掘

对于APK的校验一般会是校验签名，校验签名会用到PackageManager，所以搜索关键词PackagerManger或者是signatures在逆向过程中找到对应的关键代码的位置，查看是否是本地校验
签名或者是在线校验签名。进而以修改返回值的形式进行签名绕过。
对于DEX/JAR文件的校验一般会是进行MD5或者是文件头的checksum等校验，定位这样的关键代
码一般是查找对应文件名的位置，第一次使用这个文件时候进行校验，然后进行函数修改绕过。

## APK的JAVA层绕过定位关键代码

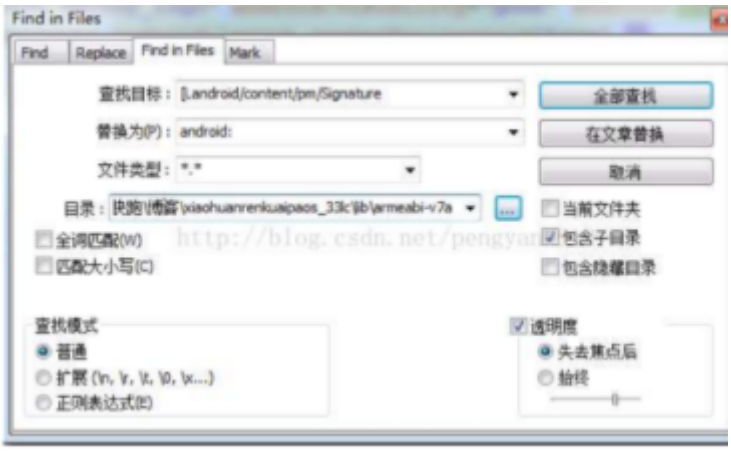
```
private String a1(String arg5) {
    CertificateException v1_1;
    IOException v1_2;
    String v0_5;
    PackageInfo v0_2;
    PackageInfo v1 = null;
    PackageManager v0 = this.mContext.getPackageManager();
    int v2 = 6
    try {
        v0_2 = v0.getPackageInfo(arg5, v2);
    }
    catch(PackageManager$NameNotFoundException v0_1) {
        v0_1.printStackTrace();
        v0_2 = v1;
    }

    if(v0_2 != null) {
        ByteArrayInputStream v2_1 = new ByteArrayInputStream(v0_2.signatures[0].toByteArray());
        try {
            v0_5 = fj.p(CertificateFactory.getInstance("X.509").generateCertificate(((InputStream)
                    v2_1)).getEncoded());
        }
        catch(IOException v0_3) {

        return v0_5;
    }

    public boolean hD() {
        String v0 = this.a1(this.mContext.getPackageName());
        String v1 = this.vs.getProperty("signature").toUpperCase().trim();
        gv.d("DEBUG", "your    signature is " + v0 + " len:" + v0.length());
        gv.d("DEBUG", "licence signature is " + v1 + " len:" + v1.length());
        return v0.equals(v1);
    }
```

## APK的SO层绕过定位关键代码








## 修复和防护

1、由于逆向C/C++代码要比逆向Java代码困难很多，所以关键代码部位应该使用Native C/C++来编写。
为了防止SO层的签名校验绕过，可以使用如下两种方式，从而尽可能的避免被破解。
 客户端将本地程序信息上传到服务端，服务端返回一段校验代码。客户端动态执行代码，返回校验
结果。
 在登陆接口将登录信息在NDK层进行加密，用签名信息进行加密，在登陆接口实现中，进行解密，
如果失败不允许登陆。
2、使用一些方式防止二次打包，比如在Manifest中添加自定义的属性来防止反编译以及二次打包，DEX
加壳保护，DEX指令动态加载保护，高级混淆保护。