

Android 安全编程指南

100tal.com

	版 本	姓 名	时 间	备 注
创建者	V1.0	吴友胜	2017/09/01	创 建
修 改				

目 录

1. 组件安全分析.....	4
1.1 组件安全准则.....	4
1.2 Activity 安全.....	4
1.2.1 私有 Activity.....	4
1.2.2 公有 Activity.....	5
1.2.3 伙伴 Activity.....	5
1.2.4 内部 Activity.....	5
1.3 Service 安全.....	6
1.3.1 私有 Service.....	6
1.3.2 公共 Service.....	6
1.3.3 合作 Service.....	6
1.3.4 内部 Service.....	6
1.4 Broadcast 安全.....	6
1.4.1 私有 Broadcast.....	7
1.4.2 公共 Broadcast.....	7
1.4.3 内部 Broadcast.....	7
1.5 Provider 安全.....	7
1.5.1 私有 Provider.....	7
1.5.2 公共 Provider.....	7
1.5.3 合作 Provider.....	7
1.5.4 内部 Provider.....	7
1.5.5 部分 Provider.....	7
1.6 组件安全问题.....	8
1.6.1 组件调用安全.....	8
1.6.2 组件通信安全.....	9
1.6.3 共享数据安全.....	10
2. 软件安全分析.....	11
2.1 防止逆向技术.....	11
2.1.1 代码混淆技术.....	11
2.1.2 证书签名校验.....	12
2.1.3 动态加载检测.....	14
2.1.4 逆向工具保护.....	17
2.2 动态分析技术.....	18
2.3 Hook 技术.....	19
2.4 Rootkit 技术.....	22
2.4.1 常用检测技术.....	22
2.4.2 Android 检测技术.....	22
3. 存储安全分析.....	23
3.1 常见安全问题.....	23
3.2 隐私明文存储.....	23
3.2.1 文件明文存储.....	23
3.2.2 数据库明文存储.....	24
3.3 软件数据存储.....	25
3.3.1 基于密钥加密.....	25

3.3.2 基于口令加密.....	27
3.3.3 KeyStore 保护.....	28
3.4 内部外部存储.....	29
3.4.1 内部存储安全.....	30
3.4.2 外部存储安全.....	30
4. 通信安全分析.....	31
4.1 常见安全问题.....	31
4.2 使用 HTTPS 传输.....	31
4.3 限定受信 SSL.....	31
4.3.1 自定义 X509TrustManager.....	31
4.3.2 自定义 HostnameVerifier.....	34
4.3.3 信任全部主机.....	35
4.3.4 WebView 的 HTTPS 安全.....	35
5. 理解 Android 系统.....	36
5.1 架构和服务.....	36
5.2 安全的边界.....	38
5.3 安全的架构.....	39

1. 组件安全分析

1.1 组件安全准则

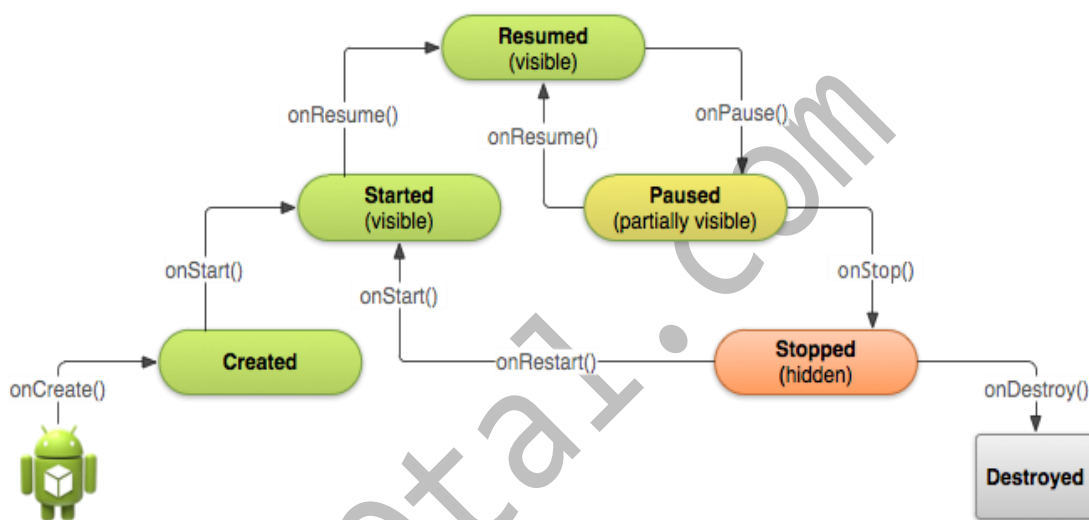
最小化组件暴露：对不会参与跨越应用间调用的组件添加 `android:exported="false"` 属性，这个属性说明它是私有的。如果是APP内部调用，给组件加入签名权限以防止不同签名的应用恶意调用。

设组件访问权限：对于跨越应用调用的组件或公开的Broadcast Receive设置权限，只有具有该权限的组件才能调用这个组件。组件间数据传入和传出都做验证，防止恶意数据的输入和敏感数据的泄露。

组件权限的检查：防止组件在运行时自身的权限被恶意篡改，如AndroidManifest.xml修改，通过Android API的`context.checkCallingOrSelfPermission`对应用运行时权限进行检查、执行和撤销。

1.2 Activity 安全

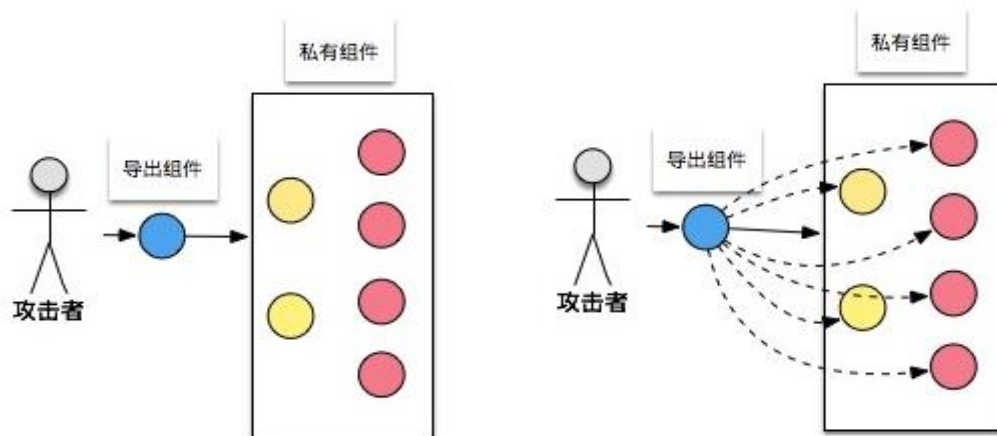
Activity是通过Intent来启动和传输相应数据，如果Intent被恶意程序拦截或者伪造就会存在风险。



1.2.1 私有 Activity

A. 组件调用漏洞

如果存在某个私有组件能被导出组件启动的话，那么这个私有组件其实就不再是私有了。如果攻击者可通过控制导出的组件对私有组件进行控制，那么攻击者的攻击面就大大的扩大了。实例如下：



漏洞原理：存在一个私有组件A和一个对外导出组件B。如果B能够根据对外传入的Intent中的内容打开私有组件A，同时启动私有组件A的Intent的内容来自启动导出组件B的Intent的内容，那么攻击者就可以通过对外导出组件B，去控制私有导出组件A。这就可能会造成严重的安全风险。

```
public class PrivateActivity extends Activity{
    private static final String SAMPLE_STRING_TEST = "sample_string_test";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        String value = getIntent().getStringExtra(SAMPLE_STRING_TEST);
        String subValue = value.substring(0);
        Toast.makeText(getApplicationContext(), subValue,
            Toast.LENGTH_SHORT).show();
    }
}
```

很明显, PrivateActivity是存在问题的, 因为从Intent中直接获取值之后, 没有做任何异常处理。如果PrivateActivity是私有的一个Activity, 并且开发工程师能保证传入到该Activity的Intent一定有价值的话, 那么其实是无法造成威胁的。但是如果存在另外一个MainActivity, 如下:

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Intent intent = this.getIntent();
        if (intent.getBooleanExtra("allowed", false)) {
            ComponentName componentName = new ComponentName(this, "PrivateActivity");
            intent.setComponent(componentName);
            this.startActivity(intent);
        }
    }
}
```

```
adb shell am start -n com.privatecomponent/.MainActivity --ez "allowed" true
```

使用以上命令对APP进行攻击, 应用因为缺失参数导致崩溃。如果PrivateActivity里面存在很重要的逻辑业务处理的话, 那么恶意攻击者可以通过控制MainActivity去控制PrivateActivity, 进而控制PrivateActivity里面的逻辑有可能造成危害。还有种攻击是Intent Scheme URL攻击的本质都是一样的。

安全建议: 对于不必对外的组件设置exported=false。如果该组件需导出, 检查该组件能不能通过intent去启动其他私有组件, 被启动的私有组件需做好安全防范。根据业务严格过滤和校验intent中内容。

1.2.2 公有 Activity

该Activity会用在多个地方启动且数据不定, 任意一个应用程序都可以发送一个Intent来启动它。

安全建议: AndroidManifest.xml中exported=true; 发送或接收Intent消息注意授权和保密。

1.2.3 伙伴 Activity

该Activity只能由伙伴应用使用的Activity且共享信息, 第三方的应用可读取启动它Intent信息。

安全建议: 不添加intent-filter设置exported=true; 使用白名单验证应用签名; 处理Intent小心。

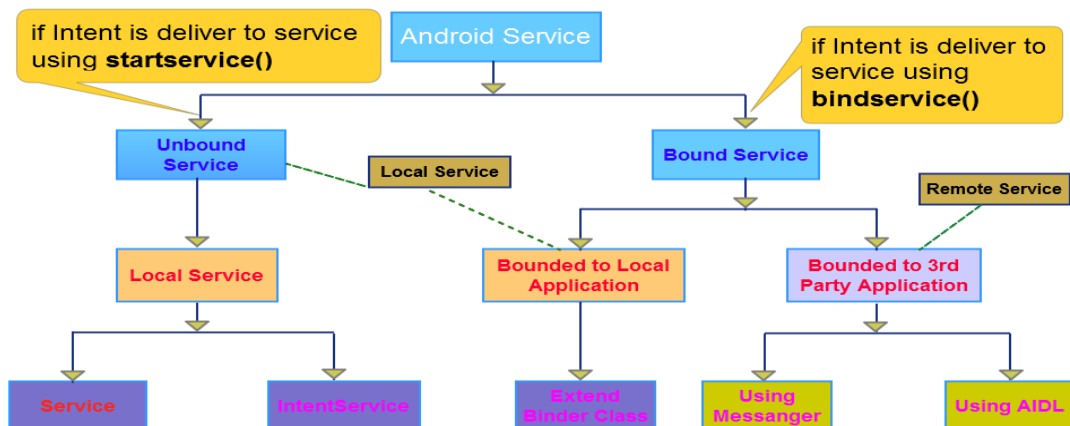
1.2.4 内部 Activity

该Activity只能由自身应用程序内部调用, 在应用程序内部共享信息, 一般设置signature的权限。

安全建议: 定义Activity的signature权限; 不声明intent-filter设置exported=true; 验证签名。

1.3 Service 安全

Service是没有界面且能长时间运行于后台的组件，其他应用的组件可以启动一个服务运行于后台，用户切换到另一个应用时会继续运行。另外一个组件可绑定到Service来交互，交互可以是进程间通信。



1.3.1 私有 Service

该Service无法由其他应用程序启动，开发的时候只要使用显式的Intent就可以保证安全。

安全建议：设置exported=false；使用显式Intent，发送或接收Intent注意信息授权和保密。

1.3.2 公共 Service

该Service没有指定发送方的Service，可以被大多数的应用程序来启动，需要注意恶意Intent。

安全建议：设置exported=true；发送或接收Intent消息时注意授权和保密。

1.3.3 合作 Service

该Service只能由存在伙伴关系的应用程序使用，即合作伙伴公司的不同应用相互调用的Service。

安全建议：设置exported=true；验证合作伙伴的包名和包Signature；

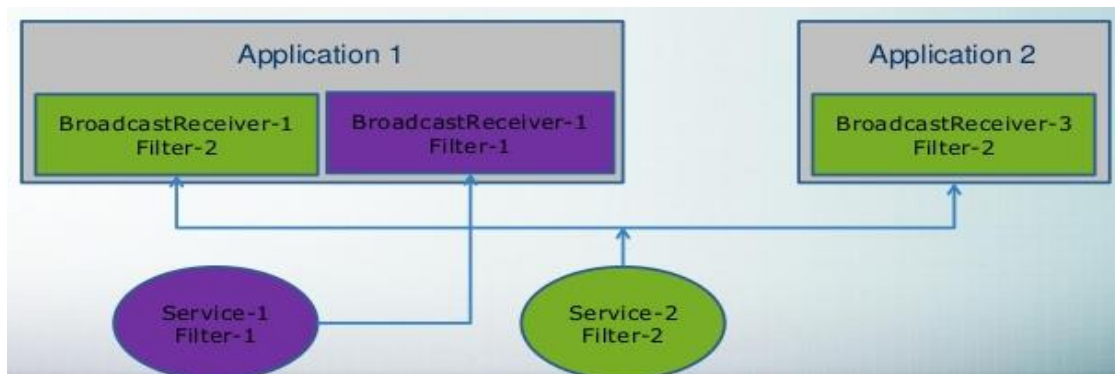
1.3.4 内部 Service

该Service只能够在一个体系内启动，比如系统启动的服务或者公司内不同产品的应用间相互通知。

安全建议：设置export=true，但无Intent-Filter；验证签名和权限；验证Intent消息防止泄露。

1.4 Broadcast 安全

广播可以存在任意数量的接收器，但是这也给广播带来了一定的风险。一些安全性较高的广播被恶意接收器接收并拦截。安全性较高的接收器也可能无法识别虚假广播而造成数据处理失误。



1.4.1 私有 Broadcast

该广播属于静态广播，只用应用程序内部能够接收，是最安全的广播。

安全建议：设置exported=false，无Intent-Filter；接收处理Intent消息注意安全；防止重要信息在Intent中传输，所有处理完毕后需要终止掉广播。

1.4.2 公共 Broadcast

该广播没有确定的接收方，使用时必须注意此类广播被一些应用接收造成的数据泄露。

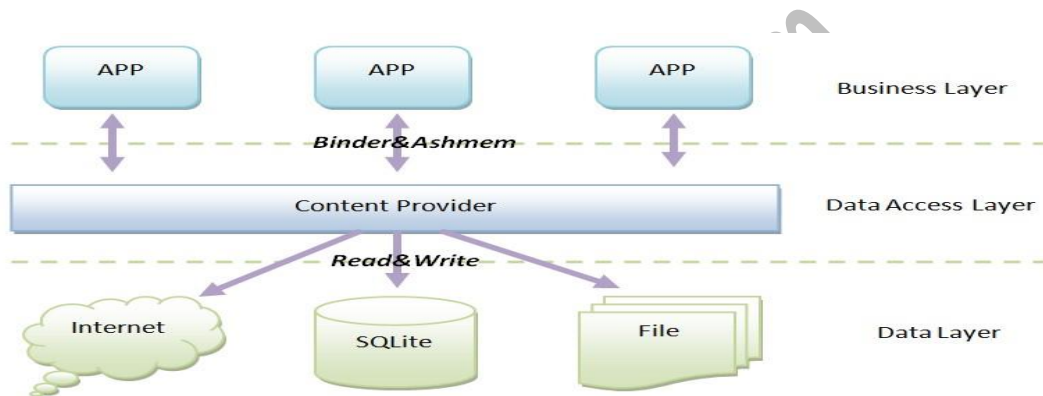
安全建议：设置exported=true；接收处理Intent消息注意安全，Return时防止敏感信息泄露。

1.4.3 内部 Broadcast

该广播除了内部应用能够发送接收其他应用都不能发送接收的广播，相对比较安全。

安全建议：设置exported=true；静态、动态广播注册声明Signature Permission来发送、接收。

1.5 Provider 安全



1.5.1 私有 Provider

该Provider仅仅在单一应用程序中，是最安全的Content Provider。

安全建议：设置exported=false；只能在同一个应用程序里进行敏感信息发送和接收。

1.5.2 公共 Provider

该Provider没有指定接收者，可能存在多个接收方，被恶意软件攻击和破坏。

安全建议：设置exported=true；发送或接收时注意输入验证和敏感信息返回的检查。

1.5.3 合作 Provider

该Provider是合作关系的APP可以进行，根据包名和签名进行校验。

安全建议：设置exported=true；需要校验包名和签名才能访问；发送或接收信息需要进行验证。

1.5.4 内部 Provider

该Provider只允许具有相同签名的内部应用使用，即同一个系统中过个产品共享的Provider。

安全建议：设置exported=true，permission=Signature；检查包名和签名；注意信息泄露。

1.5.5 部分 Provider

只开放部分Content临时访问，grant-uri-permission、read|writePermission来访问某URI。

安全建议：设置exported=false；指定URI临时路径的访问权限；注意信息部分开放的要防止泄露。

1.6 组件安全问题

常见安全问题：无意间暴露了内部组件，或被其他APP访问到。共享数据没有设置合理的权限限访问。

1.6.1 组件调用安全

基于回答开发的组件用途、保护的、接收到的恶意Intent、其数据被访问的安全风险，保护APP组件的两个安全建议是：正确配置AndroidManifest.xml、代码层级上进行权限检查。

正确配置AndroidManifest.xml：检查android:exported属性定义，看组件是否允许被其他APP调用，如果它不需要被其他APP调用或者它需要明确地与Android系统其他部分组件的交互隔离开来，则应该在该组件的XML元素中加入 `<componentName android:exported="false"></componentName>` 属性。

componentName为activity时：外部APP组件不能用startActivity或startActivityForResult。

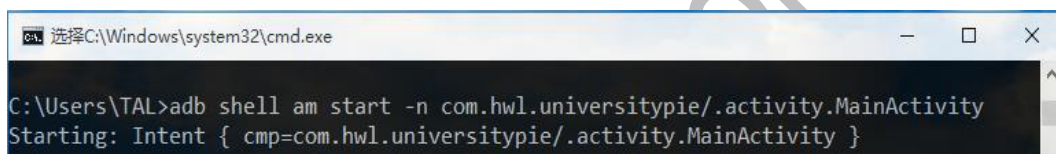
componentName为service时：外部APP不能绑定bindService()或startService()相应service。

componentName为provider时：外部APP组件被限制向该组件发送广播Intent。

componentName为broadcast时：外部APP组件被限制访问该content provider的数据。

【特别说明】：组件的XML中设置的android:permission属性会覆盖<application>元素中该属性。若组件中定义了<intent-filter>，则需要明确设置android:exported="false"，否则默认值为true。

调用逻辑问题：因为Activity属性为exported=true，通过am start可直接调用，绕过了登录逻辑。



<-需要登录 | 绕过登录->



定制组件的保护权限：Android中定义了一系列用于保护组件的默认权限，但是还需要做出定制调整。

A. 先在res/values/strings.xml中声明permission标签作为定制组件的权限：

```
<string name="custom_permission_label">Custom Permission</string>
```

B. 在AndroidManifest.xml文件中添加Permission实现对APP组件的权限保护：

```
<permission
    android:name="android.permission.CUSTOM_PERMISSION"
    android:protectionLevel="normal"
    android:description="Custom permission"
    android:label="@string/custom_permission_label">
```

C. 添加该权限到组件的android:permission属性中，使其和其他权限一样正常地工作：

```
<activity|service|provider|receiver
```



```
android:permission="android.permission.CUSTOM_PERMISSION">
```

```
</activity|service|provider|receiver>
```

D. 其他APP请求使用该定制权限时可以在其AndroidManifest.xml中添加<uses-permission>:

```
<uses-permission android:name="android.permission.CUSTOM_PERMISSION" />
```

【特别说明】: 标记为normal权限的安装时自动赋予应用, 需组件间共享的应设置signature权限。

1.6.2 组件通信安全

Intent是执行动作的抽象描述, 用来启动activities、service和broadcast, 协助完成组件间通讯。当应用发送一个广播意图, 对于隐式intent没有指明接收方-所有的应用都会收到, 系统会根据信息为该Intent选择出最匹配组件来启动。另外, 当注册了一个接收器, 它是可能从恶意的程序那里接收到广播。如果开发人员没有限制谁能广播谁不能广播, 如果接收器接收到了不信任源发来的广播, 应用会陷入危险。恶意程序指定action标识后可获取intent内容, 导致数据泄露、劫持、钓鱼等风险。示例如下:

A. 广播接收器的AndroidManifest.xml代码如下:

```
<receiver android:name="MyBroadCastReceiver" android:exported="true">
    <intent-filter>
        <action android:name="MyBroadCast"></action>
    </intent-filter>
</receiver>
```

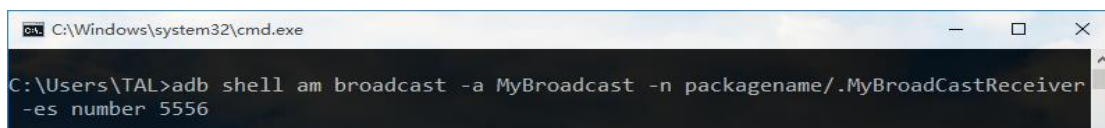
B. 广播来的手机号码通过短信发送给注册的用户, 代码如下:

```
public class MyBroadCastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        if (extras != null) {
            if (extras.containsKey("number")) {
                Object num = extras.get("number");
                String phoneNo = num.toString();
                String message = "Hi, Your Registration has been confirmed!";
                SmsManager sms = SmsManager.getDefault();
                Sms.sendTextMessage(phoneNo, null, message, null, null);
            }
        }
    }
}
```

C. 一般Intent广播方式:

```
Intent intent = new Intent();
intent.setAction("MyBroadCast");
intent.putExtra("number", input);
sendBroadcast(intent);
```

D. 尝试发送一些虚假的广播进行攻击: 通过am broadcast 加上-es选项传递电话号码即可。



安全建议: 显示调用Intent; Intent.setPackage、Intent.setComponent、Intent.setClassName、Intent.setClass、new Intent(context, Receivered.class)都需要明确指定目标接收方来显示调用。

1.6.3 共享数据安全

为了保护Content provider和数据库免受SQL注入攻击的影响，可在AndroidManifest.xml中加入特定的配置对Content provider在URI级别进行访问限制，同时实现对Content provider中URI的保护。

Provider的URI访问控制：通过读写分离、URI设置细化权限。

A. 首先在APP中向系统申请两个权限，分别对应读数据库权限和写数据库权限：

```
<permission
    android:name="com.myapp.provider.READ"
    android:label="provider read permission"
    android:protectionLevel="normal" />
<permission
    android:name="com.myapp.provider.WRITE"
    android:label="provider write permission"
    android:protectionLevel="normal" />
```

B. 在Content provider中，Provider的URI权限设置如下：

```
<provider
    android:name=".PrivateProvider"
    android:authorities="com.myapp.provider.PrivateProvider"
    android:readPermission="com.myapp.provider.READ"
    android:exported="true"
    <path-permission
        android:pathPrefix="/unsecured"
        android:readPermission="com.myapp.provider.READ" />
</provider>
```

在Content provider中，Provider的granting授予读、写权限时：

```
<provider
    android:name=".PrivateProvider"
    android:authorities="com.myapp.provider.PrivateProvider"
    android:exported="true"
    android:readPermission="com.myapp.provider.READ"
    android:writePermission="com.myapp.provider.WRITE"
    <grant-uri-permission android:pathPrefix="/unsecured"/>
</provider>
```

C. 使用第三方应用同时申请使用这两个权限才可以对数据库读写，即：

```
<uses-permission android:name="com.myapp.provider.READ"/>
<uses-permission android:name="com.myapp.provider.WRITE"/>
```

SQL注入防护：避免使用SQLiteDatabase.rawQuery()来改变一个参数化的语句，使用预编译语句如SQLiteStatement函数，提供对参数的绑定(binding)和转义(escaping)，以防御对SQL注入的攻击。或用SQLiteDatabase类的query、insert、update和delete方法，它们提供了字符串数组参数化的语句。同时，这样还可以使得数据库每次执行语句时都不再需要进行SQL解析，SQL查询的性能会提升。

```
String sql = "INSERT INTO userInfo(name, age) VALUES(?, ?)";
SQLiteDatabase db = getWritableDatabase();
SQLiteStatement statement = db.compileStatement(sql);
statement.bindString(1, "Mr.Wang");
statement.bindLong(23);
statement.executeInsert();
```

2. 软件安全分析

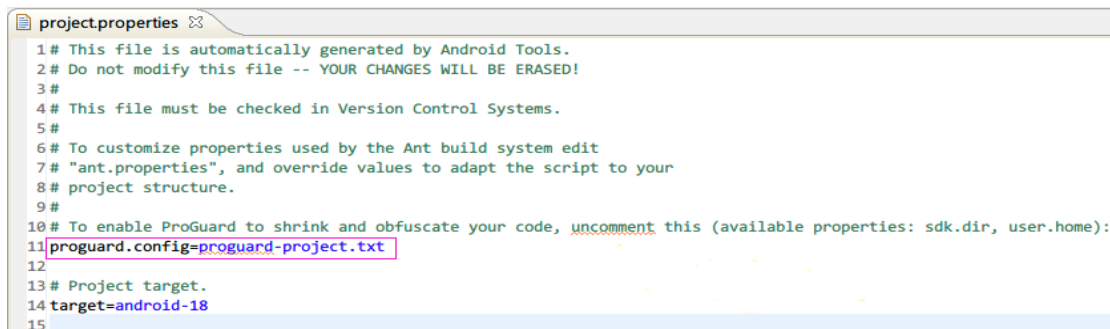
2.1 防止逆向技术

防止被逆向可分为：代码混淆、签名校验、动态检测、Java代码Native化、逆向工具保护技术等。

2.1.1 代码混淆技术

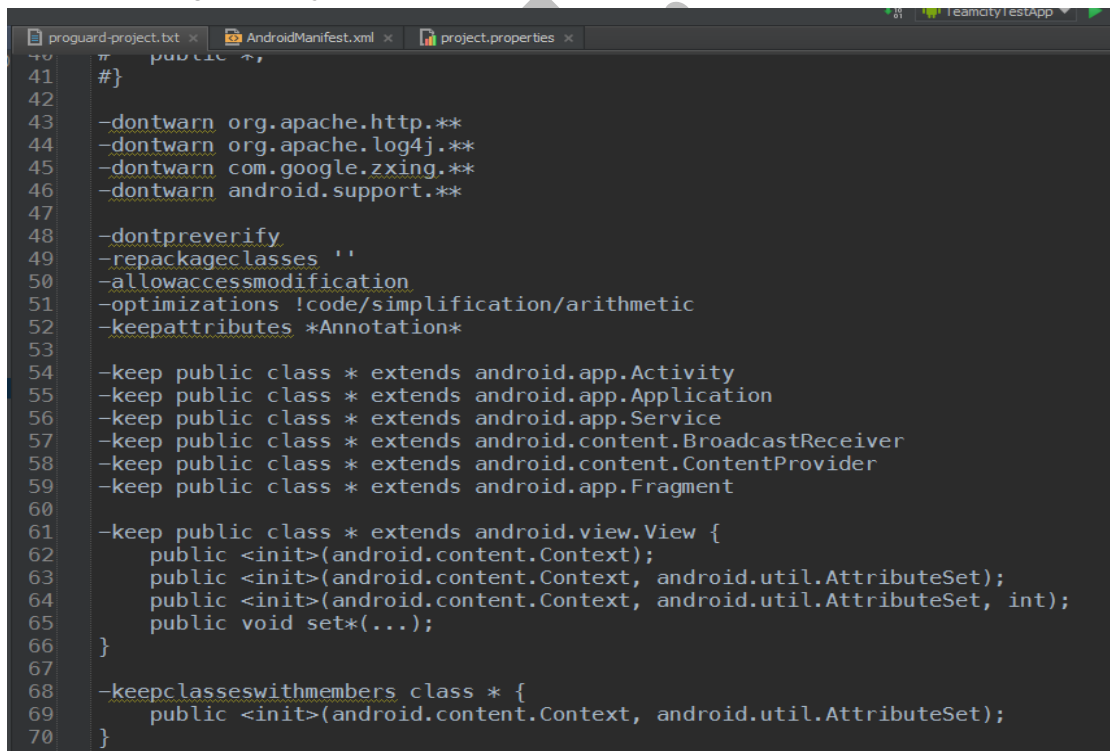
代码混淆是指将程序源代码、编译中间码转成功能上等价、形式上复杂难阅读的行为。Android平台默认使用ProGuard工具进行代码混淆，它可以压缩、优化和混淆Java字节码文件实现保护。具体包括：删除不可见字符、注释无用代码和log，创建紧凑的代码文档；重命名变量和函数；删除来自源文件中的没有调用的代码；充分利用Java6的快速加载的优点来提前检测和返回Java6中存在的类文件。

Android的SDK中包含了ProGuard，Eclipse插件生成的默认配置文件project.properties如下：



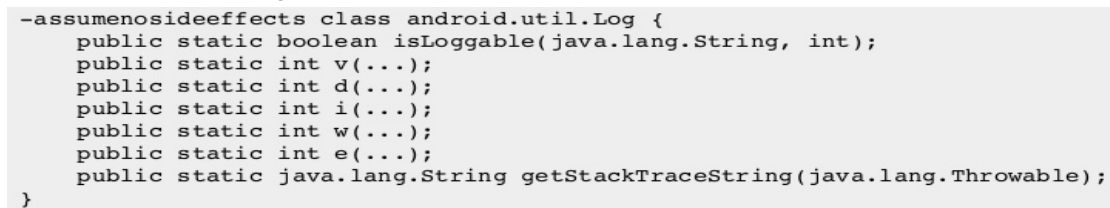
```
1# This file is automatically generated by Android Tools.
2# Do not modify this file -- YOUR CHANGES WILL BE ERASED!
3#
4# This file must be checked in Version Control Systems.
5#
6# To customize properties used by the Ant build system edit
7# "ant.properties", and override values to adapt the script to your
8# project structure.
9#
10# To enable ProGuard to shrink and obfuscate your code, uncomment this (available properties: sdk.dir, user.home):
11proguard.config=proguard-project.txt
12
13# Project target.
14target=android-18
15
```

通过配置proguard-project.txt的默认文件如下：



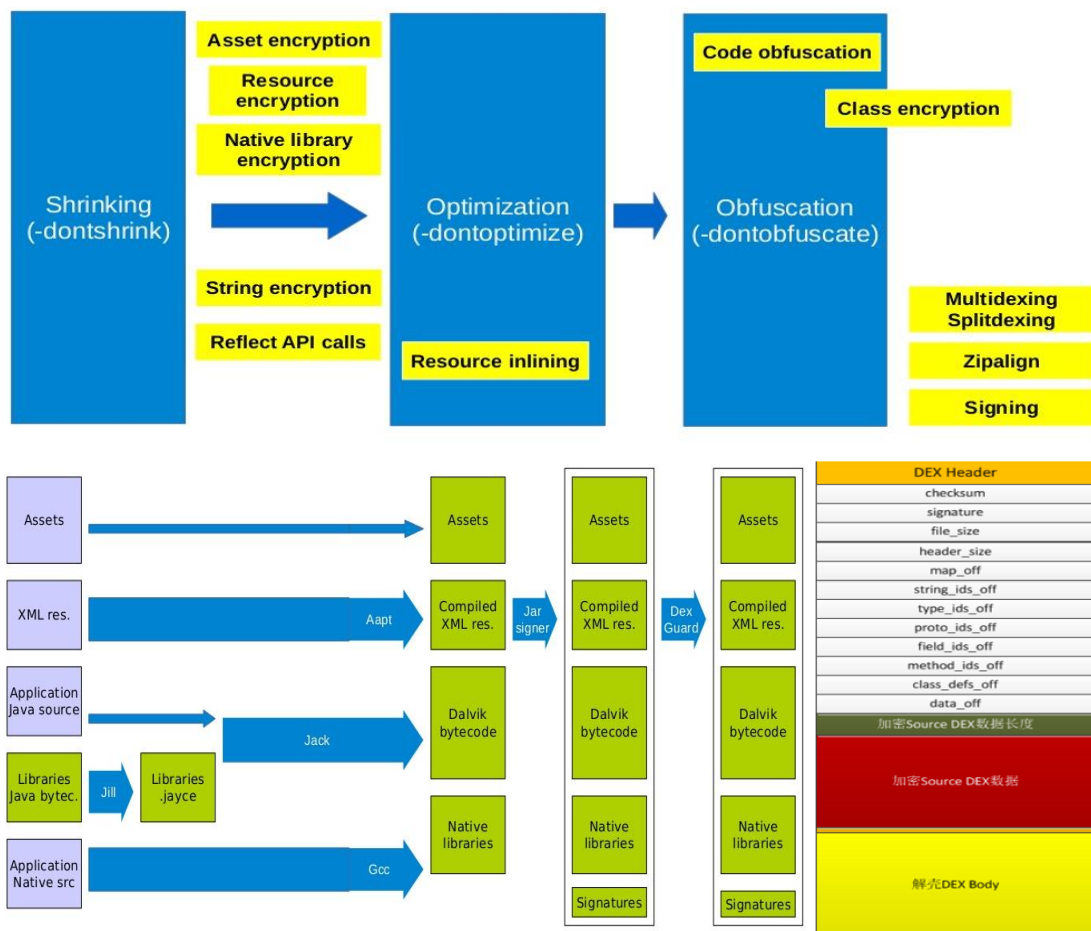
```
41#}
42
43-dontwarn org.apache.http.**
44-dontwarn org.apache.log4j.**
45-dontwarn com.google.zxing.**
46-dontwarn android.support.**
47
48-dontpreverify
49-repackageclasses ''
50-allowaccessmodification
51-optimizations !code/simplification/arithmetic
52-keepattributes *Annotation*
53
54-keep public class * extends android.app.Activity
55-keep public class * extends android.app.Application
56-keep public class * extends android.app.Service
57-keep public class * extends android.content.BroadcastReceiver
58-keep public class * extends android.content.ContentProvider
59-keep public class * extends android.app.Fragment
60
61-keep public class * extends android.view.View {
62    public <init>(android.content.Context);
63    public <init>(android.content.Context, android.util.AttributeSet);
64    public <init>(android.content.Context, android.util.AttributeSet, int);
65    public void set*(...);
66}
67
68-keepclasseswithmembers class * {
69    public <init>(android.content.Context, android.util.AttributeSet);
70}
```

如果需要删除无用Log可以配置以下优化，同时注释掉 -dontoptimize【不优化输入的类文件】。



```
-assumenosideeffects class android.util.Log {
    public static boolean isLoggable(java.lang.String, int);
    public static int v(...);
    public static int d(...);
    public static int i(...);
    public static int w(...);
    public static int e(...);
    public static java.lang.String getStackTraceString(java.lang.Throwable);
}
```

通过使用动态的注入方法可以将自定义逻辑注入到DEX文件后进行重新编译。动态加载技术是为了对抗一些常见的逆向工具，使得逆向过程产生崩溃和无法理解的信息。DexGuard可以提供DEX动态分割、Class加密、字符串加密、API加密等，通过一个无序而且无法理解的数组将字符串、Class、API进行替换实现。



开源DEX混淆器有dalvik-obfuscator、脱壳软件有android-unpacker。对于SO文件的加壳即实现ELF文件的加密,通过将核心代码放入加密的.jar或APK文件中,用Native C/C++进行解密并完成完整性校验。

2.1.2 证书签名校验

A. Java层签名校验: 可以通过逆向smali代码破解。

```
public class CheckApplication extends Application {
    private static final String SIGNATURE_MD5 = "FE13E7B234B495690P0980UYKH760KI34";
    @Override
    Public void onCreate() {
        String getMd5 = getSignatrueMd5(this);
        if (!SIGNATURE_MD5.equals(getMd5)) {
            System.exit(0);
        }
    }
    public static String getSignatureMd5(Context context) {
        String rString = "";
        try {
            PackageInfo mPackageInfo = context.getPackageManager().getPackageInfo
(context.getPackageName(), PackageManager.GET_SIGNATURES);
```

```

        Byte[] arrayOfByte = mPackageInfo.signature[0].toByteArray();
        rString = DigestUtil.md5(arrayOfByte);
    } catch (Exception e) {
        e.printStackTrace();
    }
    Return rString;
}
}

```

B. NDK层签名校验：可以通过IDA PRO破解。

```

#include <jni.h>
#include <strings.h>
#include "top_goluck_util_KeyUtil.h"
const char *RELEASE_SIGN = "08201dXXXX";
JNIEXPORT jstring JNICALL Java_top_goluck_util_KeyUtil_getKey(JNIEnv *env, jobject
object, jobject contextObject, jint type) {
    jclass native_class = env->GetObjectClass(contextObject);
    jmethodID pm_id = env->GetMethodID(native_class, "getPackageManager",
"()Landroid/content/pm/PackageManager;");
    jobject pm_obj = env->CallObjectMethod(contextObject, pm_id);
    jclass pm_clazz = env->GetObjectClass(pm_obj);
    jmethodID package_info_id = env->GetMethodID(pm_clazz, "getPackageInfo",
"(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;");
    jclass native_classss = env->GetObjectClass(contextObject);
    jmethodID mId = env->GetMethodID(native_classss,
"getPackageName", "()Ljava/lang/String;");
    jstring pkg_str = static_cast<jstring>(env->CallObjectMethod(contextObject, mId));
    jobject pi_obj = env->CallObjectMethod(pm_obj, package_info_id, pkg_str, 64);
    jclass pi_clazz = env->GetObjectClass(pi_obj);
    jfieldID signatures_fieldId = env->GetFieldID(pi_clazz, "signatures",
"[Landroid/content/pm/Signature;");
    jobject signatures_obj = env->GetObjectField(pi_obj, signatures_fieldId);
    jobjectArray signaturesArray = (jobjectArray) signatures_obj;
    jsize size = env->GetArrayLength(signaturesArray);
    jobject signature_obj = env->GetObjectArrayElement(signaturesArray, 0);
    jclass signature_clazz = env->GetObjectClass(signature_obj);
    jmethodID string_id = env->GetMethodID(signature_clazz, "toCharsString",
"()Ljava/lang/String;");
    jstring str = static_cast<jstring>(env->CallObjectMethod(signature_obj,
string_id));
    char *c_msg = (char *) env->GetStringUTFChars(str, 0);
    if (strcmp(c_msg, RELEASE_SIGN) == 0){
        return True;
    }
    return False;
}

```

2.1.3 动态加载检测

A. 模拟器的检测: 通过设备imei和Build.MODE来判断其是否是模拟器, 是则退出, 防止被恶意调试。

```
public static Boolean CheckDeviceIDS(Context context){
    try {
        TelephonyManager telephonyManager =
        (TelephonyManager)context.getSystemService(Context.TELEPHONY_SERVICE);
        String device_ids =telephonyManager.getDeviceId();
        if (know_deviceid != null && know_deviceid.equalsIgnoreCase("000000000000000"))
            return true;
        return (Build.MODEL.equals("sdk") || (Build.MODEL.equals("google_sdk")));
    } catch (Exception ioe) {
        return true;
    }
    return false;
}
```

B. 检查IDA PRO调试: 查看IDA、GDB等进程如android_server、gdbserver来判断是否在调试中。

```
shell@android:/ $ ps | busybox grep server
ps | busybox grep server
drn      126   1   11300  1668  ffffffff 00000000 $ /system/bin/drmserver
system   131   1   18220  2476  ffffffff 00000000 $ /system/bin/faradioserver
root     391   2     0     0  ffffffff 00000000 $ krpserverd
shell    13494 13472 1300   612  c05d77d0 40185604 $ ./android_server
media    15926 1   33932  8172  ffffffff 00000000 $ /system/bin/mediaserver
system   16050 15928 577476 59632  ffffffff 00000000 $ system_server
```

也可以通过检测服务监听端口判断, 如IDA会启动android_server, 它将监听在TCP的23946端口。

由TracePid判断, 通过/proc/pid/cmdline获取进程名, 一般存在/data/local/tmp/android_server。

```
root@android:/ # ps | grep com.vao
u0_a48    1070   37   188400 28640  ffffffff 40037ebc S com.yaotong.crackme
root@android:/ # cat /proc/1070/status
Name: yaotong.crackme
State: S (sleeping)
Tgid: 1070
Pid: 1070
PPid: 37
TracerPid: 0 非0非自身PID即被调试
Uid: 10048 10048 10048 10048
Gid: 10048 10048 10048 10048
FDSize: 256
Groups: 1028 50048
VmPeak: 188400 kB
VmSize: 188400 kB
VmLck: 0 kB
VmHWM: 28648 kB
VmRSS: 28648 kB
VmData: 15628 kB
VmStk: 84 kB
VmExe: 8 kB
VmLib: 29320 kB
VmPTE: 100 kB
```

检测代码: 在JNI_Onload中新建一个线程不停地进行检查该进程下状态文件中TracePid的内容:

```
void check_tracerpid()
{
    int pid = getpid();
    int bufsize = 256;
    char filename[bufsize];
```



```
char line[bufsize];
int tracerpid;
FILE *fp;
sprintf(filename, "/proc/%d/status", pid);
fp = fopen(filename, "r");
if (fp != NULL) {
    while (fgets(line, bufsize, fp)) {
        if (strstr(line, "TracerPid") != NULL) {
            if (atoi(&line[10]) != 0)
                kill(pid, SIGKILL);
            break;
        }
    }
    fclose(fp);
}
```

C. 检查GDB的调试: ptrace自身或子进程相互ptrace, 在JNI_Onload最开始加上这个函数即可:

```
void anti_ptrace(void) {
    ptrace(PTRACE_TRACEME, 0, 0, 0);
}
```

备注: PTRACE_TRACEME表示本进程被自身进程ptrace。因为一个进程只能被附加一次, 如果应用已被自身进程附加, 后面的调试附加就会失败。

D. 检查DEBUG模式: 调用Android中的flag属性: ApplicationInfo.FLAG_DEBUGGABLE 判断:

```
public static boolean isDebuggable(Context context) {
    if ((context.getApplicationInfo().flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0){
        android.os.Process.killProcess(android.os.Process.myPid());
        return true;
    }
    if (android.os.Debug.isDebuggerConnected()) {
        android.os.Process.killProcess(android.os.Process.myPid());
        return true;
    }
    return false;
}
```

备注: 为了防止在AndroidManifest.xml中添加: android:debuggable=true属性对应用进行调试。

E. 检查执行时间: 动态调试的时候关键代码的前后时间差比正常执行的时候要大多, 因此可以计算代码执行时间, 如果超出一般正常情况下设定值, 就认为代码被调试。

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
void check_time()
{
    int pid = getpid();
    struct timezone tz;
    struct timeval t1, t2;
    gettimeofday(&t1, &tz);
    gettimeofday(&t2, &tz);
}
```

```

    if ((t2.tv_sec) - (t1.tv_sec) > 1) {
        int ret = kill(pid, SIGKILL);
        return ;
    }
}

```

F. 检查so注入：调试时调试器会设置断点，需要调用ptrace函数才能进行so注入。所以，防止恶意程序对应用进行动态注入的话，可以对当前进程的so进行排查，检测存在白名单外的so即可认为被注入了。

```

public static boolean isHooked(int pid, String pkg)
{
    File file = new File("/proc/" + pid + "/maps");
    if (!file.exists())
        return false;
    try {
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(new
FileInputStream(file)));
        String lineString = null;
        while ((lineString = bufferedReader.readLine()) != null) {
            String tempString = lineString.trim();
            String pkgsString = "/data/data/" + pkg;
            if (tempString.contains("/data/data") && !tempString.contains(pkgsString))
                return false;
        }
        bufferedReader.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return true;
}

```

G. 检查内存Dump：在动态调试的过程中，一般会查看调试进程的虚拟地址空间或者是dump内存，这时就会涉及到对文件的读写以及打开的权限，这时候对它们进行检测就能发现是否正在被破解。Linux下的Inotify就可以实现对文件系统事件的打开，读写的监管。若通过Inotify收到事件变化就Kill掉进程。

```

void check_inotify()
{
    int pid = getpid();
    int ret, len, i, fd, wd;
    const int MAXLEN = 2048;
    char buf[1024], readbuf[MAXLEN];
    fd_set readfds;
    fd = inotify_init();
    sprintf(buf, "/proc/%d/maps", pid);
    wd = inotify_add_watch(fd, buf, IN_ALL_EVENTS);
    if (wd >= 0) {
        while (1) {

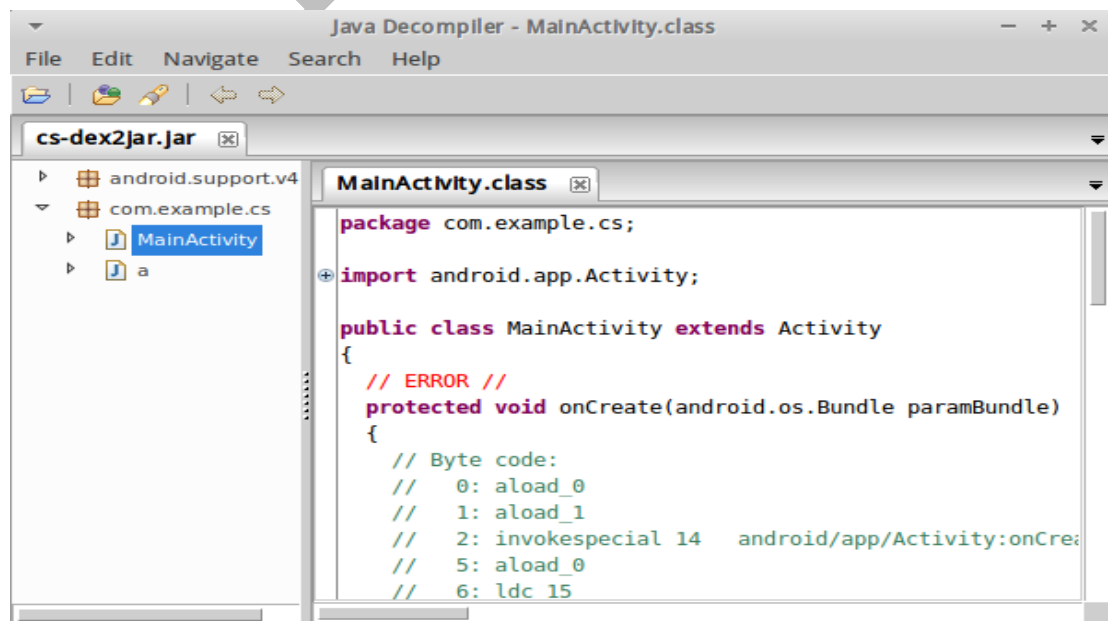
```

```
i = 0;
FD_ZERO(&readfds);
FD_SET(fd, &readfds);
ret = select(fd + 1, &readfds, 0, 0, 0);
if (ret == -1)
    break;
if (ret) {
    len = read(fd, readbuf, MAXLEN);
    while (i < len) {
        struct inotify_event *event = (struct inotify_event *) &readbuf[i];
        if ((event->mask & IN_ACCESS) || (event->mask & IN_OPEN)) {
            int ret = kill(pid, SIGKILL);
            return;
        }
        i += sizeof(struct inotify_event) + event->len;
    }
}
}
inotify_rm_watch(fd, wd);
close(fd);
}
```

H. 检查断点设置：调试时调试器会设置断点，首先会保存目标地址上的数据，然后将目标地址上的头几个字节替换为breakpoint指令，命中断点触发breakpoint，这时程序向操作系统发送SIGTRAP信号，调试器收到SIGTRAP信号后，调试器会回退被跟踪进程的当前PC值，当控制权回到原进程时，PC就指向了断点所在位置。所以，可以直接进行检测文件，遍历so中可执行segment查找是否出现breakpoint指令即可。

2.1.4 逆向工具保护

为了对抗代码查看工具的使用，可以利用工具本身的BUG，如果利用JD-GUI工具的BUG，可以在代码中添加一些无意义的代码段-如不可能的特殊分支switch语句，使得工具在使用过程中产生崩溃。如下：



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    switch(0)
    {
        case 1:
            JSONObject jsonObj;
            String date=null;
            String second=null;
            try
            {
                jsonObj=new JSONObject();
                date=jsonObj.getString("date");
                second=jsonObj.getString("second");
            }
            catch(JSONException e)
            {
                e.printStackTrace();
            }
            test.settime(date,second);
            break;
    }
}
```

```
class test
{
    public static void settime(String a,String b){}
}
```

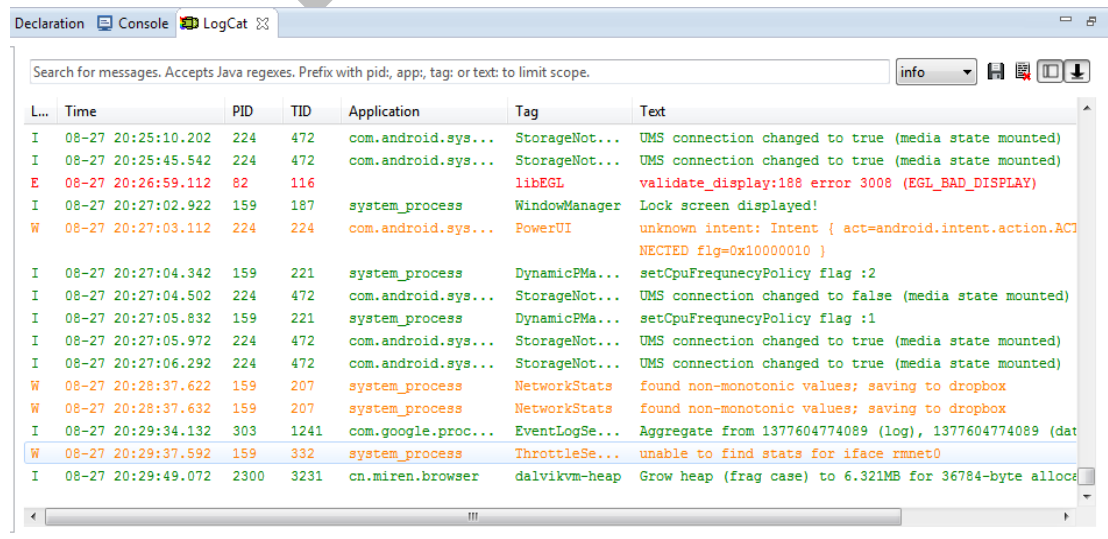
2.2 动态分析技术

Android平台上的动态分析主要是指使用一些调试工具：DDMS、Andbug、IDA PRO等工具，调试分析应用程序中LOG、文件、内存和通信信息。一般动态分析前对APK进行静态分析，打开调试开关后二次打包。

许多应用上线后忘记关闭系统的Log，通过Logcat等查看信息可以发现程序逻辑，从而发现安全隐患。

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.github.orangegangsters.lollipin" platformBuildVersionCode="23" platformBuildVersionName="6.0-2166767">
    <application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/icon" android:label="@string/app_name" android:name="com.github.orangegangsters.lollipin.CustomApplication" android:theme="@style/AppTheme">
        <activity android:label="@string/app_name" android:name="com.github.orangegangsters.lollipin.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

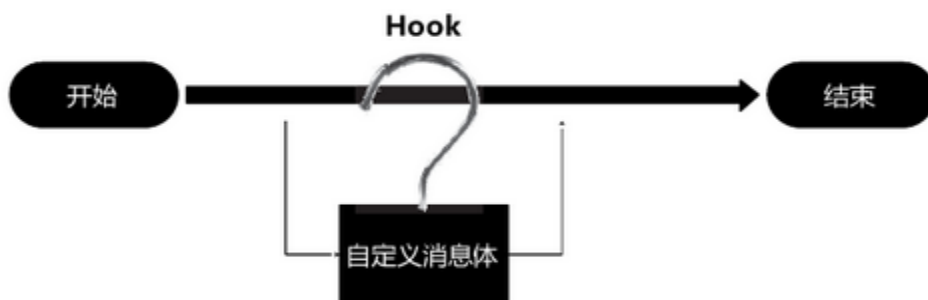
通过DDMS或者用Logcat查看应用的Log，然后根据PID、Application、Text等过滤如cookie信息。



安全建议：声明READ_LOGS权限的一般是恶意应用，关闭Log输出即设置android:debuggable="false"。许多应用都没有禁用掉缓存，其实可以通过"no-cache"，"no-store"等HTTP头来避免造成信息泄露。

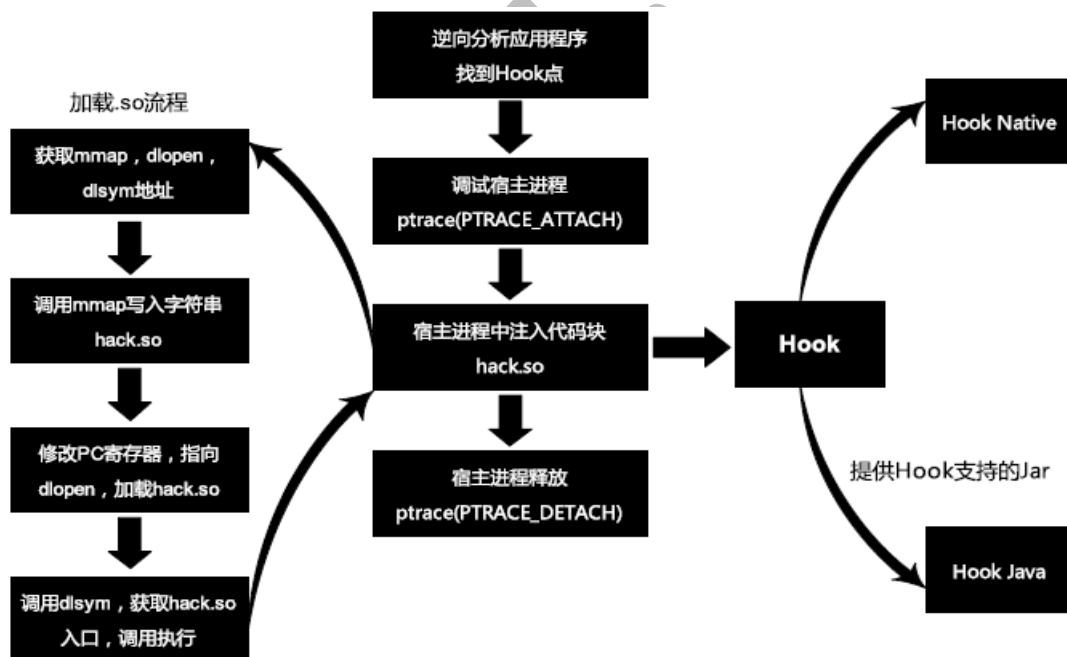
2.3 Hook 技术

Hook技术的本质就是劫持函数调用。但是由于处于Linux用户态，每个进程都有自己独立的进程空间，所以必须先注入到所要Hook的进程空间，修改其内存中的进程代码，替换其过程表的符号地址。在Android中一般是通过ptrace函数附加进程，然后向远程进程注入so库，从而达到监控以及远程进程关键函数挂钩。Hook技术的难点在于如何找到函数的入口点、替换函数，涉及理解函数的连接与加载机制。在Android平台上的Hook分为两种：Java层级的Hook，Native层级的Hook。两种模式下，通常能够通过使用JNI机制来进行调用。但是能够在Java层级完成的事基本也不会在Native层去完成。



代码注入（Hook API）是通过mmap函数分配一段临时的内存来完成代码的存放，目标进程中的mmap函数地址的寻找与Hook API函数地址的寻找，则需要通过目标进程的虚拟地址空间与ELF文件的解析。

注入动态链接库可以通过Android的内核函数ptrace让PC指向LR堆栈，动态地attach（跟踪一个目标进程）、detach（结束跟踪一个目标进程）、peektext（获取内存字节）、pocketext（向内存写入地址）进程。通过Android的内核函数dlopen，能够以指定模式打开指定的动态链接库文件注入至目标进程中。



查看当前Activity: adb shell dumpsys activity输出组件和系统状态信息。

```
C:\Windows\system32\cmd.exe
C:\Users\TAL>adb shell dumpsys activity
ACTIVITY MANAGER PENDING INTENTS (dumpsys activity intents)
* PendingIntentRecord{4a8f8f1c com.google.android.gms startService}
* PendingIntentRecord{4a89eb3c android broadcastIntent}
* PendingIntentRecord{4a9657a8 com.estrongs.android.pop broadcastIntent}
* PendingIntentRecord{4a8f2e54 com.android.providers.calendar broadcastIntent}
* PendingIntentRecord{4a91c284 android broadcastIntent}
* PendingIntentRecord{4a981e14 com.google.android.gms broadcastIntent}
* PendingIntentRecord{4a91c650 com.android.vending startService}
* PendingIntentRecord{4a935c20 android startActivity}
* PendingIntentRecord{4a783458 android broadcastIntent}
* PendingIntentRecord{4a94b210 com.google.android.gms startService}
* PendingIntentRecord{4a9643d4 com.google.android.gms startService}
* PendingIntentRecord{4a911638 com.google.android.gms startService}
```


Java层Hook检测：使用ps命令查看一下浏览器应用（com.android.browser）的进程pid，在adb shell模式下输入ps | busybox grep com.android.browser（busybox是一个扩展的命令工具）。

```
shell@android:/ $ ps | busybox grep com.android.browser
ps | busybox grep com.android.browser
u0_a4      5425  3935  550580 78260 ffffffff 00000000 $ com.android.browser
```

访问的/proc目录查看在Android内核空间和用户空间之间进行通信，能够看到当前进程的一些状态信息。而其中的maps文件又能查看进程的虚拟地址空间是如何使用的。查看地址空间中的对应的dex文件有哪些是非系统应用提供的，即过滤出/data@app（系统应用是/system@app）中的：

```
shell@android:/ # cat /proc/5425/maps | busybox grep /data/dalvik-cache/data@app
grep /data/dalvik-cache/data@app
5a6bb000-5a789000 r--p 00000000 b3:14 38549 /data/dalvik-cache/data@app@com.example.hookad-1.apk@classes.dex
5a789000-5a78a000 r--p 0000e000 b3:14 38549 /data/dalvik-cache/data@app@com.example.hookad-1.apk@classes.dex
5a78a000-5a7ba000 r--p 000cf000 b3:14 38549 /data/dalvik-cache/data@app@com.example.hookad-1.apk@classes.dex
5a7ba000-5aa58000 r--p 00000000 b3:14 38551 /data/dalvik-cache/data@app@com.example.loginhook-2.apk@classes.dex
5aa58000-5aa59000 r--p 0000e000 b3:14 38551 /data/dalvik-cache/data@app@com.example.loginhook-2.apk@classes.dex
5aa59000-5abf3000 r--p 0007f000 b3:14 38551 /data/dalvik-cache/data@app@com.example.loginhook-2.apk@classes.dex
5abf3000-5cf1b000 r--p 00000000 b3:14 38521 /data/dalvik-cache/data@app@com.saurik.substrate-1.apk@classes.dex
5cf1b000-5cf5d000 r--p 00041000 b3:14 38521 /data/dalvik-cache/data@app@com.saurik.substrate-1.apk@classes.dex
5cf5d000-5cfbf000 r--p 00042000 b3:14 38521 /data/dalvik-cache/data@app@com.saurik.substrate-1.apk@classes.dex
```

native层Hook检测：使用ps | busybox grep com.example.testndklib。

```
shell@android:/ $ ps | busybox grep com.example.testndklib
ps | busybox grep com.example.testndklib
u0_a87     15097 13979 485956 32596 ffffffff 00000000 $ com.example.testndklib
```

得到其对应的进程pid为15097，直接查看15097进程中的虚拟地址空间加载了哪些第三方的库文件，即过滤出/data/data目录中的，具体命令如图所示。

```
shell@android:/ # cat /proc/15097/maps | busybox grep /data/data/
```

得到的输出结果如图所示，发现其中多了很多的/com.saurik.substrate下面的动态库，说明该进程也已经被其注入了。所以判断com.example.testndklib应用程序已经被Hook，存在不安全的隐患。

```
/data/data/com.example.hooknative/lib/libHookNative.cy.so
/data/data/com.example.hooknative/lib/libHookNative.cy.so
/data/data/com.example.hooknative/lib/libHookNative.cy.so
/data/data/com.saurik.substrate/lib/libAndroidBootstrap.so
/data/data/com.saurik.substrate/lib/libAndroidBootstrap.so
/data/data/com.saurik.substrate/lib/libsubstrate-dvm.so
/data/data/com.saurik.substrate/lib/libsubstrate-dvm.so
/data/data/com.saurik.substrate/lib/libAndroidCydia.cy.so
/data/data/com.saurik.substrate/lib/libAndroidCydia.cy.so
/data/data/com.saurik.substrate/lib/libAndroidCydia.cy.so
/data/data/com.saurik.substrate/lib/libDalvikLoader.cy.so
/data/data/com.saurik.substrate/lib/libDalvikLoader.cy.so
/data/data/com.saurik.substrate/lib/libDalvikLoader.cy.so
/data/data/com.saurik.substrate/lib/libAndroidLoader.so
/data/data/com.saurik.substrate/lib/libAndroidLoader.so
/data/data/com.saurik.substrate/lib/libsubstrate.so
/data/data/com.saurik.substrate/lib/libsubstrate.so
/data/data/com.example.hooktestndklib/lib/libnativeHook.cy.so <deleted>
/data/data/com.example.hooktestndklib/lib/libnativeHook.cy.so <deleted>
/data/data/com.example.testndklib/lib/libtestNDKlib.so
/data/data/com.example.testndklib/lib/libtestNDKlib.so
/data/data/com.example.testndklib/lib/libtestNDKlib.so
/data/data/com.example.testndklib/lib/libtestNDKlib.so
```

相关说明：作为应用程序对自身的检测，只需要读取对应的进程的虚拟地址空间目录/proc/pid/maps文件，判断当前进程空间中载入的代码库文件是否存在于自己白名单中的，即可判断自身程序是否被Hook。但是，对于zygote进程来说如果没有Root权限是无法访问其maps文件的，那么也就无法判断Hook与否了。

Hook卸载：扫描/proc/pid/maps目录下所有so库，将自身动态库文件排除，将非自身的动态链接库全都卸载关闭。对于Java我们无法使用dlclose，所以这里我们还是采用了JNI的方式来完成，操作如下：

```
public native void removeHookSo(String soPath);
void Java_com_example_testndklib_MainActivity_removeHookSo(JNIEnv* env, jobject
thiz, jstring path) {
    constchar* chars = env->GetStringUTFChars(path, 0);
    void* handle = dlopen(chars, RTLD_NOW);
    int count = 4;
    int i = 0;
    for (i = 0; i < count; i++) {
        if (NULL != handle) {
            dlclos(handle);
        }
    }
}

public List<String> removeHooks(int pid, String pkg) {
    List<String> hookLibFile = new ArrayList<>();
    File file = new File("/proc/" + pid + "/maps");
    if (!file.exists()) {
        return hookLibFile;
    }
    try {
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader
(new FileInputStream(file)));
        String lineString = null;
        while ((lineString = bufferedReader.readLine()) != null) {
            String tempString = lineString.trim();
            if (tempString.contains("/data/data") && !tempString.contains("/data/data/"
+ pkg)) {
                int index = tempString.indexOf("/data/data");
                String soPath = tempString.substring(index);
                hookLibFile.add(soPath);
                removeHookSo(soPath);
            }
        }
        bufferedReader.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return hookLibFile;
}
```

相关说明：dlclose用于关闭指定句柄的动态链接库，只有其使用计数为0时才会真正被系统卸载。若在卸载动态链接库之前，系统已经保持对其的应用的话是无法卸载的，所以目前来说是很难卸载的干净。

2.4 Rootkit 技术

Android内核级Rootkit可能带来严重的危害，检测前需要确认设备是否已经root，比如通过检查 /system/app/Superuser.apk、/system/bin/su、/system/xbin/su，或者调用exec("su")来判断。

2.4.1 常用检测技术

Linux下的检测方法包括：基于指纹的文件完整性检测、基于文件名和内容特征库Rootkit检测法、基于函数调用时内核的指令执行路径的变化分析。大多数处理器的性能不高，限制了该方法的使用。

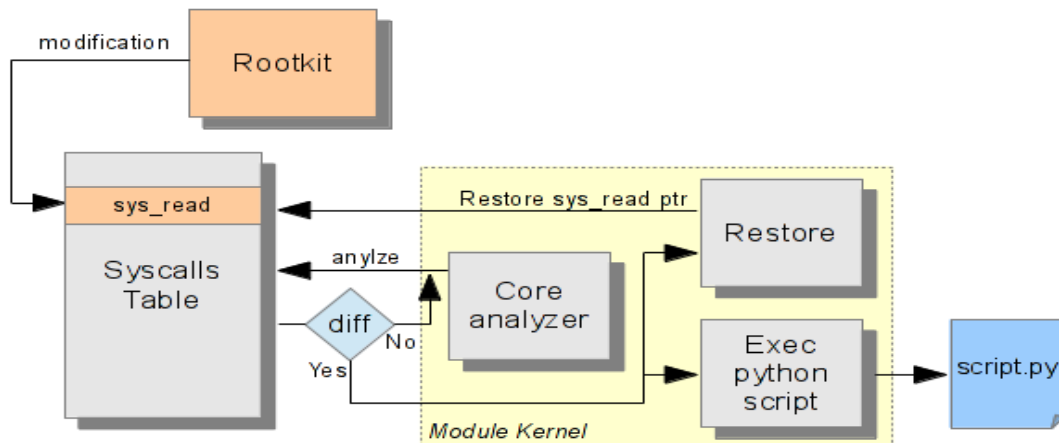
```
xmodulo@ubuntu32: ~  
Volc Rootkit [ Not found ]  
Xzibit Rootkit [ Not found ]  
X-0rg SunOS Rootkit [ Not found ]  
zaRwT.KiT Rootkit [ Not found ]  
ZK Rootkit [ Not found ]  
  
Performing additional rootkit checks  
Suckit Rootkit additional checks [ OK ]  
Checking for possible rootkit files and directories [ None found ]  
Checking for possible rootkit strings [ None found ]  
  
Performing malware checks  
Checking running processes for suspicious files [ None found ]  
Checking for login backdoors [ None found ]  
Checking for suspicious directories [ None found ]  
Checking for sniffer log files [ None found ]  
Checking for Apache backdoor [ Not found ]  
  
Performing Linux specific checks  
Checking loaded kernel modules [ OK ]  
Checking kernel module names [ OK ]  
  
[Press <ENTER> to continue]
```

2.4.2 Android 检测技术

Android 系统的 Rootkit 检测模型基于 LKM 模型实现的，而 LKM 模型工作在系统的内核层。检测模型需要在用户层为用户提供一个操作接口，通过 Proc 文件系统实现用户层和内核层的检测模型的交互。

```
entry = create_proc_entry("checkmod", 700, NULL);  
int mod_read(char *page, char **start, off_t off, int count, int *eof, void *data);  
int mod_write(struct file *fp, const char_user *buff, unsigned long len, void *data);
```

系统调用接口检测可通过判断系统是否遭到 Rootkit 攻击，包括系统调用表地址(sys_call_table)、系统调用表项地址，即检测 sys_call_table 和调用入口地址以及相关函数的地址是否发生改变即可。



基于 LKM 的检测还可以基于模块指纹、特征库检测、模块加载的日志进行监控以发现异常的情况。

3. 存储安全分析

3.1 常见安全问题

存储安全分析主要是对Android应用程序运行过程中产生的文件，如证书文件、私有文件、数据库文件等进行安全验证分析，特别是对明文存储敏感数据，导致信息泄露和内容篡改。

3.2 隐私明文存储

将隐私或者偏好数据明文保存在存储文件或者数据库中。

3.2.1 文件明文存储

在应用的数据目录下存在shared_prefs目录，里面保存了用户的偏好包括账户信息，容易造成泄露。

```
root@vbox86p:/data/data/org.owasp.goatdroid.fourgoats/shared_prefs # ls -al
ls -al
-rw-rw-r-- u0_a99 u0_a99 209 2015-01-14 13:55 credentials.xml
-rw-rw-r-- u0_a99 u0_a99 153 2015-01-14 13:55 destination_info.xml
-rw-rw-r-- u0_a99 u0_a99 148 2015-01-14 13:55 proxy_info.xml
root@vbox86p:/data/data/org.owasp.goatdroid.fourgoats/shared_prefs # cat credentials.xml
credentials.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="password">goatdroid</string>
  <boolean name="remember" value="true" />
  <string name="username">goatdroid</string>
</map>
```

安全建议：SharedPreferences存储的原始数据类型键-值对，可用开源库Secure-Preference加固。通过<https://github.com/scottyab/secure-preferences>下载Secure-preference放入项目中使用。

```
public class MainActivity extends Activity {
    EditText et;
    Button bt;
    SharedPreferences shared;
    @Override
    protect void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        shared = new SecurePreferences(this);
        et = (EditText)findViewById(R.id.editText);
        bt = (Button)findViewById(R.id.btn);
        bt.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                String input = et.getText().toString();
                Editor editor = shared.edit();
                editor.putString("PASSWORD", input);
                editor.commit();
            }
        });
    }
}
```

【特别说明】：标准SharedPreferences的键-值都放在一个简单的XML文件中，Secure-Preferences存储机制相同，但是其键-值都被AES对称密钥加密然后进行了Base64编码，密钥在第一次实例化时被存储。

3.2.2 数据库明文存储

许多应用将用户隐私如账户和密码等明文保存在数据库中。

```
ari@arinux:Secr3tMgr_680932f10ed4bb347dec46bdd8a34de487df1d13$ sqlite3 filesystem/data/system/lockset
SQLite version 3.8.7.1 2014-10-29 13:59:56
Enter ".help" for usage hints.
sqlite> .tables
android_metadata locksettings
sqlite> select * from locksettings;
1|lockscreen.disabled|0|0
2|migrated|0|true
3|migrated_user_specific|0|true
4|lockscreen.enabledtrustagents|0|org.cyanogenmod.profiles/.ProfilesTrustAgent,com.google.android.gms,
5|lockscreen.password_salt|0|-6140990771726895285
6|lock pattern autolock|0|0
8|lockscreen.password_type_alternate|0|0
9|lockscreen.password_type|0|327680
10|lockscreen.passwordhistory|0|
sqlite>
```

安全建议:SQLCipher是SQLite的开源扩展,为数据库文件提供透明的256位AES加密,具体使用如下:

下载<https://www.zetetic.net/sqlcipher/open-source/> 后复制其/assets和/libs库到相应目录。

```
% hexdump -C unencrypted-sqlite.db
00000000 53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00 |SQLite format 3.
00000010 04 00 01 01 00 40 20 20 00 00 00 02 00 00 00 03 |.....@.....
00000020 00 00 00 00 00 00 00 00 00 00 00 00 41 01 06 |.....A.....
00000030 17 1b 1b 01 5b 74 61 62 6c 65 73 65 63 72 65 74 |....[tablesecret
00000040 73 73 65 63 72 65 74 73 03 43 52 45 41 54 45 20 |ssecret.CREATE
00000050 54 41 42 4c 45 20 73 65 63 72 65 74 73 28 69 64 |TABLE secrets(id
00000060 2c 20 70 61 73 73 77 6f 72 64 2c 20 6b 65 79 29 |, password, key)
00000070 00 00 00 00 00 00 00 00 00 00 00 00 21 01 04 |.....!.....
00000080 25 1d 1f 4c 61 75 6e 63 68 20 43 6f 64 65 73 70 |%..Launch Codesp
00000090 61 24 24 77 6f 72 64 70 72 6f 6a 65 74 69 6c 65 |a$$wordprojtile
```

SQLite

Data is insecure and easily readable using off-the-shelf programs

```
% hexdump -C encrypted-sqlcipher.db
00000000 de ab bc 3a 40 2b 5d 00 b0 d2 9e 3b 75 91 76 73 |...: @+)...;u.v$
00000010 bc 41 70 0c 8c ab a0 7a 37 eb a2 a8 a9 27 a5 0a |.Ap....z7....'..
00000020 38 c9 0b 9c 06 57 78 96 67 a2 e5 78 f8 8c 58 f3 |8....Wx.g..x..X.
00000030 ea 7c c6 23 14 8a 75 33 d0 a5 2c 30 2e e1 a4 96 |.|.#..u3...0....
00000040 b1 c6 5a 21 67 0a 31 bb 3b de a2 d4 80 b4 60 e3 |..Z!g.1.;.....'
00000050 05 b0 75 04 f2 26 66 ed c7 4e 7e 9c ac 2e ec 1d |..u..&f..N~....
00000060 2d fc 31 b4 32 ce 24 0a d0 23 71 b0 1f 21 12 2c |-.1.2.$..#q...!.,
00000070 92 af 8e d9 de ac 76 e6 20 62 56 c6 f5 05 f5 b3 |.....v. bV.....
00000080 53 d0 5f 4c 5e ec 5b 8a be e7 d1 46 f0 d9 dc b9 |S..L^.[....F....
00000090 a3 59 d6 63 a4 ae cf d8 e4 82 29 83 dd c7 86 13 |.Y.c.....).....
```

SQLCipher

AES-256 encryption secures database contents making it unreadable without the key

首先, 创建一个MyDatabaseHelper继承自SQLiteOpenHelper, 属于net.sqlcipher.database包:

```
import android.content.Context;
import net.sqlcipher.database.SQLiteDatabase;
import net.sqlcipher.database.SQLiteDatabase.CursorFactory;
import net.sqlcipher.database.SQLiteOpenHelper;

public class MyDatabaseHelper extends SQLiteOpenHelper {
    public static final String CREATE_TABLE = "create table Book(name text, pages
integer)";

    public MyDatabaseHelper(Context cont, String name, CursorFactory fat, int ves) {
        super(cont, name, fat, ves);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int arg1, int arg2) {
    }
}
```

```
}
```

然后，打开或新建MainActivity作为程序主Activity，代码如下所示：

```
public class MainActivity extends Activity {
    private SQLiteDatabase db;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        SQLiteDatabase.loadLibs(this);
        MyDatabaseHelper dbHelper = new MyDatabaseHelper(this, "myapp.db", null, 1);
        db = dbHelper.getWritableDatabase("secret_key");
        Button addData = (Button)findViewById(R.id.add_data);
        Button queryData = (Button)findViewById(R.id.query_data);
        addData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                ContentValues values = new ContentValues();
                values.put("name", "Mr.Wang");
                values.put("pages", 566);
                db.insert("Book", null, values);
            }
        });
        queryData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Cursor cursor = db.query("Book", null, null, null, null, null, null);
                if (cursor != null) {
                    while (cursor.moveToNext()) {
                        String name = cursor.getString(cursor.getColumnIndex("name"));
                        int pages = cursor.getInt(cursor.getColumnIndex("pages"));
                    }
                }
                cursor.close();
            }
        });
    }
}
```

3.3 软件数据存储

为了将数据安全地保存在设备上，需要正确使用数据加密技术，实现较高强度的加密算法保护，具体来说是可以使用第三方库如Spongy Castle，或者是使用Android4.3中引入的KeyStore系统服务。

3.3.1 基于密钥加密

使用第三方库加密：首先从<https://github.com/rtyley/spongycastle>下载最新版spongycastle；解压后将.jar文件复制到APP的/libs目录下，然后添加一下代码到APP的对象中。

```
static {
    Security.insertProviderAt(new
org.spongycastle.jce.provider.BouncyCastleProvider(), 1);
}
```

备注:在static代码中添加Security.insertProviderAt()的position=1方法确保了已经被绑定在APP中的/lib文件夹里的Spongy Castle provider会被优先调用,且它不需要对现有代码进行任何修改。

对称加密密钥生成: 对称密钥是一个同时用于加密和解密的密钥,它使用到伪随机数发生器。默认 AES算法在Spongy Castle中可设置AES-GCM算法,同时具备认证功能和签名以检测密文的篡改发生。

```
public class GeneratingKeys {
    public static byte[] strongerEncrypt(String plainText, byte[] iv) throws
GeneralSecurityException, IOException {
        final Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "SC");
        cipher.init(Cipher.ENCRYPT_MODE, getKey(), new IvParameterSpec(iv));
        return cipher.doFinal(plainText.getBytes("UTF-8"));
    }
    public static SecretKey genAESKey(int keysize) throws NoSuchAlgorithmException {
        final SecureRandom random = new SecureRandom();
        final KeyGenerator generator = KeyGenerator.getInstance("AES");
        generator.init(keysize, random);
        return generator.generateKey();
    }
    private static SecretKey key;
    public static SecretKey getKey() throws NoSuchAlgorithmException {
        if (key == null) {
            key = genAESKey(256);
        }
        return key;
    }
    private static IvParameterSpec iv;
    public static IvParameterSpec getIV() {
        if (iv == null) {
            byte[] ivByteArray = new byte[32];
            new SecureRandom().nextBytes(ivByteArray);
            iv = new IvParameterSpec(ivByteArray);
        }
        return iv;
    }
    public static byte[] encrypt(String plainText) throws GeneralSecurityException,
IOException {
        final Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, getKey(), getIV());
        return cipher.doFinal(plainText.getBytes("UTF-8"));
    }
    public static String decrypt(byte[] cipherText) throws GeneralSecurityException,
IOException {
```



```

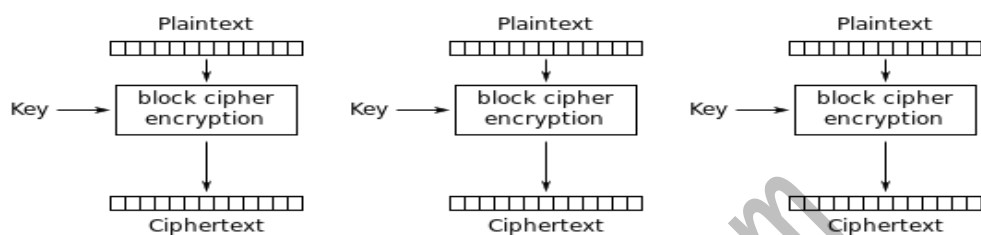
final Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(Cipher.DECRYPT_MODE, getKey());
return cipher.doFinal(cipherText).toString();
}
private GeneratingKeys() {
}
}

```

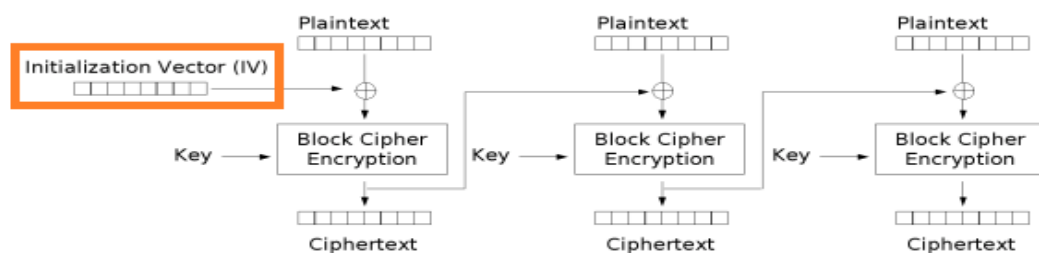
【特别说明】：在开发中经常将密钥存在SharedPerferences中并设置Context.MODE_PRIVATE访问。

Cipher配置风险：默认参数为"AES/ECB/PKCS5Padding"，因ECB模式安全性较低，使用存在风险。

final Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding", "SC"); //推荐使用



Electronic Codebook (ECB) mode encryption



Cipher Block Chaining (CBC) mode encryption

种子设置的风险：Android 4.2之前允许开发者设置种子，SecureRandom随机性是通过它的seed来保证的，如果输入相同seed会导致生成重复的随机数。SecureRandom内部维护一个internal random state，它生成随机数的方式具有确定性。如果输入相同的seed那么生成的随机数也相同，错误方式如下。

```

byte[] myCustomSeed = new byte[] { (byte)42 };
secureRandom.setSeed(myCustomSeed);
int notRandom = secureRandom.nextInt();

```

安全建议：不人工设置种子，使用系统默认的即可。

3.3.2 基于口令加密

对于不安全的APP沙箱来说，基于口令的加密-PBE是理想的解决方案：在运行时根据用户口令加密。

```

public class GeneratingPBEKeys {
    private static IvParameterSpec iv;
    public static IvParameterSpec getIV() {
        if (iv == null) {
            iv = new IvParameterSpec(generateRandomByteArray(32));
        }
        return iv;
    }
}

```

```

private static byte[] salt;
public static byte[] getSalt() {
    if (salt == null) {
        salt = generateRandomByteArray(32);
    }
    return salt;
}
public static byte[] generateRandomByteArray(int sizeInBytes) {
    byte[] randomNumberByteArray = new byte[sizeInBytes];
    new SecureRandom().nextBytes(randomNumberByteArray);
    return randomNumberByteArray;
}
public static SecretKey generatePBEKey(char[] password, byte[] salt) throws
NoSuchAlgorithmException, InvalidKeySpecException {
    SecretKeyFactory sKFat = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    KeySpec keySpec = new PBEKeySpec(password, salt, 10000, 256);
    return sKFat.generateSecret(keySpec);
}
public static byte[] encryptWithPBE(String plainText, String userPassword) throws
GeneralSecurityException, IOException {
    SecretKey secretKey = generatePBEKey(userPassword.toCharArray(), getSalt());
    final Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, secretKey, getIV());
    return cipher.doFinal(plainText.getBytes("UTF-8"));
}
public static String decryptWithPBE(byte[] cipherText, String userPassword) throws
GeneralSecurityException, IOException {
    SecretKey secretKey = generatePBEKey(userPassword.toCharArray(), getSalt());
    final Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, secretKey, getIV());
    return cipher.doFinal(cipherText).toString();
}
private GeneratingPBEKeys() {
}
}

```

3.3.3 KeyStore 保护

Android KeyStore 是证书存储区，可将加密私钥保存在其中，但不支持对称密钥存储。示例如下：

```

import java.security.KeyPairGenerator;
import java.security.KeyStore;
import java.security.PrivateKey;
import javax.security.auth.x500.X500Principal;
import android.security.KeyPairGeneratorSpec;
@TargetApi(Build.VERSION_CODES.JELLY_BEAN_MR2)
public class AndroidKeystoreUtil {

```

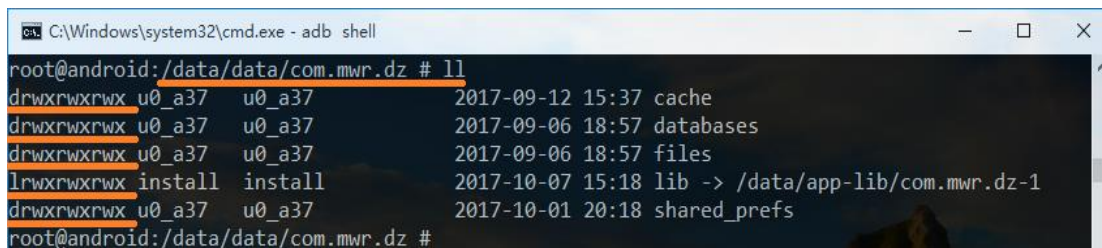
```
KeyStore keyStore;
private static final String KEY_ALIAS = "MY_KEY";
private static final String TAG = "AndroidKeystoreUtil";
public static final String ANDROID_KEYSTORE = "AndroidKeyStore";
public void loadKeyStore() {
    try {
        keyStore = KeyStore.getInstance(ANDROID_KEYSTORE);
        keyStore.load(null);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void generateNewKeyPair(String alias, Context context) throws Exception {
    Calendar start = Calendar.getInstance();
    Calendar end = Calendar.getInstance();
    end.add(1, Calendar.YEAR);
    KeyPairGeneratorSpec spec = new KeyPairGeneratorSpec.Builder(context)
        .setAlias(alias).setSubject(new X500Principal("CN=" + alias))
        .setSerialNumber(BigInteger.TEN).setStartDate(start.getTime())
        .setEndDate(end.getTime()).build();
    KeyPairGenerator gen = KeyPairGenerator.getInstance("RSA", ANDROID_KEYSTORE);
    gen.initialize(spec);
    gen.generateKeyPair();
}

public PrivateKey loadPrivateKey(String alias) throws Exception {
    if (keyStore.isKeyEntry(alias)) {
        return null;
    }
    KeyStore.Entry entry = keyStore.getEntry(KEY_ALIAS, null);
    if (!(entry instanceof KeyStore.PrivateKeyEntry)) {
        return null;
    }
    return ((KeyStore.PrivateKeyEntry) entry).getPrivateKey();
}
```

3.4 内部外部存储

文件权限：当/data/data/app目录下文件全局可读写时，可能出现文件被恶意操作的情况。



```
C:\Windows\system32\cmd.exe - adb shell
root@android:/data/data/com.mwr.dz # ll
drwxrwxrwx u0_a37 u0_a37 2017-09-12 15:37 cache
drwxrwxrwx u0_a37 u0_a37 2017-09-06 18:57 databases
drwxrwxrwx u0_a37 u0_a37 2017-09-06 18:57 files
lrwxrwxrwx install install 2017-10-07 15:18 lib -> /data/app-lib/com.mwr.dz-1
drwxrwxrwx u0_a37 u0_a37 2017-10-01 20:18 shared_prefs
root@android:/data/data/com.mwr.dz #
```

安全建议：一般文件权限不应该设置为777。

3.4.1 内部存储安全

Android也使用本地文件来保存应用的数据，应用文件保存数据的示例代码如下：

```
public void save() {
    String data = et.getText().toString();
    data = "Card Number: " + data + "\n";
    FileOutputStream fos;
    try {
        fos = openFileOutput("secret.txt", Context.MODE_PRIVATE);
        fos.write(data.getBytes());
        fos.close();
        Toast.makeText(getApplicationContext(), "Data Save", Toast.LENGTH_LONG).show();
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

安全建议：在内部存储敏感信息的时候，对数据进行加密；文件设置Context.MODE_PRIVATE的权限。

3.4.2 外部存储安全

SDCARD是Android中重要的存储应用数据的地方，需要注意的是：SDCARD上的数据可被任意访问，把SD卡从设备上拆下来，放到其他设备也可以完全读取其中的数据。应用保存数据的示例代码如下：

```
public void save() {
    String data = et.getText().toString();
    data = "Card Number: " + data + "\n";
    FileOutputStream fos;
    try {
        File myFile = new File("/sdcard/" + "secret.txt");
        myFile.createNewFile();
        fos = new FileOutputStream(myFile);
        fos.write(data.getBytes());
        fos.close();
        Toast.makeText(getApplicationContext(), "Data Save", Toast.LENGTH_LONG).show();
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

以上代码需要"WRITE_EXTERNAL_STORAGE"权限，所以需要在AndroidManifest.xml加入权限声明：

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

安全建议：在外部存储敏感信息的时候，需要对数据进行加密保护，防止泄露。

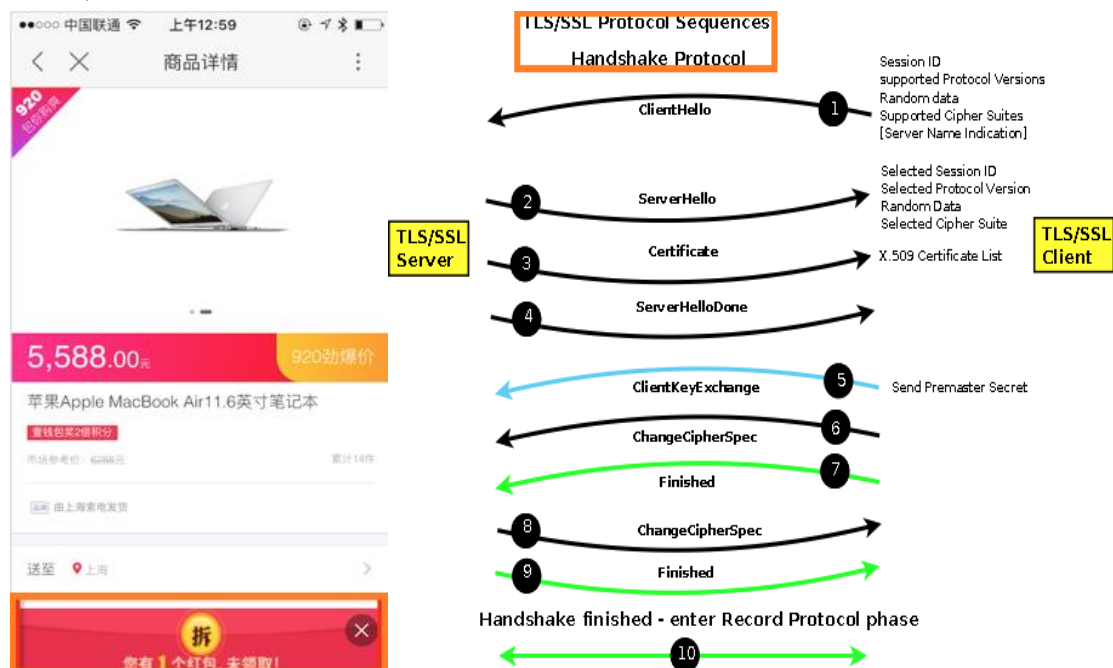
4. 通信安全分析

4.1 常见安全问题

通信安全是APP在传输过程中十分重要的安全问题，我们需要对敏感数据进行加密保护-基于SSL/TLS的HTTPS传输保护，使用证书加密技术验证服务器的合法性和通道的安全性，中间的验证环节是问题关键。

4.2 使用 HTTPS 传输

HTTP协议是没有加密的明文传输协议，如果APP采用HTTP传输数据，则会泄露传输内容，可能被中间人劫持，修改传输的内容。如下图所示就是典型的APP HTTP通信被运营商劫持修改插入了广告：



为了保护用户的信息被嗅探、流量劫持保障信息安全、保护自己的商业利益，减少攻击面，需要保障通信信道的安全，采用HTTPS是比较好的方式。但是如果HTTPS使用不当，就很难起到应有的保护效果。HTTPS是HTTP over SSL/TLS，HTTP是应用层协议，TCP是传输层协议，在应用层和传输层之间，增加了一个安全套接层SSL/TLS，SSL/TLS层负责客户端和服务端之间的加解密算法协商、密钥交换、连接建立。

4.3 限定受信 SSL

缺少相应的服务证书和主机信任安全校验很容易导致中间人攻击，目前可以通过两种途径实现校验：限定可信证书和可信公钥。可信证书是把全部证书和根限定在本地truststore中记录的证书集中，使得整个证书链获得全面验证。而可信公钥是从SSL证书中提取公钥进行校验，好处是SSL证书升级不需要升级APP。

4.3.1 自定义 X509TrustManager

在使用HttpsURLConnection发起 HTTPS 请求的时候，提供了一个自定义的X509TrustManager，未实现安全校验逻辑，如果不提供自定义的X509TrustManager，忘记实现相应的方法，代码运行可能会异常。

```
TrustManager xtm = new X509TrustManager() {  
    public void checkClientTrusted(X509Certificate[] chain, String authType) throws  
        CertificateException {  
        //do nothing, 接受任意客户端证书  
    }  
    public void checkServerTrusted(X509Certificate[] chain, String authType) throws
```

```
CertificateException {  
    //do nothing, 接受任意服务端证书  
}  
public X509Certificate[] getAcceptedIssuers() {  
    return null;  
}  
};  
sslContext.init(null, new TrustManager[] { xtm }, null);
```

安全建议：不论是权威机构颁发的证书还是自签名的，打包一份到app内部如asset里。通过这份内置的证书初始化一个KeyStore，然后用这个KeyStore去引导生成的TrustManager来提供验证，代码如下：

```
try {  
    CertificateFactory cf = CertificateFactory.getInstance("X.509");  
    InputStream caInput = new BufferedInputStream(getAssets().open("myca.crt"));  
    Certificate ca;  
    try {  
        ca = cf.generateCertificate(caInput);  
        Log.i("Longer", "ca=" + ((X509Certificate) ca).getSubjectDN());  
        Log.i("Longer", "key=" + ((X509Certificate) ca).getPublicKey());  
    } finally {  
        caInput.close();  
    }  
    String keyStoreType = KeyStore.getDefaultType();  
    KeyStore keyStore = KeyStore.getInstance(keyStoreType);  
    keyStore.load(null, null);  
    keyStore.setCertificateEntry("ca", ca);  
    String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();  
    TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);  
    tmf.init(keyStore);  
    SSLContext context = SSLContext.getInstance("TLSv1", "AndroidOpenSSL");  
    context.init(null, tmf.getTrustManagers(), null);  
    URL url = new URL("https://www.mysite.com/");  
    HTTPSURLConnection urlConnection = (HTTPSURLConnection)url.openConnection();  
    urlConnection.setSSLSocketFactory(context.getSocketFactory());  
    InputStream in = urlConnection.getInputStream();  
    copyInputStreamToOutputStream(in, System.out);  
} catch (CertificateException e) {  
    e.printStackTrace();  
} catch (IOException e) {  
    e.printStackTrace();  
} catch (NoSuchAlgorithmException e) {  
    e.printStackTrace();  
} catch (KeyStoreException e) {  
    e.printStackTrace();  
} catch (KeyManagementException e) {  
    e.printStackTrace();  
}
```



```
} catch (NoSuchProviderException e) {  
    e.printStackTrace();  
}
```

【特别说明】：用上述代码访问其他网站会抛出SSLHandshakeException异常，它是对于特定证书生成的TrustManager，只能验证与特定服务器的安全链接，提高了安全性，对于非浏览器APP来说可以接受。

安全建议：打包一份到证书到APP内部，直接自定义一个TrustManager实现校验逻辑：服务器证书是否过期，证书签名是否合法。具体的代码如下：

```
try {  
    CertificateFactory cf = CertificateFactory.getInstance("X.509");  
    InputStream caInput = new BufferedInputStream(getAssets().open("myca.crt"));  
    final Certificate ca;  
    try {  
        ca = cf.generateCertificate(caInput);  
    } finally {  
        caInput.close();  
    }  
    SSLContext context = SSLContext.getInstance("TLSv1", "AndroidOpenSSL");  
    context.init(null, new TrustManager[] {  
        new X509TrustManager() {  
            @Override  
            public void checkClientTrusted(X509Certificate[] chain, String authType)  
throws CertificateException {  
            }  
            @Override  
            public void checkServerTrusted(X509Certificate[] chain, String authType)  
throws CertificateException {  
                for (X509Certificate cert : chain) {  
                    cert.checkValidity();  
                    try {  
                        cert.verify(((X509Certificate) ca).getPublicKey());  
                    } catch (NoSuchAlgorithmException e) {  
                        e.printStackTrace();  
                    } catch (InvalidKeyException e) {  
                        e.printStackTrace();  
                    } catch (NoSuchProviderException e) {  
                        e.printStackTrace();  
                    } catch (SignatureException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        }  
    }, null);  
    @Override  
    public X509Certificate[] getAcceptedIssuers() {  
        return new X509Certificate[0];  
    }  
}
```

```
    }  
    }, null);  
    URL url = new URL("https://www.mysite.com/");  
    HTTPSURLConnection urlConnection = (HTTPSURLConnection)url.openConnection();  
    urlConnection.setSSLSocketFactory(context.getSocketFactory());  
    InputStream in = urlConnection.getInputStream();  
    copyInputStreamToOutputStream(in, System.out);  
} catch (CertificateException e) {  
    e.printStackTrace();  
} catch (IOException e) {  
    e.printStackTrace();  
} catch (NoSuchAlgorithmException e) {  
    e.printStackTrace();  
} catch (KeyManagementException e) {  
    e.printStackTrace();  
} catch (NoSuchProviderException e) {  
    e.printStackTrace();  
}  
}
```

【特别说明】：代码只能访问 <https://www.mysite.com/> 相关的地址，如果访问其他网站，则会在 `cert.verify(((X509Certificate) ca).getPublicKey())` 处抛异常，导致连接失败。

4.3.2 自定义 HostnameVerifier

在握手期间如果URL的主机名和服务器的标识主机名不匹配，则验证机制可以回调此接口的实现程序来确定是否应该允许此连接。如果回调内实现不恰当，默认接受所有域名，则有安全风险。

```
HostnameVerifier htv = new HostnameVerifier() {  
    @Override  
    public boolean verify(String hostname, SSLSession session) {  
        return true; //接受任意域名服务器  
    }  
};  
HTTPSURLConnection.setDefaultHostnameVerifier(htv);
```

安全建议：简单的话就是根据域名进行字符串匹配校验；复杂的业务可以结合配置中心、白名单、黑名单、正则匹配等多级别动态校验；总体来说，逻辑就是要正确地实现verify方法。

```
HostnameVerifier htv = new HostnameVerifier() {  
    @Override  
    public boolean verify(String hostname, SSLSession session) {  
        if("yourhostname".equals(hostname)){  
            return true;  
        } else {  
            HostnameVerifier hnv = HTTPSURLConnection.getDefaultHostnameVerifier();  
            return hnv.verify(hostname, session);  
        }  
    }  
};  
HTTPSURLConnection.setDefaultHostnameVerifier(htv);
```

4.3.3 信任全部主机

```
SSLSocketFactory sf = new MySSLSocketFactory(trustStore);  
sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
```

安全建议：主机名验证策略改成严格模式。

```
SSLSocketFactory sf = new MySSLSocketFactory(trustStore);  
sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
```

4.3.4 WebView 的 HTTPS 安全

webview加载H5页面，如果采用的是可信CA证书，在webView.setWebViewClient(webviewClient)时重载WebViewClient的onReceivedSslError()，如果出现证书错误，直接调用handler.proceed()会忽略错误继续加载证书有问题的页面，如果调用handler.cancel()可以终止加载证书有问题的页面，证书出现问题了，可提示用户风险让用户选择加载与否，如果是需要安全级别比较高，可以终止页面加载。

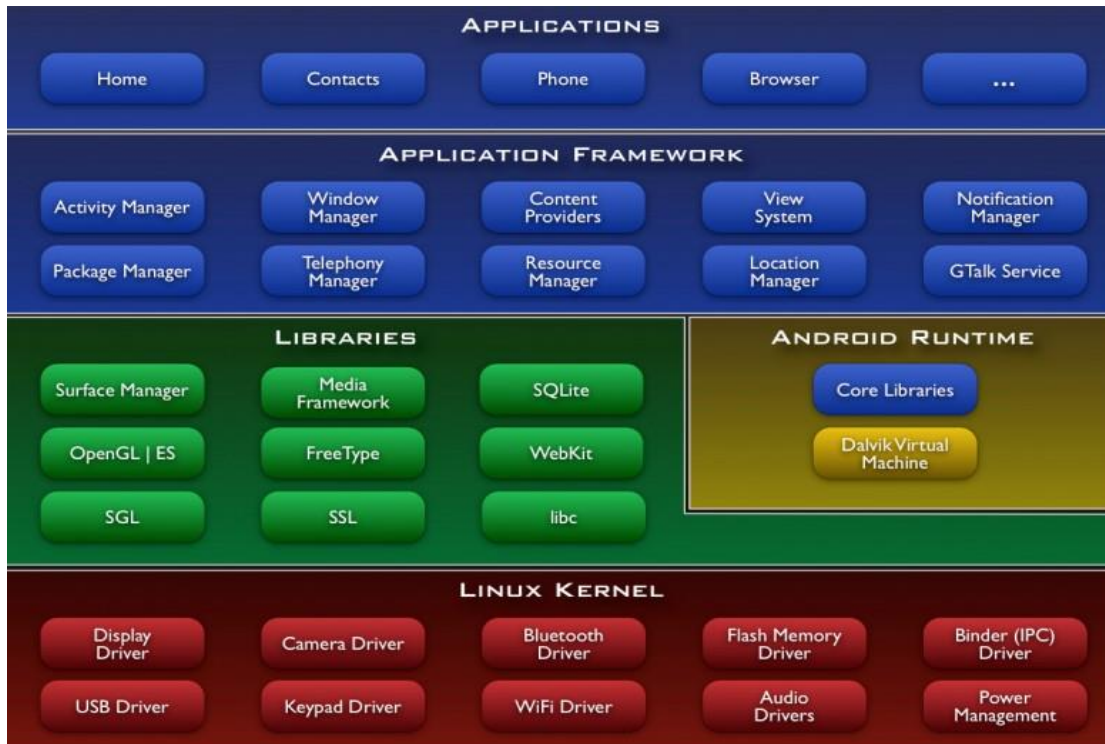
```
@Override  
public void onReceivedSslError(WebView view, final SslErrorHandler handler, SslError error) {  
    final AlertDialog.Builder builder = new AlertDialog.Builder(MyWebviewActivity.this);  
    Log.d("error toString()", error.toString());  
    Log.d("error getPrimaryError()", "" + error.getPrimaryError());  
    SslCertificate sslCertificate = error.getCertificate();  
    Log.d("sslCertificate", sslCertificate.toString());  
  
    switch (error.getPrimaryError()) {  
        case SslError.SSL_DATE_INVALID:  
            Log.d("test", SslError.SSL_DATE_INVALID + " ssl date invalid");  
            break;  
        case SslError.SSL_IDMISMATCH:  
            Log.d("test", SslError.SSL_IDMISMATCH + " hostname mismatch");  
            break;  
        case SslError.SSL_EXPIRED:  
            Log.d("test", SslError.SSL_EXPIRED + " cert has expired");  
            break;  
        case SslError.SSL_UNTRUSTED:  
            Log.d("test", SslError.SSL_UNTRUSTED + "cert is not trusted");  
            break;  
        case SslError.SSL_INVALID:  
            Log.d("test", SslError.SSL_INVALID + "cert is invalid");  
            break;  
        case SslError.SSL_NOTYETVALID:  
            Log.d("test", SslError.SSL_NOTYETVALID + "cert is not yet valid");  
            break;  
    }  
  
    builder.setTitle("SSL证书错误");  
    builder.setMessage("SSL错误码: " + error.getPrimaryError());  
    builder.setPositiveButton("继续", new DialogInterface.OnClickListener() {  
        @Override  
        public void onClick(DialogInterface dialogInterface, int i) {  
            handler.proceed();  
        }  
    });  
    builder.setNegativeButton("取消", new DialogInterface.OnClickListener() {  
        @Override  
        public void onClick(DialogInterface dialogInterface, int i) {  
            handler.cancel();  
        }  
    });  
    final AlertDialog dialog = builder.create();  
    dialog.show();  
}
```

【特别说明】：禁用handler.proceed()，如果webview加载https需要强校验服务端证书，可以在onPageStarted()中用HttpsURLConnection强校验证书的方式来校验服务端证书，校验不通过停止加载。

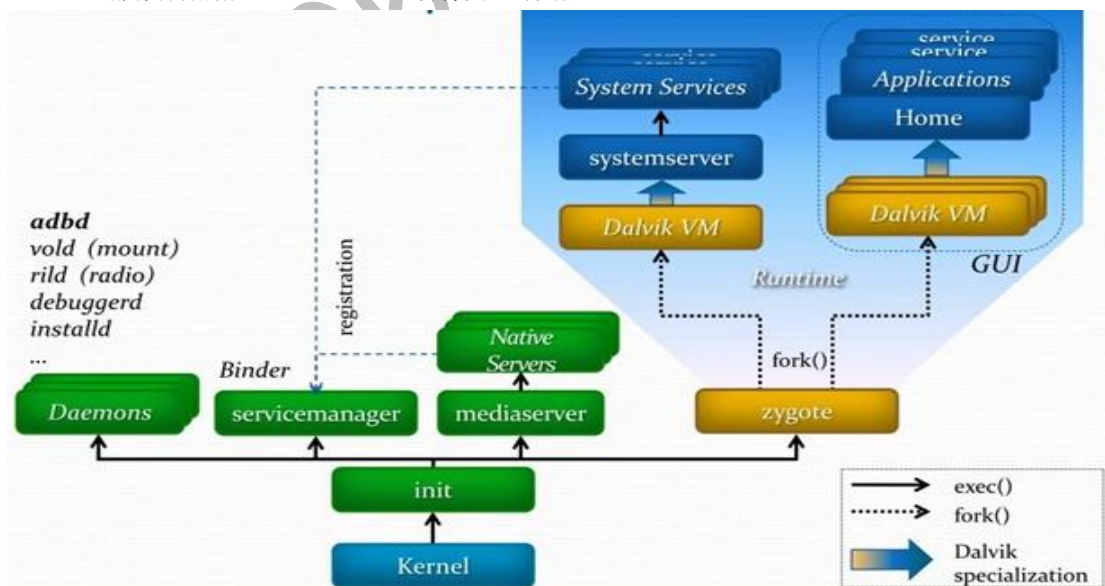
5. 理解 Android 系统

5.1 架构和服务

Android整体架构主要分成5层: Linux内核层(Kernel)、核心库(Libraries)、运行环境层(Android Runtime)、应用框架层(Application Framework)、应用层(Application)。



Android系统的分区结构: Boot Loader (初始化、挂载硬件和环境)、Boot (存储Android的核心)、Radio (基带通信驱动)、Recovery (Mini型Android的镜像)、System (Android的框架等镜像)、User Data (内部存储数据分区)、Cache (各种实用的文件)。



Android系统的启动过程: Boot Loader加载(定制过程初始化代码、启动RAM、Recover Image等)、Kernel加载(加载Linux Kernel和initrd到RAM)、init加载(启动Android系统核心服务)、虚拟机的初始化阶段(启动Zygote进程来创建Dalvik VM, 启动Java组件系统服务和Android Framework)。当系统完全启动之后载入桌面应用程序(Home), 做初始化应用层工作并播ACTION_BOOT_COMPLETED。

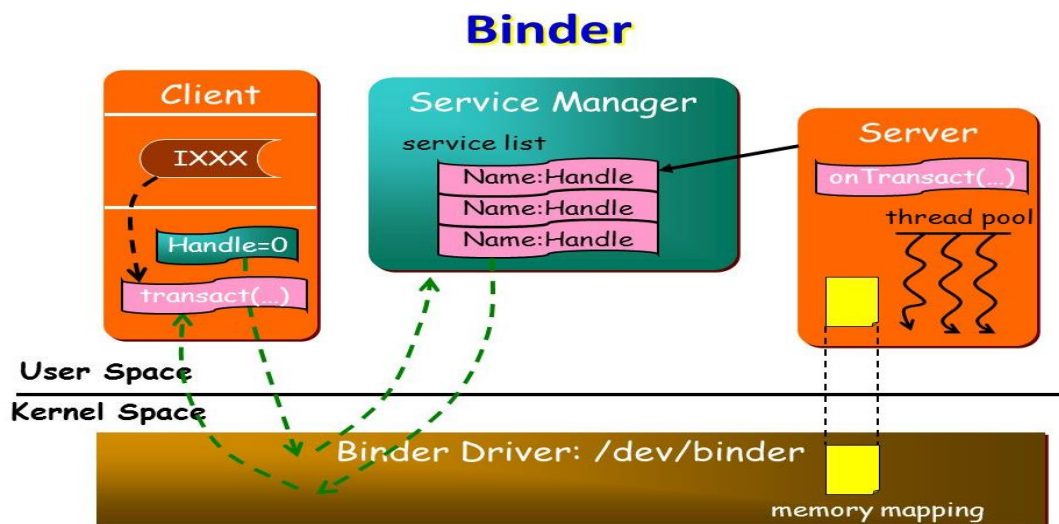
Init进程是一个内核启动的用户级进程: 完成init.rc解析和初始化、属性服务(getprop)初始化、

进入无限for循环以建立子进程并对关键服务的异常进行重启和异常处理。ADB进程分：ADB Client（5037端口）、ADB Server（5037端口）、ADB Daemon（模拟器守护进程）。VOLD（存储类守护）进程：负责CDROM、USB大容量存储等扩展存储挂载自动完成，支持外设的热插拔。Service Manager是用来管理系统中的Service如Input Method、Activity Manager Service，其中有两个比较重要的方法：add_service（系统服务的注册）、check_service（系统服务的检查）。

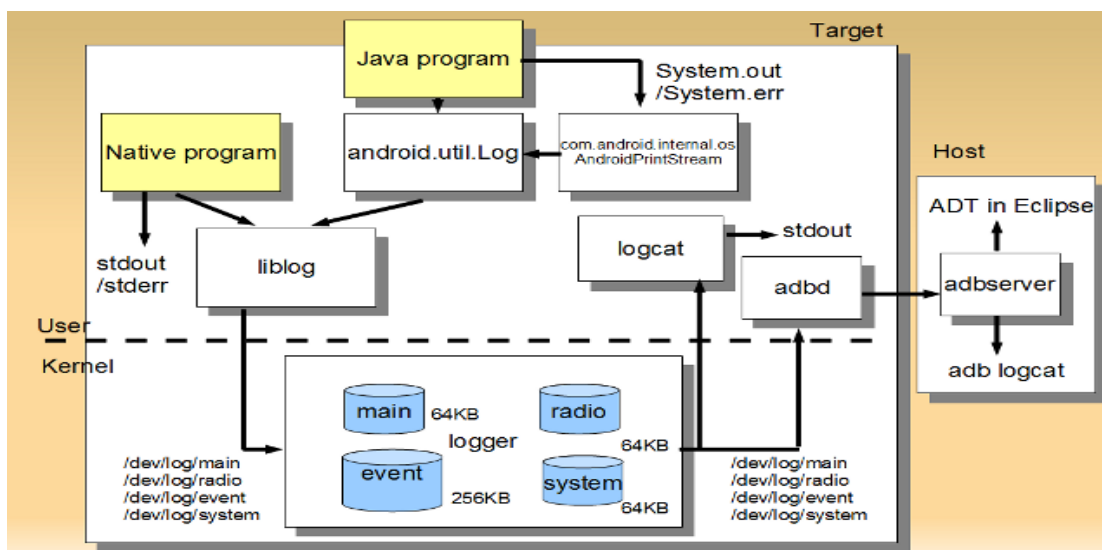
```
C:\Users\Administrator\Desktop>adb shell getprop
[ARGH]: [ARGH]
[dalvik.vm.heapsize]: [32m]
[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
[dev.bootcomplete]: [1]
[gsm.current.phone-type]: [1]
[gsm.defaultpdpcontext.active]: [true]
[gsm.network.type]: [Unknown]
```

key:value
键值对的格式

Binder基于C/S，分为：Client、Service、Service Manager（DNS）、Binder驱动（Route）。Android虚拟机启动前先启动Service Manager进程，它打开Binder驱动并通知该程序，然后Service Manager进入一个循环监听来自客户端的请求，而每个服务启动时都会在服务Manager中进行注册。为了完成进程间通信，Binder采用了AIDL（Android Interface Definition Language）来描述进程间的接口，它遵循Linux设备驱动模型，以特殊的字符型设备节点形式 /dev/binder存在。

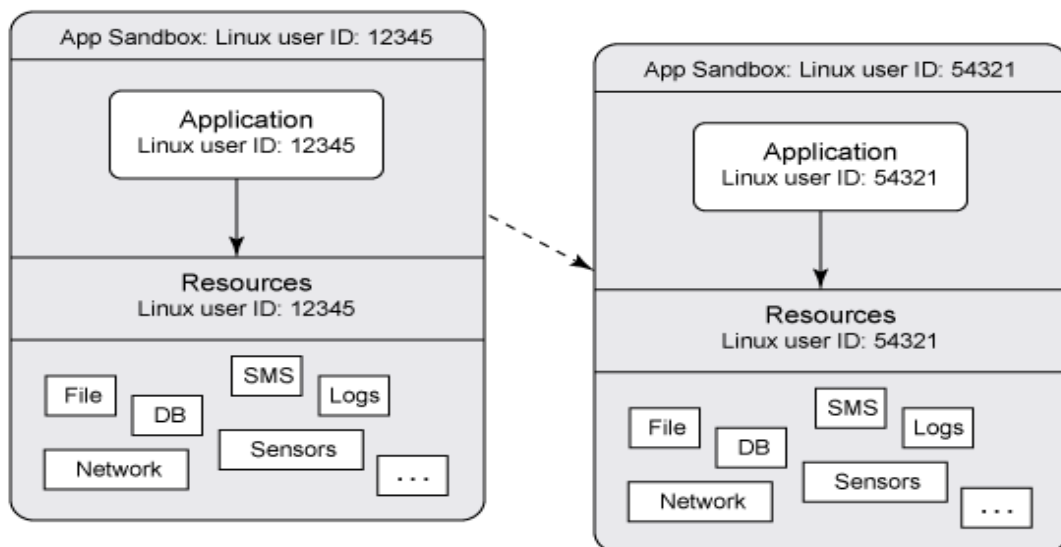


Android使用了另外一套日志系统Logger，它这次Logcat命令用于查看日志缓冲区。它提供了4个独立的日志缓冲区：主缓冲区、广播缓冲区、事件缓冲区以及系统缓冲区。通过调用的android.util.log方法类可以写到主缓冲区，根据日志条目优先级Log.i-信息、Log.d-调试、Log.e-错误。



5.2 安全的边界

Android系统继承了Linux的执行访问文件系统方式，使用UID/GID隔离机制，需要特别说明的是Android中还使用sharedUserId共享一个UID来让两个应用共存在一个进程中实现资源的共享。

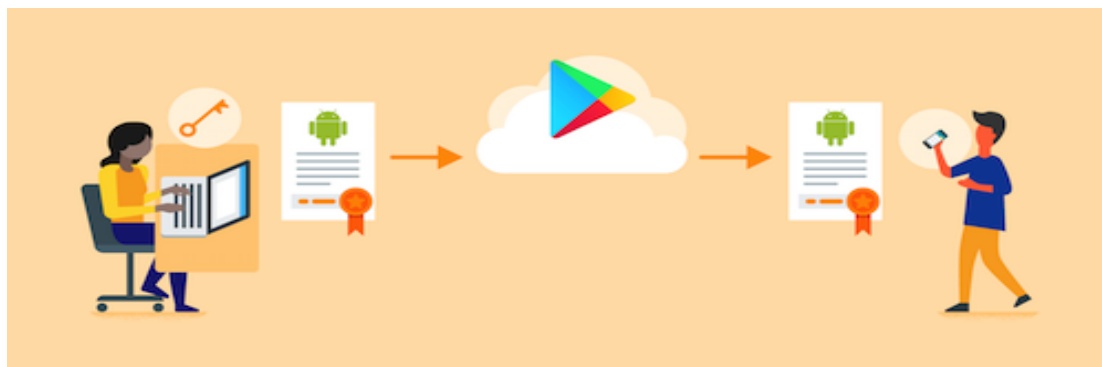


Two applications on different processes (with different user-ids)

Android的权限授予机制分成：文件权限、IPC权限、API权限，其中文件权限和IPC权限和Linux一致，而API权限则是Android特有的，它包括调用Android API、Framework以及第三方框架所使用的权限等。被授予权限后可以调用各种服务接口，低级别权限在用户安装应用时就会直接授予如READ_PHONE_STATE，高级别权限如SEND_SMS在使用时系统会出现弹框，需要用户知晓并授权才能够使用。

```
<permission android:description="string resource"
    android:icon="drawable resource"
    android:label="string resource"
    android:name="string"
    android:permissionGroup="string"
    android:protectionLevel=["normal" | "dangerous" |
        "signature" | "signatureOrSystem"] />
```

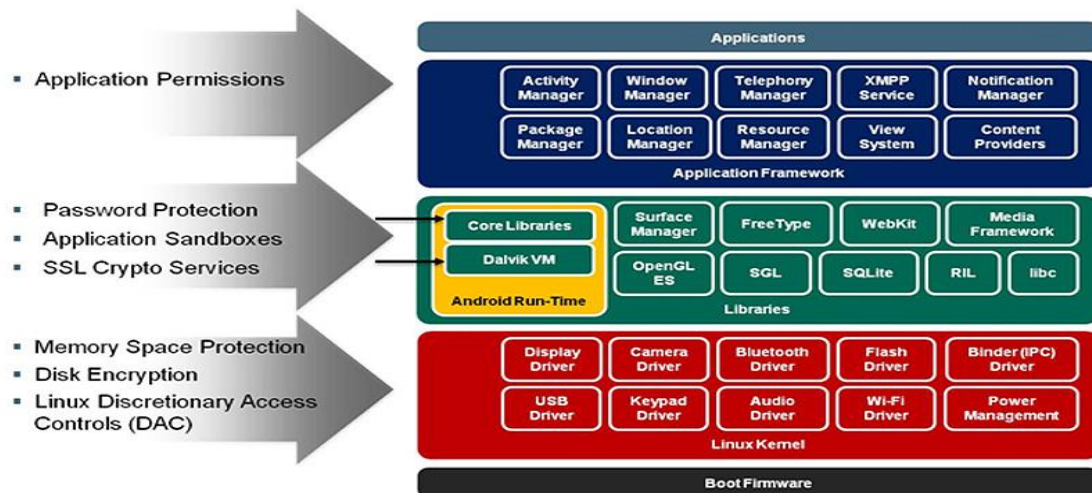
Android利用数字签名机制来标识应用程序与其作者之间的信任关系，它可以用语识别应用二次打包。只有相同签名、包名才被认为是同一程序并允许覆盖安装，基于数字证书的权限授予机制允许和其它程序共享功能或数据给具有相同数字证书程序，permission为android:protectionLevel="signature"。执行签名命令：java -jar signapk.jar publickey privatekey input.apk output.apk。



安装应用程序安装包APK时，通过CERT.RSA查找公钥和算法，并对CERT.SF进行解密和签名验证，确认MANIFEST.MF最终对每个文件签名校验，具体可用程序自检测、可信赖的第三方检测、限定证书安装。

5.3 安全的架构

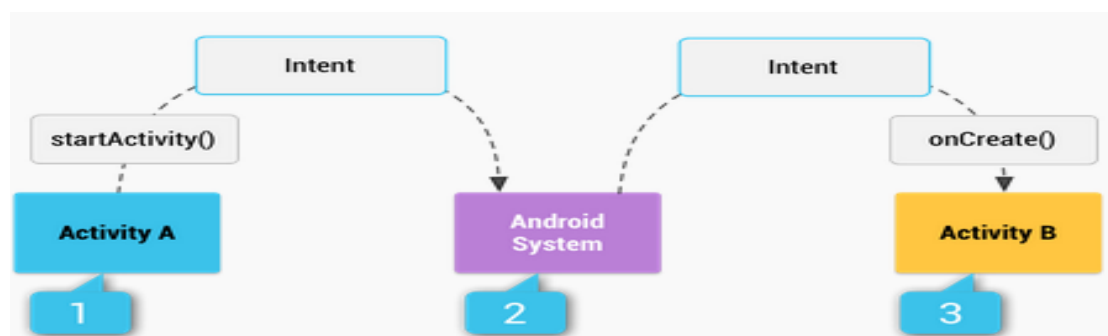
Android系统在每一层都采取了不同的安全措施：内核层（Linux文件权限）、运行时环境（Dalvik沙箱机制）、框架层（数字证书）、应用层（配置权限、代码保护）。Android使用特殊的平台密钥来签名预安装的应用程序软件包，这些特殊签名的应用有系统用户特权，且无法再未获得系统授权时进行更新。



Android应用程序主要模块包括：Activity、Service、Broadcast Receiver、Content Providers、Intent以及Manifest.xml。其中Intent是一种运行时绑定机制，通过它可以向Android表达某种请求。

Android Components

Application components are the essential building blocks of an Android application.



Activity是应用程序的交互组件，都是运行在系统的主线程上，不能用来处理耗时的操作。Service是运行在后台的功能模块，它运行在系统主线程中，当需要进行一项耗时操作时需要另起一个线程来处理。Content Provider向其他应用提供统一的数据访问方式：共享文件、Shared Preferences、SQLite。Broadcast Receiver用来接收广播通知消息并做出对应处理的组件，很多广播都是源自于系统代码的。广播的注册有两种方式：静态注册（AndroidManifest.xml）、动态注册（Java代码中进行声明）。