第 3 章 移动应用安全基线

CHAPTER 3

3.1 移动应用评估思路

3.1.1 移动应用可能面临的威胁

在风起云涌的移动互联网时代，随着智能手机移动终端设备的普及，人们逐渐习惯了使用应用客户端上网的方式，而智能终端的普及不仅推动了移动互联网的发展，也带来了移动应用的爆炸式增长。在海量的移动应用中，应用可能会面临如下威胁。

1. 手机木马

比较常见的手机木马传播方式是通过诈骗短信，利用受害者贪小便宜的心理，在短信内链接网址植入木马病毒。受害者主动或被动安装软件后，木马会在后台运行，随时随地监控手机信息，并通过网络上传到攻击者的服务器上。

有的木马还会盗取受害者在手机操作过的银行卡信息、盗取卡号、截取验证码，进而骗取受害人银行卡中的钱。

2. 手机病毒

手机病毒的传播特点与计算机病毒类似，也会伪装成普通的应用程序。受害用户通过恶意链接地址下载安装病毒程序后，病毒程序将对受害者的手机进行破坏。比较有名的是各类勒索软件，伪装成游戏外挂或者工具插件等程序，安装后通过修改锁屏密码并强制锁屏等手段阻止用户对手机的正常操作，从而勒索受害者。

3. 应用篡改

就像第 2 章破解 Android 应用时对反编译的应用进行修改一样，如果某款应用不对代码进行保护或没有对其完整性进行验证，那么攻击者可通过对应用反编译后增加恶意代码，实现对应用逻辑的篡改，实现恶意攻击行为。

4. 应用破解

破解行为通常针对应用的付费功能，即针对应用中需要付费才能使用的功能进行破解。如果一款应用被破解，会对应用的发行商利益造成损害。最直接的应用破解方式就是篡改应用，修改付费验证的逻辑。

5. 恶意软件钓鱼

钓鱼软件类似于手机木马，都是通过诈骗短信或者伪装成其他应用的方式，欺骗受害者将应用安装到手机中，目的同样是获取手机用户的数据。

6. 软件反编译后二次打包

二次打包通常出现在应用的篡改和破解中。只要不修改 Manifest 文件中的包名和应用名的信息,二次打包的程序与原来的程序名称就是一模一样的,有些甚至能像原版应用一样触发应用包更新。因此,在不正规的渠道网站上下载的应用不能保证都是官方上线的正版应用,有些就可能是经过二次打包的程序。

7. 通过应用进行账号窃取

通过应用窃取账号不仅可以使用经过篡改的二次打包程序,对于某些使用明文保存用户数据或者明文传输请求的应用,通过 ZAP 等抓包工具可以实现不需要修改应用而盗取用户账号。

8. 修改应用进行广告植入

某些应用经过二次打包后植入了自己的广告,这类程序会对用户的体验造成极大影响。并且这类广告极有可能是用来传播木马或钓鱼软件的方式,用户下载安装这类应用就相当于开了后门。

9. 通过应用劫持用户信息

应用劫持也是一类用于窃取用户数据的方式。Activity 界面劫持就是当手机中恶意应用检测到当前运行的目标应用时,就会启动自身的页面覆盖在目标应用之上,这种钓鱼页面设计得与原程序极为相似,从而引诱用户在钓鱼页面输入用户密码,进而窃取用户信息。

3.1.2 移动应用的评估方向

针对上面提到的应用可能会面临的威胁,下面提出 7 个对移动应用进行安全评估的方向。

(1) 敏感信息安全。
(2) 能力调用。
(3) 资源访问。
(4) 反逆向。
(5) 通信安全。
(6) 键盘输入。
(7) 认证鉴权。

3.1.3 移动应用自动化检测思路

根据前面介绍的移动应用安全评估方向,来整理一下使用自动化应用安全检测工具来对 Android 应用进行安全评估的思路。

首先是代码审计阶段,包括检查 AndroidManifest.xml 配置文件中的配置选项,例如,权限配置、组件配置等; Android 应用源代码中较明显的漏洞,例如,敏感信息的硬编码、使用了不安全的方法函数等。

然后是配置验证阶段,对常见的安全问题进行配置验证,通过模拟攻击的方式,验证终端应用存在的问题。通常会使用脚本框架,例如,Xposed、Frida 脚本触发漏洞,用于模拟针对漏洞的攻击。

最后是人工验证,对于一些无法通过脚本触发的漏洞,可以采用人工验证的方式。例

如，应用篡改，可以通过人工反编译应用，在解包得到的代码中添加自定义逻辑并通过二次打包的方式进行验证。

3.2　安全检测的要点

安全检测主要针对 Android 应用开发过程中的常见漏洞以及危险操作进行。

3.2.1　Android 常见漏洞分析


视频 4

本节介绍一些常见的 Android 风险漏洞以及修复建议。

1. 程序可被任意调试

风险描述：AndroidManifest.xml 中的 android:debuggable 属性值为 true。

危害描述：应用可以被任意调试。

修复建议：将 android:Debugable 属性值设置为 false。

2. 程序数据任意备份

风险描述：AndroidManifest.xml 中的 android:allowBackup 属性被设置为 true。

危害描述：应用数据可以被备份导出。

修复建议：将 android:allowBackup 属性值设置为 false。

3. Activity 组件暴露

风险描述：Activity 组件的属性 exported 被设置为 true 或是未设置 exported 值但 IntentFilter 不为空时，Activity 被认为是导出的，可通过设置相应的 Intent 唤起 Activity。

危害描述：导出的 Activity 可能会遭到黑客的越权攻击。

修复建议：如果组件不需要与其他应用共享数据或交互，那么应将 AndroidManifest.xml 配置文件中设置该组件为 exported="false"。如果组件需要与其他应用共享数据或交互，那么应对组件进行权限控制和参数校验。

4. Service 组件暴露

风险描述：Service 组件的属性 exported 被设置为 true 或是未设置 exported 值但 IntentFilter 不为空时，Service 被认为是导出的，可通过设置相应的 Intent 唤起 Service。

危害描述：未经保护的导出 Service 可能会遭到黑客通过构造恶意数据实施的越权攻击。

修复建议：在组件不需要与外部应用进行数据共享与交互的情况下，在 AndroidManifest.xml 中将该组件的 exported 属性设置为 false。如果需要暴露组件，那么应对该组件添加权限控制。

5. ContentProvider 组件暴露

风险描述：ContentProvider 组件的属性 exported 被设置为 true 或是 Android API <= 16 时，ContentProvider 被认为是导出的。

危害描述：黑客可能访问到应用本身不想共享的数据或文件。

修复建议：对需要暴露的组件添加权限控制，对于不需要暴露的组件在 AndroidManifest.xml 中将该组件设置为 exported="false"。

6. BroadcastReceiver 组件暴露

风险描述：BroadcastReceiver 组件的属性 exported 被设置为 true 或是未设置 exported 值但 IntentFilter 不为空时，BroadcastReceiver 被认为是导出的。

危害描述：导出的广播可以导致数据泄露或者是越权。

修复建议：如果组件不需要暴露，需要在 AndroidManifest. xml 文件中明确将 exported 属性设置为 false，如果需要与其他应用进行数据交互，则谨慎使用 IntentFilter，并添加相应的参数校验。

7. WebView 存在本地 Java 接口

风险描述：Android 的 WebView 组件有一个非常特殊的接口函数 addJavascriptInterface，能实现本地 Java 与 JavaScript 之间的交互。

危害描述：在 targetSdkVersion 小于 17 时，攻击者利用 addJavascriptInterface 这个接口添加的函数，可以远程执行任意代码。

修复建议：建议开发者不要使用 addJavascriptInterface，应使用注入 JavaScript 和第三方协议的替代方案。

8. SSL 通信服务端检测信任任意证书

风险描述：开发者重写了 checkServerTrusted 方法，但是在方法内没有做任何服务端的证书校验。

危害描述：黑客可以使用中间人攻击获取加密内容。

修复建议：对服务端和客户端的证书进行严格校验，出现异常事件时不要直接返回空值。

9. 隐式意图调用

风险描述：封装 Intent 时采用隐式设置，只设定 action，未限定具体的接收对象，导致 Intent 可被其他应用获取并读取其中的数据。

危害描述：Intent 隐式调用发送的意图可能被第三方劫持，导致内部隐私数据泄露。

修复建议：将隐式调用改为显式调用。

10. WebView 忽略 SSL 证书错误

风险描述：WebView 调用 onReceivedSslError 方法时，直接执行 handler. proceed()来忽略该证书错误。

危害描述：忽略 SSL 证书错误可能引起中间人攻击。

修复建议：不要重写 onReceivedSslError 方法，或者对于 SSL 证书错误问题按照业务场景判断，避免数据明文传输。

11. HTTPS 关闭主机名验证

风险描述：构造 HttpClient 时，设置 HostnameVerifier 参数使用 ALLOW_ALL_HOSTNAME_VERIFIER 或空的 HostnameVerifier。

危害描述：关闭主机名校验可以导致黑客使用中间人攻击，获取加密内容。

修复建议：设置 HostnameVerifier 时，使用 STRICT_HOSTNAME_VERIFIER 替代 ALLOW_ALL_HOSTNAME_VERIFIER 来进行证书严格校验；自定义实现 HostnameVerifier 时，在实现的 verify 方法中对 Hostname 进行严格校验。

12. Intent Scheme URL 攻击

风险描述：在 AndroidManifast. xml 设置 Scheme 协议之后，可以通过浏览器打开对应的 Activity。

危害描述：攻击者通过访问浏览器构造 Intent 语法唤起应用的相应组件，轻则引起拒绝服务，重则可能演变为提权漏洞。

修复建议：配置 category filter，以添加 android. intent. category. BROWSABLE 方式规避风险。

13. 全局文件可读

风险描述：应用为内部存储文件设置了全局的可读权限。

危害描述：攻击者恶意读取文件内容，获取敏感信息。

修复建议：开发者在读取文件时尽量不要为文件设置全局可读属性，以避免敏感数据通过外部方式被读取。

14. 全局文件可写

风险描述：应用在写内部文件时为文件设置了全局可写权限。

危害描述：攻击者恶意写文件内容，破坏应用的完整性。

修复建议：开发者在写文件时不要使用全局可写权限，以避免关键文件通过外部方式被篡改。

15. 全局文件可读可写

风险描述：应用在创建内部存储文件时为文件设置了全局的可读写权限。

危害描述：攻击者恶意写文件内容，破坏应用的完整性；或者攻击者恶意读取文件内容，获取敏感信息。

修复建议：去掉敏感文件的全局可读写属性，以避免关键数据被攻击者篡改或者读取。

16. SSL 通信客户端检测信任任意证书

风险描述：用户自定义了 TrustManager 并重写 checkClientTrusted()方法，在方法内不对任何服务端进行证书校验。

危害描述：黑客可以使用中间人攻击获取加密内容。

修复建议：不要在 checkClientTrusted()方法内直接返回空值，需要对服务端和客户端的证书进行严格校验。

17. 配置文件可读

风险描述：使用 getSharedPreferences()方法打开配置文件时，第二个参数被设置为 MODE_WORLD_READABLE。

危害描述：当前文件可以被其他应用读取，导致信息泄露。

修复建议：使用 getSharedPreferences()方法读取配置文件时使用 MODE_PRIVATE 参数；如果需要将配置文件提供给其他程序使用，则需要保证相关的数据是经过加密的或者是非隐私数据。

18. 配置文件可写

风险描述：getSharedPreferences()方法读取配置文件时使用了 MODE_WORLD_WRITEABLE 参数。

危害描述：其他应用可以写入该配置文件，导致配置文件内容被篡改，从而导致应用程序的正常运行受到影响或更严重的安全问题。

修复建议：使用 getSharedPreferences()方法读取配置文件时，控制参数必须设置为 MODE_PRIVATE。

19. 配置文件可读可写

风险描述：调用 getSharedPreferences()方法打开配置文件时，将 MODE_WORLD_READABLE | MODE_WORLD_WRITEABLE 作为控制参数。

危害描述：打开的配置文件可以被外部应用随意读取和写入，导致文件信息泄露、配置文件的内容被篡改，从而影响应用程序的正常运行或者导致更加严重的问题。

修复建议：打开配置文件时，使用 MODE_PRIVATE 代替 MODE_WORLD_READABLE | MODE_WORLD_WRITEABLE。

20. Dex 文件动态加载

风险描述：使用 DexClassLoader 加载外部的 Apk、Jar 或 Dex 文件，当外部文件的来源无法控制或是被篡改时，无法保证加载文件的安全性。

危害描述：加载恶意的 Dex 文件将会导致任意命令的执行。

修复建议：加载外部文件前，必须使用校验签名或 MD5 等方式确认外部文件的安全性。

21. AES 弱加密

风险描述：在 AES 加密时，使用“AES/ECB/NoPadding”或“AES/ECB/PKCS5padding”的模式。

危害描述：ECB 是将文件分块后对文件块做同一加密，破解加密只需要针对一个文件块进行解密，降低了破解难度和文件安全性。

修复建议：禁止使用 AES 加密的 ECB 模式，显式指定加密算法为 CBC 或 CFB 模式，可带上 PKCS5Padding 填充。AES 密钥长度最少是 128 位，推荐使用 256 位。

22. Provider 文件目录遍历

风险描述：当 Provider 被导出且覆写了 openFile()方法时，没有对 Content Query Uri 进行有效判断或过滤。

危害描述：攻击者可以利用 openFile()方法进行文件目录遍历，以达到访问任意可读文件的目的。

修复建议：一般情况下无须覆写 openFile()方法，如果必要，校验提交的参数中是否存在“../”等目录跳转符。

23. Activity 绑定 browserable 与自定义协议

风险描述：Activity 设置 android.intent.category.BROWSABLE 属性并同时设置自定义的协议 android:scheme，这意味着可以通过浏览器使用自定义协议打开此 Activity。

危害描述：可能通过浏览器对应用进行越权调用。

修复建议：应用对外部调用过程和传输数据进行安全检查或检验。

24. 动态注册广播

风险描述：使用 registerReceiver 动态注册的广播在组件的生命周期中是默认导出的。

危害描述：导出的广播可以导致拒绝服务、数据泄露或是越权调用。

修复建议：使用带权限检验的 registerReceiver API 进行动态广播的注册。

25. 开放 socket 端口

风险描述：应用绑定端口进行监听，建立连接后可接收外部发送的数据。

危害描述：攻击者可构造恶意数据对端口进行测试，对于绑定了 IP 0.0.0.0 的应用可发起远程攻击。

修复建议：如无必要，只绑定本地 IP 127.0.0.1，并且对接收的数据进行过滤、验证。

26. Fragment 注入

风险描述：通过导出的 PreferenceActivity 的子类，没有正确处理 Intent 的 extra 值。

危害描述：攻击者可绕过限制访问未授权的界面。

修复建议：当 targetSdk 大于或等于 19 时，强制实现了 isValidFragment()方法；当 targetSdk 小于 19 时，在 PreferenceActivity 的子类中都要加入 isValidFragment()，两种情况下都要在 isValidFragment()方法中进行 fragment 名的合法性校验。

27. WebView 启用访问文件数据

风险描述：WebView 中使用 setAllowFileAccess(true)，应用可通过 WebView 访问私有目录下的文件数据。

危害描述：在 Android 中，mWebView. setAllowFileAccess(true)为默认设置。当 setAllowFileAccess(true)时，在 File 域下，可执行任意的 JavaScript 代码，如果绕过同源策略对私有目录文件进行访问，会造成用户隐私泄露。

修复建议：使用 WebView. getSettings(). setAllowFileAccess(false)来禁止访问私有文件数据。

28. Unzip 解压缩(ZipperDown)

风险描述：解压 zip 文件，使用 getName()获取压缩文件名后未对名称进行校验。

危害描述：攻击者可构造恶意 zip 文件，被解压的文件将进行目录跳转而被解压到其他目录，覆盖相应文件导致任意代码执行。

修复建议：解压文件时，判断文件名是否含有特殊字符“../”。

29. 未使用编译器堆栈保护技术

风险描述：为了检测栈中的溢出，引入了 Stack Canaries 漏洞缓解技术。在所有函数调用发生时，向栈帧内压入一个额外的被称作 canary 的随机数，当栈中发生溢出时，canary 将被首先覆盖，之后才是 EBP 和返回地址。在函数返回之前，系统将执行一个额外的安全验证操作，将栈帧中原先存放的 canary 和. data 中副本的值进行比较，如果两者不吻合，则说明发生了栈溢出。

危害描述：不使用 Stack Canaries 栈保护技术，发生栈溢出时系统并不会对程序进行保护。

修复建议：使用 NDK 编译 so 时，在 Android. mk 文件中添加：

```
LOCAL_CFLAGS := -Wall-O2-U_FORTIFY_SOURCE-fstack-protector-all
```

30. 未使用地址空间随机化技术

风险描述：PIE 全称为 Position Independent Executables，是一种地址空间随机化技术。当 so 被加载时，在内存里的地址是随机分配的。

危害描述：不使用 PIE，将会使得 shellcode 的执行难度降低，攻击成功率增加。

修复建议：使用 NDK 编译 so 时，加入“LOCAL_CFLAGS:=-fpie -pie”，开启对 PIE 的支持。

31. 动态链接库中包含执行命令函数

风险描述：在 Native 程序中，有时需要执行系统命令，在接收外部传入的参数执行命令时没有做过滤或检验。

危害描述：攻击者传入任意命令，导致恶意命令的执行。

修复建议：对传入的参数进行严格的过滤。

32. 随机数不安全使用

风险描述：调用 SecureRandom 类中的 setSeed()方法。

危害描述：生成的随机数具有确定性，存在被破解的可能性。

修复建议：使用/dev/urandom 或者/dev/random 来初始化伪随机数生成器。

33. FFmpeg 文件读取

风险描述：使用了低版本的 FFmpeg 库进行视频解码。

危害描述：在 FFmpeg 的某些版本中可能存在本地文件读取漏洞，可以通过构造恶意文件获取本地文件内容。

修复建议：升级 FFmpeg 库到最新版。

34. Libupnp 栈溢出漏洞

风险描述：使用了低于 1.6.18 版本的 Libupnp 库文件。

危害描述：构造恶意数据包可造成缓冲区溢出，造成数据包中的恶意代码被系统意外执行。

修复建议：升级 Libupnp 库到 1.6.18 版本或以上。

35. AES/DES 硬编码密钥

风险描述：使用 AES 或 DES 加解密时，采用硬编码在程序中的密钥。

危害描述：通过反编译拿到密钥，可以轻易解密应用通信数据。

修复建议：将密钥进行加密存储或者将密钥进行变形后再用于加解密运算，切勿硬编码到代码中。

3.2.2 Android 权限安全

Android 应用运行过程中会申请使用手机设备各组件的权限，有些权限会被非法用于访问用户手机中的敏感信息，或被非法用于对用户的手机进行攻击。Android 的安全架构设计是默认情况下，任何应用都没有权限执行对用户、操作系统或其他应用有不利影响的任何操作。如果应用需要申请某种权限时，必须告知用户，并由用户决定是否赋予应用该权限。

自 Android 6.0 版本开始，权限被分为正常权限和危险权限。正常权限通常是访问对用户隐私或其他应用操作风险较小的区域，危险权限涵盖应用需要涉及用户隐私信息的数据或资源，或者可能对用户存储的数据或其他应用的操作产生影响的区域。例如，能够读取用户的联系人属于危险权限。如果应用声明需要某些危险权限，则需要由用户明确对该权限进行授予。

表 3.1 列出了危险权限与权限组。

表 3.1　危险权限与权限组

权　限　组	危 险 权 限
CALENDAR	READ_CALENDAR
	WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS
	WRITE_CONTACTS
	GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION
	ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE
	CALL_PHONE
	READ_CALL_LOG
	WRITE_CALL_LOG
	ADD_VOICEMAIL
	USE_SIP
	PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS
	RECEIVE_SMS
	READ_SMS
	RECEIVE_WAP_PUSH
	RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE
	WRITE_EXTERNAL_STORAGE

Android 权限除了常见的正常权限与危险权限外，还有 signature 和 signatureOrSystem 两种。在介绍后两种权限等级之前，先介绍两种应用分类：系统应用(System App)与特权应用(Privilege App)。

1. 系统应用

在 PackageManagerService 中，判断具有 ApplicationInfo. FLAG_SYSTEM 标记的，被视为系统应用。一般来说，系统应用有两种类型：shared uid 为 android. uid. system、android. uid. phone、android. uid. log、android. uid. nfc、android. uid. bluetooth、android. uid. shell，这类应用都被赋予了 ApplicationInfo. FLAG_SYSTEM 标志；还有一种处于特定目录，比如/vendor/overlay、/system/framework、/system/priv-app、/system/app、/vendor/app、/oem/app，这些目录中的应用都被视为系统应用。

2. 特权应用

特权应用可以使用 protectionLevel 为 signatureOrSystem 或者 protectionLevel 为 signature | privileged 的权限。PackageManagerService 通过判断应用是否具有 ApplicationInfo. PRIVATE _FLAG_PRIVILEGED 标志来判断是否为特权应用。特权应用首先是系统应用，

也就是说,前面提到的一些系统应用会被赋予特权权限。直观来说,目录/system/priv-app下的应用都是特权应用。

权限等级为 signature 的含义是只有当请求该权限的应用具有与声明权限的应用具有相同的签名时,才会授予该权限给应用,并且不用弹窗通知用户或者征求用户同意。权限为signatureOrSystem 的含义是请求权限的应用与声明权限的应用签名相同或者请求权限的应用是系统应用。

视频 5

3.3 OWASP 移动平台十大安全问题

OWASP 在 2016 年的时候提出了 Android 移动平台的十大安全问题,分别是平台使用不当、不安全数据存储、不安全通信、不安全的认证、加密不足、不安全的授权、客户端的代码质量、代码篡改、逆向工程、多余的功能,并通过这 10 个方面规定了如何在代码方面防范软件安全问题。

本节将基于 OWASP 所提出的十大安全问题,详细讨论在 Android 编程开发中经常容易忽略的漏洞以及应对方法。

1. 组件安全

Android 有 Activity、Service、ContentProvider、Broadcast Receiver、Intent 五大组件,对这些组件的使用过程中如果出现配置或编码不规范,很有可能会造成组件恶意调用、恶意广播、恶意启动应用服务、恶意调用组件、恶意拦截数据、恶意代码的远程执行等。

1) 组件暴露风险

对于不需要进行跨应用运行的组件,如果在 AndroidManifest. xml 配置文件中将其属性 android:exported 设置为 true,则该组件可以被任意应用启动,存在被恶意调用的风险。

```
<activity
    android:name = "com.example.haohai.HelloWorldActivity"
    android:exported = "true"
    android:label = "@string/app_name" >
</activity>
```

为避免此风险,需要将组件的 android:exported 设置为 false,这样该组件只能被同一应用程序的组件或者属于同一用户的应用程序所启动或绑定。

```
<activity
    android:name = "com.example.haohai.HelloWorldActivity"
    android:exported = "false"
    android:label = "@string/app_name" >
</activity>
```

2) 公开组件的访问权限

针对需要被其他应用访问而将 android:exported 属性设置为 true 的组件,为了防止被未授权或者恶意应用调用,可以使用 android:permission 属性指定自定义权限。

自定义权限：

```
<permission
android:name = "example.permission.USESERVICE"
android:protectionLevel = "normal"/>
```

为组件设置自定义权限：

```
<service
      android:name = "com.example.haohai.HelloWorldService"
      android:exported = "true"
      android:label = "@string/app_name"
      android:permission = "example.permission.USESERVICE" >
</service>
```

如果应用需要调用 HelloWorldActivity，则需要在 AndroidManifest.xml 中声明权限，否则系统就会抛出 SecurityException。

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

3）ContentProvider 数据权限

ContentProvider 组件可以为外部应用提供统一的数据存储和读取接口，如果不对数据的操作严格控制权限，就可能会造成数据泄露或数据完整性被破坏等风险。可以针对 ContentProvider 组件设置全局的读写访问控制权限，也可以针对某个路径下的文件访问添加自定义权限。

针对 Apk 目录下的文件添加读取权限：

```
<provider
      android:name = "com.example.haohai.HelloWorldProvider"
      android:authorities = "com.example.haohai.HelloWorldProvider">
      <path-permission
            android:pathPattern = "/Apk/.*"
            android:readPermission = "com.example.haohai.permission.READ"
            android:protectionLevel = "normal" />
</provider>
```

设置全局可读权限：

```
<provider
    android:name = "com.example.haohai.HelloWorldProvider"
    android:authorities = "com.example.haohai.HelloWorldProvider"
    android:readPermission = "com.example.haohai.permission.READ">
</provider>
```

4) Intent 调用风险

Intent 在进行组件间的跳转时有两种调用方式：一个是显式调用，即通过指定 Intent 组件的名称，使用 Intent. setComponent()、Intent. setClassName()、Intent. setClass()方法进行目标组件的指向或者在 Intent 对象初始化“new Intent(A. class,B. class)”时指明需要转向的组件，一般在应用程序内部组件跳转时使用；另一个是隐式调用，通过在配置文件中设置 Intent Filter 实现，Android 系统会根据设置的 action、category、数据等隐式意图来进行组件跳转。隐式调用一般用于不同应用程序之间的组件跳转。

由于隐式调用是由系统根据意图来判断最匹配的组件，存在判断失误的可能性，有导致数据泄露的风险。因此为了数据安全，应尽量减少隐式调用，尽量使用显式调用。

显式调用的代码：

```
Intent intent = new Intent(HelloWorldActivity.this, TargetActivity.class);
startActivity(intent);
```

2. 数据存储安全

对平台功能的误用以及安全控件的失败使用会威胁到 SharedPreference 数据储存安全、密码储存安全、sdcard 数据储存安全，产生数据泄露或数据完整性被破坏等风险。

1) SharedPreference 数据存储安全

SharedPreference 是 Android 中轻量级的数据存储方式，其内部使用键-值对的方式进行存储，以 XML 格式的结构保存在/data/data/packageName/shared_prefs 目录下，一般用来保存一些简单的数据类型。SharedPreference 存储时可以设定存储模式，MODE_WORLD_READABLE 表示当前文件可以被其他应用读取，MODE_WORLD_WRITEABLE 表示当前文件可以被其他应用写入。这两个操作模式在 Android 4.2 以上的版本已经被弃用。

MODE_PRIVATE 表示当前文件使用私有化存储模式，只能被应用本身访问，写入内容时会覆盖原文件的内容。在 MODE_APPEND 模式下会向文件中追加内容。建议在使用 SharedPreference 存储时设定为 MODE_PRIVATE，以防止数据被恶意应用修改或泄露。

```
SharedPreferences mySharedPreferences = getSharedPreferences("HelloWorld", Activity.MODE_
PRIVATE);
```

2) 密码存储安全

在某些情况下需要将密码存储在本地，为了预防设备被 Root 后受保护目录被随意访问造成密码泄露，可以对密码进行哈希处理并保存密码的信息摘要。当需要进行密码匹配时，直接将用户输入的密码进行哈希处理，通过两个摘要的比对实现密码校验，以避免直接将密码明文或弱加密的密码保存在本地。

3) 避免使用外部存储

外部存储一般是指 sdcard，任何有 sdcard 访问权限的应用都可以访问 sdcard。如果对关键信息使用外部存储以实现数据持久化，容易导致数据泄露。重要的数据尽可能保存在

应用的私有目录下或者使用 Sqlite 和 SharedPreferences 进行数据存储。

使用 Sqlite 进行增、删、改、查：

```
public void addData(DataType data){
    myDataBase.beginTransaction();
    ContentValues contentValues = new ContentValues();
    contentValues.put("id",data.getId());
    contentValues.put("name",data.getName());
    contentValues.put("number",data.getNumber());
    myDataBase.insertOrThrow(MySqliteHelper.TABLE_NAME,null,contentValues);
    myDataBase.setTransactionSuccessful();
    myDataBase.endTransaction();
}
```

```
public void deleteData(String id){
    myDataBase.beginTransaction();
    myDataBase.delete(MySqliteHelper.TABLE_NAME, "id = ?", new String[]{id});
    myDataBase.setTransactionSuccessful();
}
```

```
public void updateData(ContentValues contentValues,String id){
    myDataBase.beginTransaction();
    myDataBase.update(MySqliteHelper.TABLE_NAME,contentValues,"id = ?",new String[]{id});
    myDataBase.setTransactionSuccessful();
}
```

```
public List<Data> getDatalist() {
    Cursor cursor = myDataBase.query(MySqliteHelper.TABLE_NAME,
        new String[]{"name","number"},
        "id = ?",
        new String[]{"1"}, null, null, null);
    if (cursor.getCount() > 0) {
      List<Data> dataList = new ArrayList<Data>(cursor.getCount());
      while (cursor.moveToNext()) {
        Data data = parseData(cursor);
        dataList.add(data);
      }
      cursor.close();
      return dataList;
    }
    return null;
}
```

3. 通信安全

在编写 Android 程序的过程中使用不安全的方式在客户端与服务端之间进行数据传输或业务交互，会导致服务端数据泄露的风险。为了保护客户端与服务端之间的通信安全，在

使用 HTTP 进行会话时,建议将 session ID 设置在 Cookie 头中,服务器根据该 session ID 获取对应的 Session,而不是重新创建一个新 Session。当客户端访问一个使用 Session 的站点,同时在自己机器上建立一个 Cookie 时,如果未使用服务端的 Session 机制进行会话通信,则可能造成服务端存储的数据存在被任意访问的风险。

Java. net. HttpURLConnection 获取 Cookie:

```
URL url = new URL("request_url");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
String cookie_string = connection.getHeaderField("set-cookie");
String sessionid;
if (cookie_string != null) {
  sessionid = cookie_string.substring(0, cookie_string.indexOf(";"));
}
```

Java. net. HttpURLConnection 发送设置 Cookie:

```
URL url = new URL("request_url");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
if (sessionid != null) {
  con.setRequestProperty("cookie", sessionid);
}
```

org. apache. http. client. HttpClient 设置 Cookie:

```
HttpClient http = new DefaultHttpClient();
HttpGet httppost = new HttpGet("url");
httppost.addHeader("cookie", sessionId);
```

4. 认证安全

在 Android 应用中,如果仅通过客户端来为用户验证或授权,那么在没有其他安全措施的前提下,存在着不安全认证的风险。最典型的是 WebView 的自动保存密码功能。WebView 是一个基于 WebKit 引擎展现 Web 页面的控件,用于渲染和显示网页。WebKit 引擎提供了 WebView 控制网页前进后退、放大缩小、搜索网址等功能。WebView 可以在 App 中嵌入显示网页,也可以开发浏览器。

WebView 组件中自带有记住密码的功能,为网页上的账号密码登录提供便利,然而这个功能会将密码以明文的形式保存在/data/data/com. package. name/databases/webview. db 中,当设备被 Root 后,获得 Root 权限的应用可以直接读取被 WebView 记住的密码,造成密码泄露。因此,在使用 WebView 时应当关闭 WebView 的自动保存密码的功能。

```
myWebView.getSettings().setSavePassword(false);
```

5. 数据加密

在数据存储目录保护措施不足的情况下,对数据采用弱加密、不规范使用加密算法、用

硬编码的方法存储密钥等就可能导致敏感数据被破解与窃取。

下面介绍几种常见的加密算法。

1）MD5

MD2与MD4由于存在缺陷，现在已不再使用。MD5（Message Digest Algorithm，消息摘要算法）产生于1991年，是现在广泛使用的版本，但由于碰撞算法的出现，导致MD5的安全性开始受到质疑。因此不建议对密码等敏感数据使用MD5加密。

2）SHA-1

SHA算法是哈希算法的一种，表示加密哈希算法，产生不可逆的和独特的哈希值，两个不同的数据不能产生同样的哈希值。SHA-1产生的是160位的哈希值，而它的继承者SHA-2采用多种位数的组合值，于2016年起替代SHA-1成为新的标准。推荐使用其中最受欢迎的SHA-256。SHA-1已经被淘汰，因此不建议对密码等敏感数据进行加密。

3）PIPEMD-160

PIPEMD-160是于1996年设计出的一种能够产生160位的哈希值的单向哈希函数，是欧盟PIPE项目所设计的RIPEMD单向哈希函数的修订版，这一系列的函数还包括PIPEMD-128、PIPEMD-256、PIPEMD-320等。比特币使用的就是PIPEMD-160。PIPEMD的强抗碰撞性已经于2004年被攻破，PIPEMD-160尚未被攻破。但是在CRYPTREC密码清单中，PIPEMD-160已经被列入“可谨慎运用的密码清单”，即除了用于保持兼容性的目的以外，其他情况都不推荐使用。

4）SHA-3

在SHA-1的强抗碰撞性被攻破后，NIST开始指定取代SHA-1的下一代单向哈希函数SHA-3的标准。2012年，Keccak算法在公开竞争中胜出并被标准化成为SHA-3。Keccak算法设计简单，硬件实现方便，且根据第三方密码分析，Keccak没有非常严重的弱点且抗碰撞性好。

根据上面对常见的几种加密算法的分析，建议对敏感信息使用SHA-256或SHA-3加密，不建议使用MD5、SHA-1、PIPEMD进行处理，以免被破解。

对于某些数据或文件需要使用DES或AES等密钥加密算法进行处理的情况，需要特别注意所使用密钥的保存。常用的加密算法都是公开的，加密内容的保密依赖于密钥的保密，如果密钥泄露，那么很可能会导致加密内容被破解与窃取。有些开发者为了贪图方便，将密钥硬编码保存在代码中，特别是Java等代码被反编译后与源码无异的语言，硬编码的密钥比较容易暴露。通常对密钥的保护是使用SharedPreference对密钥进行存储，并对密钥进行加密处理，或者是加密存储在应用目录下。也可以将密钥保存在so文件中，把加密解密操作放在Native层进行，并对so文件进行加固保护。

6. 授权安全

在Android应用中有许多操作与数据会与设备绑定，在执行操作或访问数据之前需要对用户权限进行检测。在没有适当的安全措施的情况下，只通过客户端检测用户是否有权限访问数据或执行操作，会出现信息伪造、数据替换等风险。

1）分配唯一ID

通过为每个用户分配一个唯一的ID，可以对所有访问关键数据和敏感操作进行追溯，并且这个唯一的ID应当是不可伪造的。IMEI国际移动设备识别码在移动电话网络中唯

一标识了每台独立的移动通信设备,通常用来生成唯一的识别 ID。但是不应该将 IMEI 直接作为唯一 ID,因为 Android 模拟器本身没有 IMEI 号,但它可以模拟或伪造 IMEI,如果直接将 IMEI 作为唯一 ID,会出现被模拟伪造的风险。为避免该风险,应该使用 DEVICE_ID、MAC ADDRESS、Sim Serial Number、IMEI 等多条数据进行组装后生成的哈希值来作为设备的唯一 ID。

```
//获取 DEVICE_ID
TelephonyManager tm = (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);
String DEVICE_ID = tm.getDeviceId();

//获取 MAC ADDRESS
WifiManager wifi = (WifiManager) getSystemService(Context.WIFI_SERVICE);
WifiInfo info = wifi.getConnectionInfo();
String macAdress = info.getMacAddress();

//获取 Sim Serial Number
TelephonyManager tm = (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);
String SimSerialNumber = tm.getSimSerialNumber();

//获取 IMEI
String IMEI = ((TelephonyManager) getSystemService(TELEPHONY_SERVICE)).getDeviceId();
```

2) ID 与数据的绑定

如果生成的用户的唯一 ID 没有与敏感数据绑定,那么当数据被复制到其他终端上后仍然可以被使用,这可能造成数据的盗用。从上面的数据加密中得到的密钥可以与其他数据组合形成唯一 ID,从而将数据与设备绑定在一起。在数据被解密的时候,如果唯一 ID 不匹配,则会导致数据解密的失败,因此能有效预防数据被盗用的风险。

7. 客户端代码质量

客户端的编码质量有时也会导致一些潜在的风险,比如,对组件间传递的参数没有进行非空验证,可能会因为空参数导致应用崩溃;设置 targetSdkVersion 版本过低导致调用过时的不安全 API,从而造成的风险;应用中错误的日志输出信息导致重要代码逻辑暴露和敏感数据泄露风险。

1) 参数非空验证

Activity 等组件根据其业务需要会对外部应用开放,而从外部应用传入参数的情况会比较复杂,如果不对传入的参数做检测,则会导致传入异常参数,进而导致应用崩溃。因此,对外开放的组件要严格检验输入的参数,需要注意判断空值与数据类型,以防范异常参数导致的应用崩溃风险。

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Intent intent = getIntent();
    if (intent == null){
```

```
        return;
    }
    Bundle mBundle = intent.getExtras();
    if (mBundle == null){
        return;
    }
    String getValue = mBundle.getString("value");
    if (getValue == null){
        return;
    }
}
```

2）targetSdkVersion 版本过低

targetSdkVersion 是 Android 系统提供前向兼容的主要手段，随着 Android 系统版本的升级，某个 API 或组件的实现会发生改变，包括性能和安全性的改进。但是为了保证旧的 Apk 可以兼容新的 Android 系统，会根据 Apk 的 targetSdkVersion 调用相应版本的 API。这样即使 Apk 安装在新版本的 Android 系统上，其功能实现仍然按照旧版本的系统来运行，以此保证新版本系统对老版本系统的兼容性。

有些 Android 组件的 API 由于安全性的问题，在新版本的 API 中进行了修改或者移除。比如 WebView 组件支持 Android 原生页面与 Web 页面上的 JavaScript 进行交互，为此提供了一个方法 addJavascriptInterface，这个方法可以暴露一个 Java 对象给 JavaScript，使得 JavaScript 可以直接调用 Java 对象的方法，在 API 17（Android 4.2）之前，这个方法并没有对 JavaScript 的调用范围作出限制，借助 Java 的反射机制 JavaScript 脚本甚至可以执行应用中的任意 Java 代码。

API 17 之前 addJavascriptInterface 的使用方法：

```
//获取网页
myWebView = (WebView) this.findViewById(R.id.mwebview);
myWebView.getSettings().setJavaScriptEnabled(true);
myWebView.loadUrl("file:///Android_asset/index.html");
myWebView.addJavascriptInterface(new JSInterface(), "test_js");
```

这样网页中的 JavaScript 脚本就可以利用接口 test_js 调用应用中的 Java 代码，而如果有恶意网站执行了以下 JavaScript 代码，则可能会产生非常严重的后果。

```
function execute(cmdArgs)
{
    for (var obj in window) {
       if ("getClass" in window[obj]) {
          alert(obj);
          return window[obj].getClass().forName("java.lang.Runtime")
               .getMethod("getRuntime",null).invoke(null,null).exec(cmdArgs);
        }
    }
}
```

这段 JavaScript 代码可以遍历 Windows 对象,找到其中存在 getClass()方法的对象,通过反射机制获取 Runtime 对象。每个 JVM 进程中都对应一个 Runtime 实例,是由 JVM 负责实例化的单例,该对象只能由 getRuntime()方法获得,而一旦获得 Runtime 对象,就可以调用 Runtime 的方法去查看 JVM 的状态或者控制 JVM 的行为,比如访问本地文件并从执行命令后返回的输入流中获得文件信息。

API 17 以后的版本,需要为每条 JavaScript 调用的 Java 方法添加@JavascriptInterface 注解。如果没有添加注解的 Java 方法,则不会被 JavaScript 反射调用。

```
@SuppressLint("JavascriptInterface")
@JavascriptInterface
```

为了避免恶意 JavaScript 脚本远程执行以及其他 WebView 安全漏洞,推荐将 targetSdkVersion 设置高于 17。如果 targetSdkVersion 设置低于 17,那么程序在运行时会根据 targetSdkVersion 的设置选择旧版本 API 进行调用,这样就会产生安全风险。

3) 不正确的日志输出信息

在应用开发过程或者应用维护的过程中,为了把握应用运行状态,一般会在代码中插入日志信息的输出。如果日志的输出不遵循安全编码的规范,且在发布应用前没有将日志输出清理干净,那么这些残留的日志信息就有可能会暴露应用的运行逻辑,为应用的破解提供突破口。

日志打印输出建议遵循的编码规范:

(1) 不推荐使用 System.out/err 输出日志,推荐 android.util.Log 类输出日志。

(2) Log.e()/Log.w()/Log.i()打印操作日志。

(3) Log.d()/Log.v()建议打印开发日志。

(4) 敏感信息的打印建议使用 Log.d()/Log.v(),不建议使用 Log.e()/Log.w()/Log.i()。

(5) 不建议将日志输出到外部存储,防止被其他应用访问与读写。

(6) 公开的应用应该是日志较少的发行版而不是有许多调试日志的开发版。

(7) 建议使用全局变量控制 Log.w()的输出。

在 Android 应用开发中,建议使用 Proguard 配置文件控制开发版应用去除 Log 信息。首先在 gradle 中进行如下配置:

```
//在 gradle 中的配置
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-
rules.pro'
    }
}
```

之后在 proguard-rules.pro 文件中添加优化项如下:

```
-assumenosideeffects class android.util.Log{
    public static *** v(...);
    public static *** i(...);
    public static *** d(...);
    public static *** w(...);
    public static *** e(...);
}
```

这样，在打包 release 版本的时候就可以移除所有的 Log 日志相关语句。

8. 代码篡改防范

当 Android 应用安装到设备后，应用的代码和数据资源就已经存放在设备存储中了，攻击者可以通过修改应用代码、篡改应用程序使用的系统 API、拦截修改应用内存数据等方法，颠覆 Android 应用的运行过程以及结果，从而达到不正当的目的。代码篡改会造成的风险包括二进制修改、本地资源修改、Hook 注入、函数重要业务逻辑篡改。

1）程序完整性

经过二次打包的 Apk 文件中的文件一定是有所修改的，因此二次打包的 Apk 文件会将原本在包内的签名文件删除并重新签名，而同一个应用的不同签名的 MD5 值是不同的，因此可以通过校验签名的 MD5 值来判断应用是否被二次打包过。

```
/获取应用签名
public static String getSignature(Context context) {
   PackageManager pm = context.getPackageManager();
   PackageInfo pi;
   StringBuilder sb = new StringBuilder();

   try {
      pi = pm.getPackageInfo(context.getPackageName(), PackageManager.GET_SIGNATURES);
      Signature[] signatures = pi.signatures;
      for (Signature signature : signatures) {
         sb.append(signature.toCharsString());
      }
   } catch (PackageManager.NameNotFoundException e) {
      e.printStackTrace();
   }

   return sb.toString();
}

//与存放在本地的原加密字符串进行比较
//originalSignature 为原加密字符串
public static boolean verifySignature(Context context, String originalSignature){
   String currentSignature = getSignature(context);
   if (originalSignature.equals(currentSignature)) {
      return true;
   }
   return false;
}
```

2）重要函数逻辑保护

由于 Java 是基于虚拟机技术与解释器的高级编程语言，反编译后生成的伪码逻辑与源码相差不大，甚至部分伪码直接可以作为源码进行再编译运行，相比较于编译成汇编语言的 C/C++语言来说，反编译是相对比较容易的。因此，在不对应用进行加固的情况下，为了保护重要的函数逻辑，推荐将重要的函数放到 Native 层，使用 C/C++语言编写功能逻辑，并编译成 so 文件。

如图 3.1 所示为 Android Java 层的反编译结果。

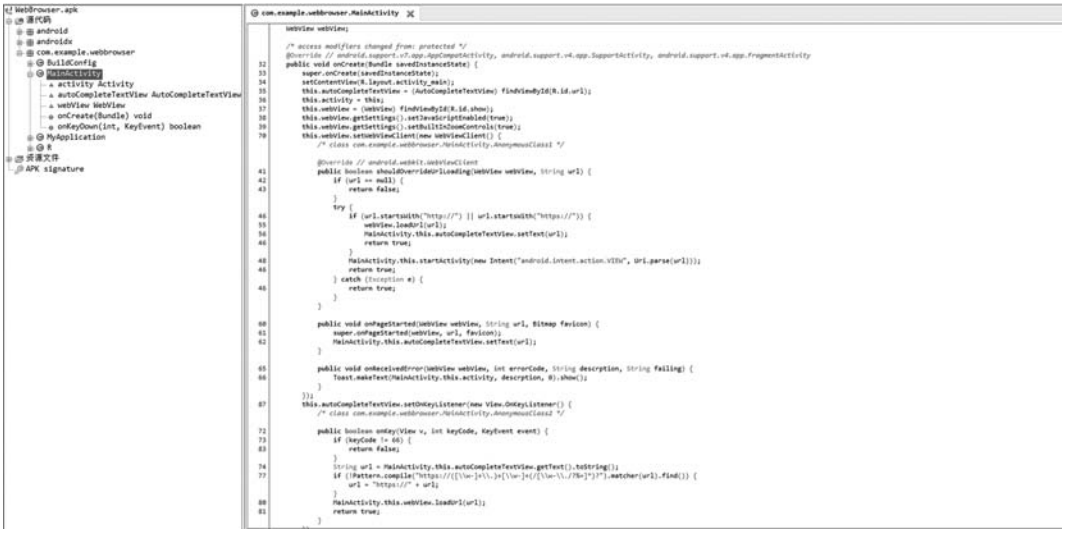

图 3.1　Android Java 层的反编译结果

3）动态加载 Dex 文件风险

Android 系统提供了一种类加载器 DexClassLoader，允许其在应用运行时动态加载并解释执行包含在 Jar 或 Apk 文件内的 Dex 文件。Android 4.1 前的系统版本允许 Android 应用动态加载保存在外部存储中的 Dex 文件，使得该 Dex 文件可能会遭到恶意代码的注入或替换。如果应用没有正确地动态加载 Dex 文件，则会导致恶意代码被执行，进一步产生其他恶意行为。

为了防范动态加载 Dex 文件所带来的风险，建议对于需要动态加载的 Dex 文件保存在 Apk 包内部。对于需要加载的 Dex，建议使用加密网络进行下载，并放置在应用的私有目录下。为了防止攻击者获取 Root 权限后进入应用私有目录对 Dex 文件做手脚，或者使用了没有加密网络的下载源，建议在加载 Dex 前对 Dex 进行完整性校验。

9. 逆向防范

Android 应用逆向工程是指针对 Android 应用中的一些不安全的配置，采用 IDA Pro 和 Apktool 等工具对应用程序进行调试及反编译等行为。逆向工程容易造成核心代码逻辑泄露、核心代码被篡改、内存调试等高危安全风险。

1）关闭调试属性

在 AndroidManifest.xml 中定义的 android:debuggable 属性控制着应用是否可以在调试模式下运行。如果 android:debuggable 设置为 true，则应用可以被调试程序调试，导致代

码执行流程可被追踪，敏感信息存在泄露风险。

建议在 Android 应用发布时，应当将应用的 debuggable 属性显示设置为 false。

如图 3.2 所示为 AndroidManifest.xml 文件中 android:debuggable 的设置。

图 3.2　AndroidManifest.xml 文件中 android:debuggable 的设置

2）Dex 文件保护

未经过保护的 Dex 文件能够轻易地被 Baksmali、Apktool、Jd-gui 等反编译工具逆向出代码，造成核心功能代码的泄露及代码被篡改等风险。

如图 3.3 所示为使用 Baksmali 将 Dex 文件反编译为 Smali 的结果。

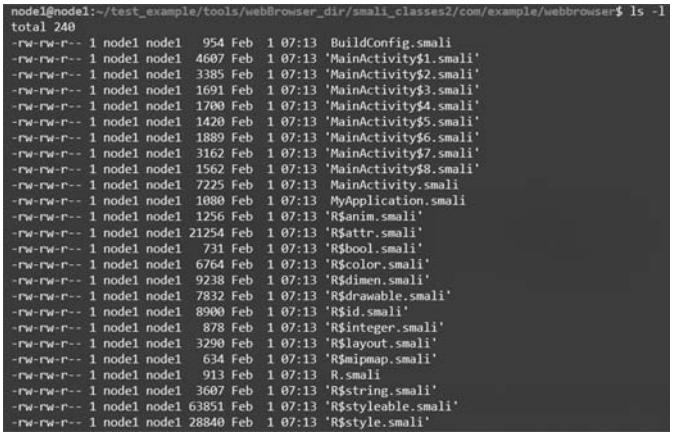

图 3.3　使用 Baksmali 将 Dex 文件反编译为 Smali 的结果

建议对 Dex 文件进行加壳保护，这能够有效地保护 Dex 文件不被反编译工具直接逆向为源代码。

如图 3.4 所示为加固后的 Dex 文件被反编译后的结果。

10. 多余功能

多余功能是指在开发阶段测试时用的数据或者内部调试功能在发布时没有清理干净，带到了发布版本中。这些多余的数据与功能相当于为攻击者打开了后门，造成敏感数据窃取、未授权访问等安全风险。

1）测试数据的移除

如果应用中残留有测试时使用的测试数据（如测试账号等），会造成测试账号或测试信息的外泄。攻击者可以利用测试账号进行未授权访问或攻击。如果测试账号中有重要数据，则会造成重要数据的泄露。

2）内网信息残留

发布版本时残留在程序中的内网信息会导致服务器信息的泄露。内网中的 IP 地址及

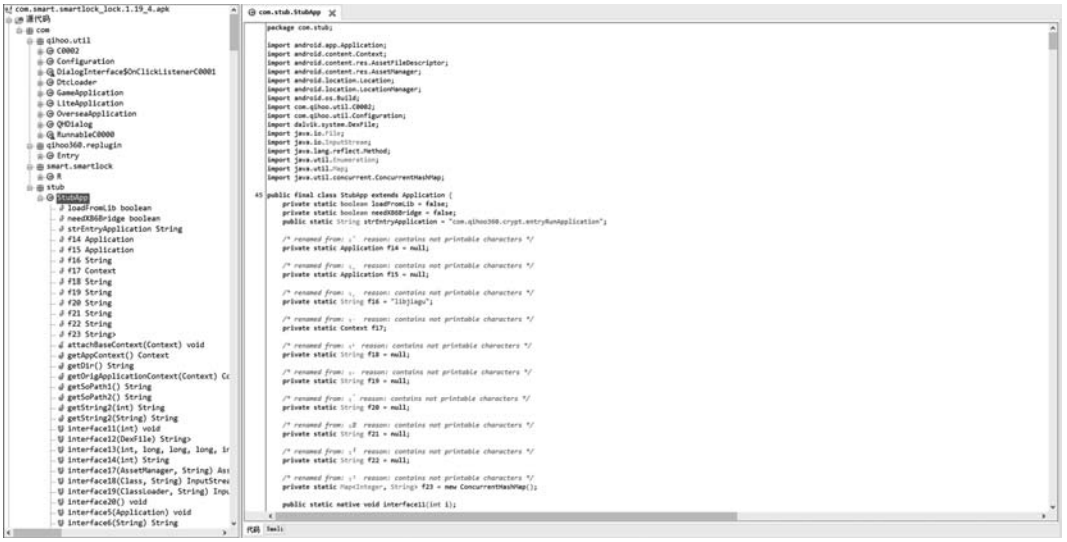

图 3.4　加固后的 Dex 文件被反编译后的结果

测试用的密码等可能被攻击者利用并组织更高强度的内网渗透攻击,或者利用账号密码及证书等辅助对公网服务器端渗透攻击。

3.4　本章小结

本章介绍了 Android 移动应用常见的漏洞以及容易遇到的威胁。应用的漏洞不仅可以为分析者逆向分析 Android 程序提供入手点,也对 Android 安全开发提供参考。结合后面章节介绍的 MobSF 安全框架,读者可以初步建立一个 Android 移动应用安全评估的框架概念。