XPrivacy 分析

MindMac

2013/11/21

XPrivacy 是基于 Xposed 框架的隐私管理工具，官方介绍为：The ultimate, yet easy to use, privacy manager for Android。其源码地址及相关文档：https://github.com/M66B/XPrivacy。

http://www.kanxue.com/bbs/showthread.php?t=181561 给出了 Xposed 框架实现的基本原理。在分析 Xposed 框架的基础上，此文接着对 XPrivacy 进行分析。

在 XposedBridge.main 函数中，完成对所需要 hook 的函数处理后，调用了 loadModules(String)函数，如下代码所示。

```
1.  if (startClassName == null) {
2.      // Initializations for Zygote
3.      initXbridgeZygote();
4.  }
5.  loadModules(startClassName);
```

loadModules 函数对每个基于 Xposed 框架的模块调用 loadModule(String, String)函数，该函数完成对模块的加载工作。不妨先沿此思路分析在 XposedBridge 中完成了哪些加载工作。loadModule 函数的代码如下：

```
1.  private static void loadModule(String apk, String startClassName) {
2.          ......
3.          ClassLoader mcl = new PathClassLoader(apk, BOOTCLASSLOADER);
4.          InputStream is = mcl.getResourceAsStream("assets/xposed_init");
5.          ......
6.          BufferedReader moduleClassesReader = new BufferedReader(new InputStreamReader(is));
7.          try {
8.              String moduleClassName;
9.              while ((moduleClassName = moduleClassesReader.readLine()) != null) {
10.                 moduleClassName = moduleClassName.trim();
11.                 .....
12.                 try {
13.                     ......
14.                     Class<?> moduleClass = mcl.loadClass(moduleClassName);
15.                     // call the init(String) method of the module
16.                     final Object moduleInstance = moduleClass.newInstance();
17.                    if (startClassName == null) {
```

```
18.                  if (moduleInstance instanceof IXposedHookZygoteInit) {
19.                     IXposedHookZygoteInit.StartupParam param = new
    IXposedHookZygoteInit.StartupParam();
20.                      param.modulePath = apk;
21.                     ((IXposedHookZygoteInit) moduleInstance).initZygote(param);
22.                  }
23.                  if (moduleInstance instanceof IXposedHookLoadPackage)
24.    hookLoadPackage(new IXposedHookLoadPackage.Wrapper((IXposedHookLoadPackage) moduleInstance));
25.                  ......
26.               }
27.        ......
28.    }
```

参数 apk 是模块的 APK 文件的路径，此处 apk 值为/data/app/biz.bokhorst.xprivacy-1.apk；startClassName 为所启动类的名称，当启动 Zygote 时，值为 null。第 3 行获取类加载器，第 4 行获取输入流，其中传入的参数值"assets/xposed_init"，该路径指向的文件即为 XPrivacy 源码中 assets/xposed_init，文件中存储后续需要进行初始化的类，对于 XPrivacy 来说，该类为 biz.bokhorst.xprivacy.XPrivacy 。第 6-10 行获取 XPrivacy 中定义的初始化类名，即 biz.bokhorst.xprivacy.XPrivacy。14-16 行加载该类并实例化。18-24 行分别根据该实例的类别执行相应的操作，由于 biz.bokhorst.xprivacy.XPrivacy 类实现了 IXposedHookLoadPackage 和 IXposedHookZygoteInit ，因此此处仅列出了与此相关的两个对应操作。biz.bokhorst.xprivacy.XPrivacy 的类图关系如图 1 所示。

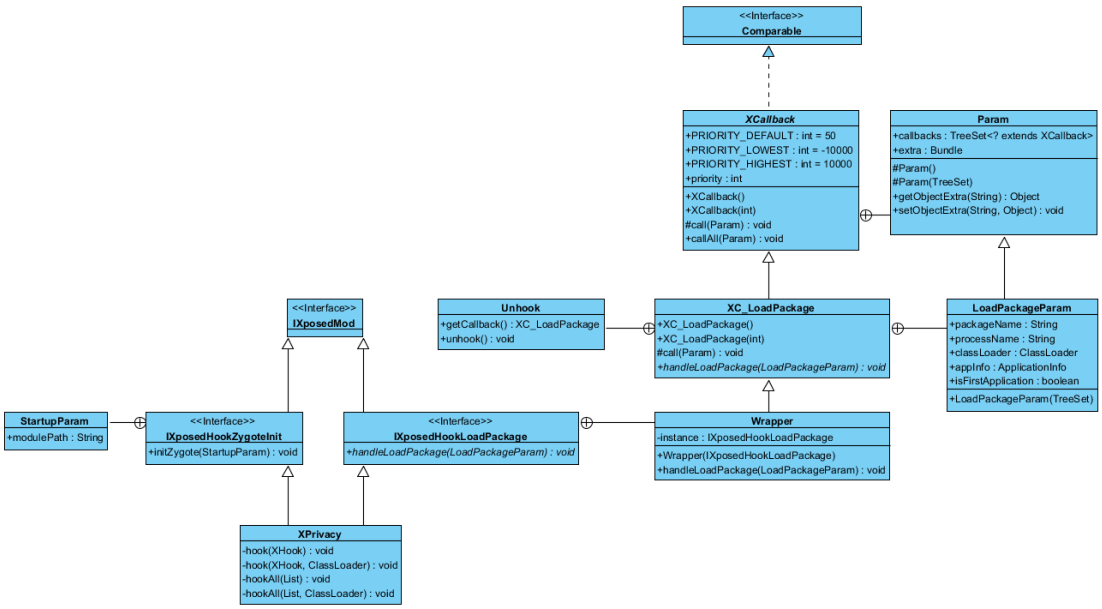

图 1 biz.bokhorst.xprivacy.XPrivacy 类图关系

19-20 行实例化 IXposedHookZygoteInit.StartupParam 并将 modulePath 成员赋值为 XPrivacy 模块 APK 的路径。21 行调用 initZygote 函数完成 Zygote 启动前的函数 hook 的处理。查看 XPrivacy 的 initZygote 函数会发现，该函数调用了 hookAll(List<XHook>)函数完成函数 hook 的处理，一共调用了 33 次 hookAll 函数，分别实现了对 AccountManager、ActivityManager、SmsManager 等 Android 系统类中函数的 hook。

```
1.  public void initZygote(StartupParam startupParam) throws Throwable {
2.      ......
3.      // Account manager
4.      hookAll(XAccountManager.getInstances());
5.      // Activity manager
6.      hookAll(XActivityManager.getInstances());
7.      ......
8.     // SMS manager
9.     hookAll(XSmsManager.getInstances());
10.     // Intent send
11.     hookAll(XActivity.getInstances());
12. }
```

以对 android.telephony.SmsManager 类的 hook 为例进行分析，SmsManager 负责对短信的操作，可参考 http://developer.android.com/reference/android/telephony/SmsManager.html 。hookAll(List<XHook>)函数代码如下：

```
1.  private static void hookAll(List<XHook> listHook) {
2.      for (XHook hook : listHook)
3.          hook(hook);
4.  }
```

该函数遍历参数 listHook，对 listHook 中每个成员调用 hook(XHook)函数，hook(XHook)函数最终会调用 hook(XHook, ClassLoader)函数。在继续分析该函数之前，先分析 XSmsManager 类。XSmsManager 类图关系如图 2 所示。

XSmsManager 中定义了枚举 Methods，存储了可以进行 hook 的函数名，如下代码所示。

```
1.  private enum Methods {
2.      getAllMessagesFromIcc, sendDataMessage, sendMultipartTextMessage, sendTextMessage
```

```
3.  };
```

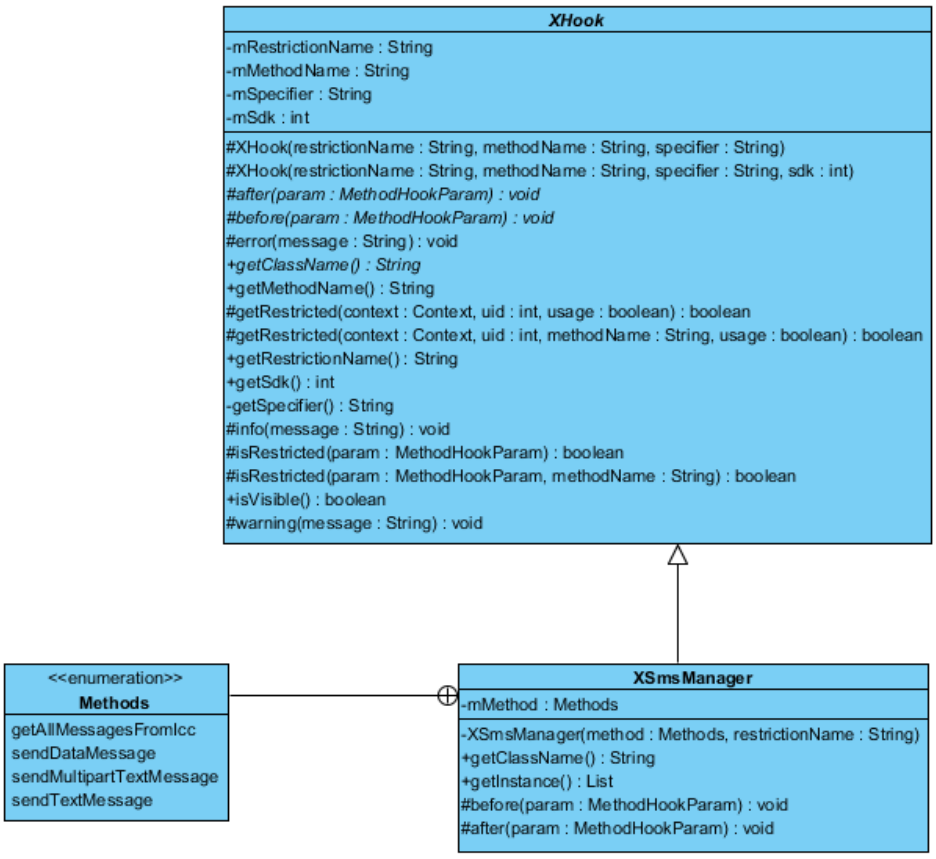

图 2　XSmsManager 类图关系

XPrivacy 实现了对 getAllMessagesFromIcc, sendDataMessage, sendMultipartTextMessage, sendTextMessage 函数的 hook。奇怪的是，getAllMessagesFromIcc 在 Google 官方的 API 文档中找不到，但是在源码中又确实存在 http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.0.4_r2.1/android/telephony/SmsManager.java#SmsManager.getAllMessagesFromIcc%28%29，而且还是个 public 的方法。

getInstances 函数根据上述的 4 个方法实例化 XSmsManager 对象，并添加到 listHook 中，返回 listHook。代码如下：

```
1.  public static List<XHook> getInstances() {
2.      List<XHook> listHook = new ArrayList<XHook>();
3.      listHook.add(new XSmsManager(Methods.getAllMessagesFromIcc, PrivacyManager.cMessages));
4.      listHook.add(new XSmsManager(Methods.sendDataMessage, PrivacyManager.cCalling));
5.      listHook.add(new XSmsManager(Methods.sendMultipartTextMessage, PrivacyManager.cCalling));
6.      listHook.add(new XSmsManager(Methods.sendTextMessage, PrivacyManager.cCalling));
7.      return listHook;
```

```
8.  }
```

现在让我们回到 XPrivacy 的 hook(XHook, ClassLoader)函数中，该函数完成 hook 函数的注册工作。代码如下：

```
1.  private static void hook(final XHook hook, ClassLoader classLoader) {
2.     // Check SDK version
3.     if (Build.VERSION.SDK_INT < hook.getSdk())
4.        return;
5.     try {
6.        // Create hook method
7.        XC_MethodHook methodHook = new XC_MethodHook() {
8.        @Override
9.        protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
10.          ......
11.          if (Process.myUid() <= 0)
12.             return;
13.          hook.before(param);
14.          ......
15.       }
16.       @Override
17.       protected void afterHookedMethod(MethodHookParam param) throws Throwable {
18.          if (!param.hasThrowable())
19.             ......
20.             if (Process.myUid() <= 0)
21.                return;
22.             hook.after(param);
23.             ......
24.          }
25.       };
26.       // Find class
```

```
27.       Class<?> hookClass = findClass(hook.getClassName(), classLoader);
28.       if (hookClass == null) {
29.          ......
30.          return;
31.       }
32.       // Add hook
33.       Set<XC_MethodHook.Unhook> hookSet = new HashSet<XC_MethodHook.Unhook>();
34.       if (hook.getMethodName() == null) {
35.          for (Constructor<?> constructor : hookClass.getDeclaredConstructors())
36.              if (Modifier.isPublic(constructor.getModifiers()) ?
    hook.isVisible() : !hook.isVisible())
37.                  hookSet.add(XposedBridge.hookMethod(constructor, methodHook));
38.          } else
39.             for (Method method : hookClass.getDeclaredMethods())
40.                if (method.getName().equals(hook.getMethodName())
41.                      && (Modifier.isPublic(method.getModifiers()) ?
    hook.isVisible() : !hook.isVisible()))
42.                     hookSet.add(XposedBridge.hookMethod(method, methodHook));
43.       // Check if found
44.       if (hookSet.isEmpty()) {
45.          ......
46.          return;
47.       }
48.       ......
49.    } catch (Throwable ex) {
50.       ......
51.    }
52. }
```

第 3 行检测当前运行设备的 SDK 版本，若低于 hook 函数中声明的 SDK 版本则直接返回。

7-25 行实例化 XC_MethodHook 类，实现 beforeHookedMethod 和 afterHookedMehtod 方法，这两个函数分别在被 hook 的函数调用之前和之后调用。在 beforeHookedMethod 中首先检查当前进程的 uid，若<=0 则返回，否则调用参数 hook 的 before 函数；在 afterHookedMethod 中，首先检查是在 after 函数调用之前是否有异常抛出，然后对当前进程的 uid 进行检查，最后调用参数 hook 的 after 函数。参数的 hook 的 before 和 after 函数定义会在后续对 XSmsManager 分析中继续介绍。27-31 行检查需要 hook 的函数所在的类是否存在，若不存在则返回。33-42 行将需要 hook 的函数以及对应的处理函数进行注册，也就是调用 XposedBridge.hookMethod 将其添加到 XposedBridge. hookedMethodCallbacks 哈希表中，由 XposedBridge 统一负责管理 hook 函数及其对应的处理函数。在进行注册之前，会对需要 hook 的函数做一些检查工作，包括：

- 是否是构造函数；
- hook 函数所在的类中是否确实存在此方法；
- 是否是 public；

44-47 行检查是否成功注册了需 hook 的函数，如果 hookSet 为空，则表明在注册未通过。

上述分析了 hook 函数的注册过程，以 SmsManager 类为例，继续分析其 before 和 after 函数执行了哪些操作。before 函数的代码如下：

```
1.  protected void before(MethodHookParam param) throws Throwable {
2.      if (mMethod == Methods.sendDataMessage || mMethod == Methods.sendMultipartTextMessage || mMethod
    == Methods.sendTextMessage)
3.          if (isRestricted(param))
4.              param.setResult(null);
5.  }
```

第 2 行首先判断函数名，在 before 函数中，getAllMessagesFromIcc 没有进行相关处理。第 3 行调用 isRetricted(MethodHookParam) 判断函数是否被设置为需要 hook。isRetricted(MethodHookParam) 接着会调用 isRetricted(MethodHookParam， String)，isRetricted(MethodHookParam，String)函数代码如下：

```
1.  protected boolean isRestricted(MethodHookParam param, String methodName) throws Throwable {
2.      int uid = Binder.getCallingUid();
3.      return PrivacyManager.getRestricted(this, null, uid, mRestrictionName, methodName, true, true);
4.  }
```

第 2 行获取调用者的 uid，然后调用 PrivacyManager 类的 getRestricted(final XHook hook, Context context, int uid, String restrictionName,String methodName, boolean usage, boolean useCache)函数，其核心代码如下，主要分析下是如何去判断是否需要进行 hook 的逻辑，省略了不少代码。

```
1.  // Check if restricted
```

```
2.  boolean fallback = true;
3.  boolean restricted = false;
4.  if (context != null)
5.        try{
6.           // Get content resolver
7.           ContentResolver contentResolver = context.getContentResolver();
8.           ......
9.           // Query restriction
10.          Cursor cursor = contentResolver.query(PrivacyProvider.URI_RESTRICTION, null,
    restrictionName, new String[] { Integer.toString(uid), Boolean.toString(usage), methodName },
    null);
11.          ......
12.          try {
13.                // Get restriction
14.                if (cursor.moveToNext()) {
15.                      restricted = Boolean.parseBoolean(cursor.getString(cursor
16.                            .getColumnIndex(PrivacyProvider.COL_RESTRICTED)));
17.                      fallback = false;
18.                   } else {
19.                      ......
20.                   }
21.                } finally {
22.                   cursor.close();
23.                }
24.             // Send usage data async
25.             sendUsageData(hook, context);
26.          }
27.       ......
28.    // Add to cache
29.    synchronized (mRestrictionCache) {
```

```
30.       if (mRestrictionCache.containsKey(keyCache))
31.          mRestrictionCache.remove(keyCache);
32.          mRestrictionCache.put(keyCache, new CRestriction(restricted));
33.    }
34.    ......
35.    return restricted;
```

第 7-10 行获取 ContentResolver，并对 Uri 为 content://biz.bokhorst.xprivacy.provider/restriction 进行查询操作。该 contentResolver.query(...)会调用 PrivacyProvider 函数的 query 方法。PrivacyProvider 是一个 Content Provider 组件，需要实现 query 函数。后续的分析依赖于 XPrivacy 应用下的 shared_prefs 目录下的文件格式，因此先对 PrivacyProvider 进行分析(有点乱！)。PrivacyProvider 的 onCreate 函数代码如下：

```
1.  public boolean onCreate() {
2.     try {
3.        writeMetaData();
4.        convertRestrictions();
5.        convertSettings();
6.        fixFilePermissions();
7.     } catch (Throwable ex) {
8.        Util.bug(null, ex);
9.     }
10.    return true;
11. }
```

writeMetaData 函数将 assets/meta.xml 文件写入到 /data/data/biz.bokhorst.xprivacy/meta.xml 文件中并设置为全局可读。meta.xml 文件存储 restriction 值，可进行 hook 的函数，被 hook 的函数需要的权限以及 SDK 版本号，基本格式如下：

```
1.  <Meta>
2.      <Hook restriction="accounts" method="addOnAccountsUpdatedListener"
    permissions="GET_ACCOUNTS" sdk="14" />
```

```
3.      <Hook restriction="accounts" method="blockingGetAuthToken" permissions="USE_CREDENTIALS"
    sdk="14" />
4.      ......
5.      <Hook restriction="browser" method="BrowserProvider"
    permissions="READ_HISTORY_BOOKMARKS,GLOBAL_SEARCH" sdk="14" />
6.      ......
7.  </Meta>
```

convertRestrictions 函数解析 shared_perfs/biz.bokhorst.xprivacy.provider.xml 文件，将解析结果重新存储到 shared_perfs/biz.bokhorst.xprivacy.provider.<uid>.xml 文件中。根据代码处理逻辑，shared_perfs/biz.bokhorst.xprivacy.provider.xml 文件中应该是存储了每个应用程序被限制的类别以及方法，也就是针对每个应用程序，XPrivacy 应该需要 hook 的函数。不过在安装了 XPrivacy 重启后，shared_perfs 目录下并没有 biz.bokhorst.xprivacy.provider.xml 文件，所以以下代码不会执行。

XPrivacy 提供对每个应用进行细粒度的访问控制的设置，在主界面上点击应用程序图标，进入设置界面，可以选择对哪些类别的访问进行控制，同时在每个类别下还可以选择对哪些方法的调用进行控制，如图 3 所示，设置完毕后，查看 shared_perfs 目录会发现增加了一个 biz.bokhorst.xprivacy.provider.10051.xml 文件，10051 为 Google+应用的 uid。根据我们的设置，

```
1.  if (source.exists() && !backup.exists()) {
2.    ......
3.     SharedPreferences prefs = getContext().getSharedPreferences(PREF_RESTRICTION,
    Context.MODE_WORLD_READABLE);
4.     for (String key : prefs.getAll().keySet()) {
5.        String[] component = key.split("\\.");
6.        if (key.startsWith(COL_RESTRICTED)) {
7.           String restrictionName = component[1];
8.           String value = prefs.getString(key, null);
9.           List<String> listRestriction = new ArrayList<String>(Arrays.asList(value.split(",")));
10.          listRestriction.remove(0);
11.          for (String uid : listRestriction)
12.             updateRestriction(Integer.parseInt(uid), restrictionName, null, false);
13.       } else if (key.startsWith(COL_METHOD)) {
14.          int uid = Integer.parseInt(component[1]);
```

```
15.          String restrictionName = component[2];
16.          String methodName = component[3];
17.          boolean value = prefs.getBoolean(key, false);
18.          updateRestriction(uid, restrictionName, methodName, value);
19.          } else
20.             ......
21. }
```

会生成如图 3 所示的内容。biz.bokhorst.xprivacy.provider.<uid>.xml 文件存储了针对每个应用程序需要进行 hook 的处理，name 值有两类，分别是 Restricated.<category> 和 Method.<category>.<method>。其中 Restricated.<category>表示对 category 下的方法进行 hook，category 表示所属的类别，也就是图 3 中显示的 Accounts, Browser 等等；Method.<category>.<method>表示不需要 hook 的方法，method 是不需要进行 hook 处理的函数名(以上规则基于 value=true 的情况)。图 4 表示在 calling 这个类别下，除 sendMultipartTextMessage 以及 sendDataMessage 之外的函数都要进行 hook。虽然没有 biz.bokhorst.xprivacy.provider.xml 文件，但是仍然可以根据 biz.bokhorst.xprivacy.provider.10051.xml 文件的内容对上述代码进行分析，主要的思想是调用 updateRestriction 函数根据 uid 参数将 hook 规则写入到 biz.bokhorst.xprivacy.provider.<uid>.xml 文件中。

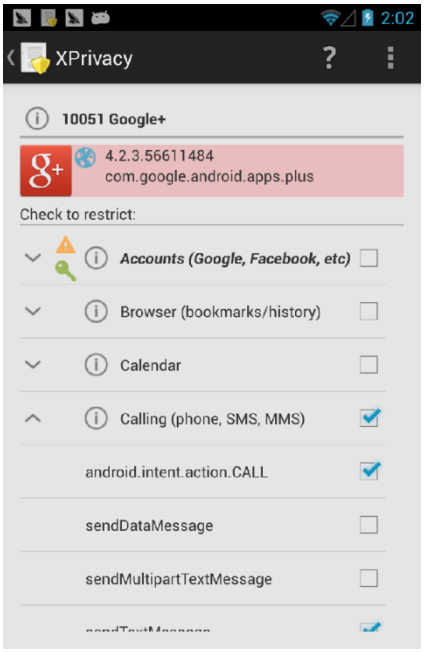

图 3 针对 Google+进行的访问控制设置

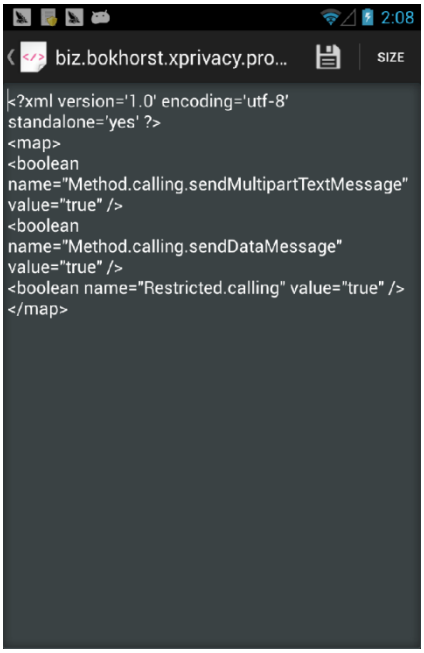

图 4 biz.bokhorst.xprivacy.provider.10051.xml

继续回到 PrivacyProvider 的 onCreate 函数中，第 5 行调用 convertSettings 函数，该函数的代码如下：

```
1.  private void convertSettings() throws IOException {
```

```
2.     SharedPreferences prefs = getContext().getSharedPreferences(PREF_SETTINGS,
    Context.MODE_WORLD_READABLE);
3.     SharedPreferences.Editor editor = prefs.edit();
4.        for (String key : prefs.getAll().keySet())
5.           try {
6.              String value = prefs.getString(key, null);
7.              if (PrivacyManager.cValueRandomLegacy.equals(value))
8.                 editor.putString(key, PrivacyManager.cValueRandom);
9.           } catch (Throwable ex) {
10.          }
11.       editor.apply();
12.       setPrefFileReadable(PREF_SETTINGS);
13.    }
```

convertSettings 函数主要功能是读取 shared_prefs/biz.bokhorst.xprivacy.provider.settings.xml 文件，将其中"\nRandom\n"字符串替换为"#Random#"。在 XPrivacy 主界面点击 Settings 可以设置需要伪造的字段，如图 5 所示，点击 OK 后信息会保存到 biz.bokhorst.xprivacy.provider.settings.xml 文件中。这些信息提供给 XPrivacy，用于返回给其他应用程序的伪造值。

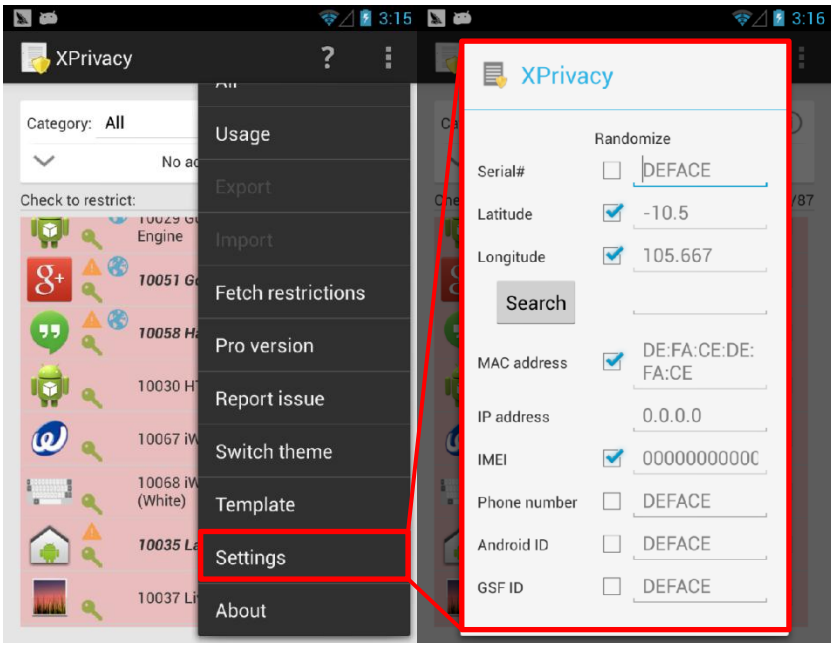

图 5　XPrivacy Settings

onCreate 函数第 6 行调用 fixFilePermissions 将 shared_prefs 目录下以

biz.bokhorst.xprivacy.provider 开头的 xml 格式文件访问权限设置为全局可读，但是*.usage.*.xml 文件除外，关键代码如下。

```
1.  if (file.getName().startsWith("biz.bokhorst.xprivacy.provider.") &&
    file.getName().endsWith(".xml") && !file.getName().contains(".usage."))
2.      file.setReadable(true, false);
```

.usage..xml 文件存储了应用程序调用特定 API 的时间以及具体函数名，基本格式如图 6 所示。在 XPrivacy 主界面点击 Usage，显示的内容即为从这些 usage 文件中解析的，如图 7 所示。

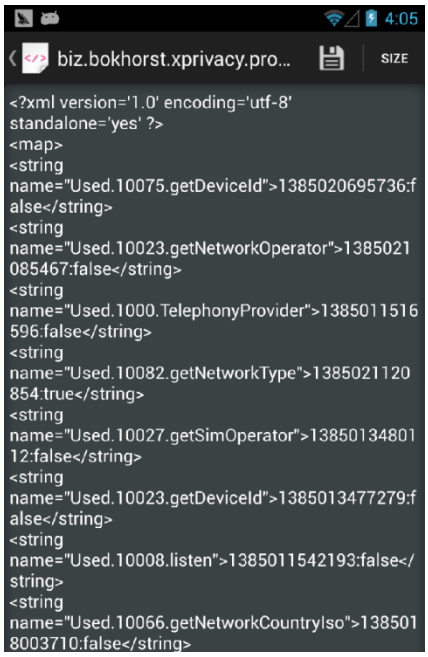

图 6　*.usage.phone.xml 基本内容

图 7　XPrivacy Usage

OK，继续分析 PrivacyProvider 的 query 函数。其代码如下：

```
1.  public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String
    sortOrder) {
2.      if (sUriMatcher.match(uri) == TYPE_RESTRICTION && selectionArgs != null &&
    selectionArgs.length >= 2) {
3.          // Get arguments
4.          final String restrictionName = selection;
5.          final int uid = Integer.parseInt(selectionArgs[0]);
6.          boolean usage = Boolean.parseBoolean(selectionArgs[1]);
```

```
7.            final String methodName = (selectionArgs.length >= 3 ? selectionArgs[2] : null);
8.            return queryRestrictions(uid, restrictionName, methodName, usage);
9.        }
10.       ......
11.       throw new IllegalArgumentException(uri.toString());
12.   }
```

第 4 行获取 restrictionName，对于 XSmsManager 中可 hook 的函数，其值为"calling"(getAllMessagesFromIcc 的 restrictionName 值为"messages")。第 5 行获取调用者的 uid。第 6 行获取 usage 值，此处为 true。第 7 行获取 hook 函数的名称。第 8 行调用 queryRestirctions 函数，其代码如下：

```
1.  private Cursor queryRestrictions(final int uid, final String restrictionName, final String
    methodName, boolean usage) {
2.     MatrixCursor cursor = new MatrixCursor(new String[] { COL_UID, COL_RESTRICTION, COL_METHOD,
    COL_RESTRICTED });
3.     // Build restriction list
4.     ......
5.     listRestrictionName = new ArrayList<String>();
6.     listRestrictionName.add(restrictionName);
7.     ......
8.     SharedPreferences prefs = getContext().getSharedPreferences(PREF_RESTRICTION + "." + uid,
9.             Context.MODE_WORLD_READABLE);
10.    // Process restrictions
11.    boolean restricted = false;
12.    for (String eRestrictionName : listRestrictionName) {
13.       boolean eRestricted = getRestricted(eRestrictionName, methodName, prefs);
14.       cursor.addRow(new Object[] { uid, eRestrictionName, methodName,
    Boolean.toString(eRestricted) });
15.       restricted = restricted || eRestricted;
16.    }
17.    // Update usage data
```

```
18.     if (usage && restrictionName != null && methodName != null) {
19.         final boolean isRestricted = restricted;
20.         mExecutor.execute(new Runnable() {
21.             public void run() {
22.                 long timeStamp = new Date().getTime();
23.                 updateUsage(uid, restrictionName, methodName, isRestricted, timeStamp);
24.             }
25.         });
26.     }
27. }
```

第 8 行获取 SharedPreferences 实例，其对应的文件路径为/data/data/biz.bokhorst.xprivacy/shared_perfs/biz.bokhorst.xprivacy.provider.<uid>.xml。该文件存储了对于每个应用程序，需要进行 hook 的函数，格式如上述所分析的 10051.xml。13 行调用 getRestricted(String restrictionName, String methodName, SharedPreferences prefs)，该函数代码如下：

```
1.  private static boolean getRestricted(String restrictionName, String methodName, SharedPreferences
    prefs) {
2.         // Check for restriction
3.         boolean restricted = prefs.getBoolean(getRestrictionPref(restrictionName), false);
4.         // Check for exception
5.         if (restricted && methodName != null)
6.             if (prefs.getBoolean(getExceptionPref(restrictionName, methodName), false))
7.                 restricted = false;
8.         return restricted;
9.     }
```

第 3 行首先调用 getRestrictionPref 函数，进行字符串格式化，返回类似"Restricted.<restrictionName>"的字符串，对于 SmsManager 类，返回"Restricted.calling"。接着调用 SharedPreferences 的 getBoolean 方法获取 boolean 值。第 6 行主要是判断是否有例外的情况，也就是上述分析时说的在某个 category 下排除在 hook 函数之外的情况，最后返回 restricted 值。

回到 queryRestrictions 函数中，18-24 行更新被 hook 函数的使用情况，也就是需要更新

前述的*.usage.*.xml 文件。

回到 PrivacyManager 类的 getRestricted(final XHook hook, Context context, int uid, String restrictionName,String methodName, boolean usage, boolean useCache)函数中，29-33 行将需要 hook 函数的信息存入到 cache 中，并返回 restricted 值。

回到 SmsManager 类的 before 函数中，根据前述分析，若设置了 hook sendTextMessage 等函数，则第 3 行返回 true，接着执行第 4 行，将执行结果设置为 null。

SmsManager 类中的 after 函数针对 getAllMessagesFromIcc，将执行结果设置为空的 ArrayList，这样调用该 getAllMessagesFromIcc 的程序就无法得到真正的结果了。

上述是对 SmsManager 类的 hook 操作，其他类的操作流程基本类似，不一样的地方是 before 和 after 函数的处理，有兴趣的可以继续分析。XPrivacy 类中的 handleLoadPackage 会在类加载过程进行 hook 的处理，基本思路应该和 initZygote 的处理基本相同。

XPrivacy 中对需 hook 的函数的注册过程如图 8 所示；XSmsManager 中 before 函数执行过程如图 9 所示。

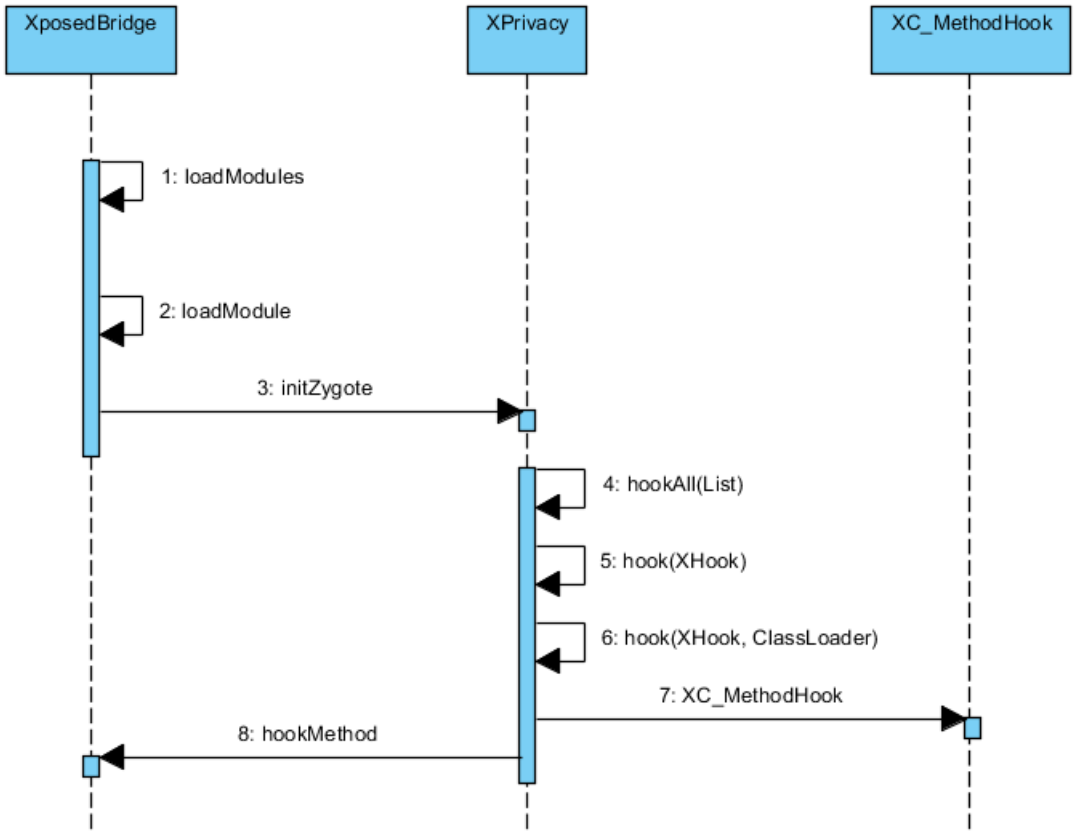

图 8 initZygote 过程 hook 函数注册流程

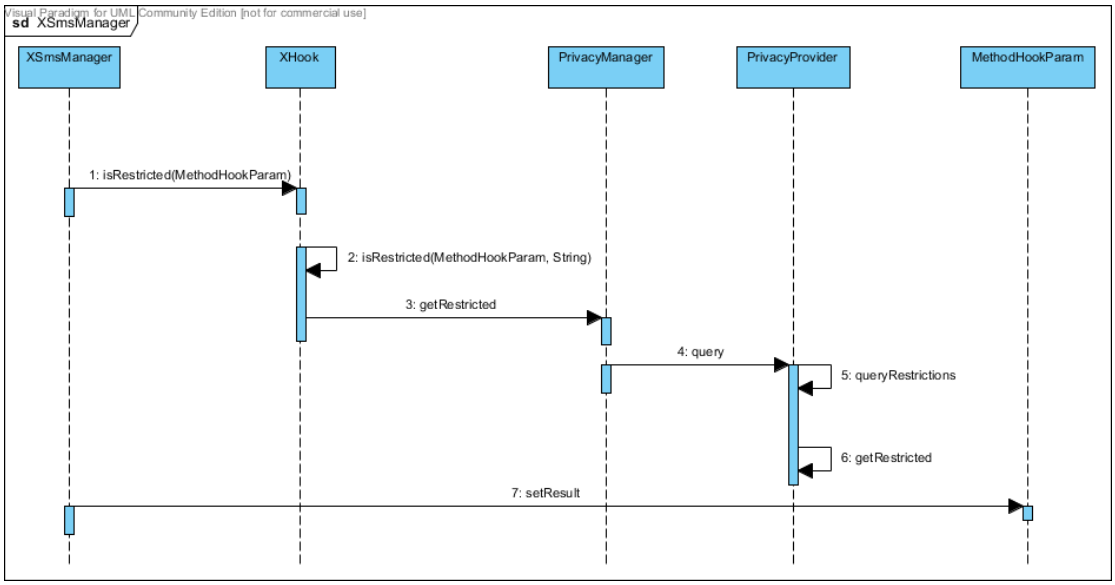

图 9　XSmsManager.before 执行流程

附.PrivacyManager 中 static 块的分析

PrivacyManager 类中 122-143 行声明了一个 static 块，Java 中 static 块中的代码会在类初始化过程中执行，分析下这部分代码执行的操作，代码如下：

```
1.  static {
2.      // Scan meta data
3.      try {
4.          String packageName = PrivacyManager.class.getPackage().getName();
5.          File in = new File(Environment.getDataDirectory() + File.separator + "data" + File.separator
    + packageName + File.separator + "meta.xml");
6.          Util.log(null, Log.INFO, "Reading meta=" + in.getAbsolutePath());
7.          FileInputStream fis = null;
8.          try {
9.              fis = new FileInputStream(in);
10.             XMLReader xmlReader = SAXParserFactory.newInstance().newSAXParser().getXMLReader();
11.             MetaHandler metaHandler = new MetaHandler();
12.             xmlReader.setContentHandler(metaHandler);
13.             xmlReader.parse(new InputSource(fis));
14.         }
15.        ......
```

```
16. }
```

这部分代码的主要作用是解析/data/data/biz.bokhorst.xprivacy/meta.xml 文件中的内容，存入到 PrivacyManager 的类型为 LinkedHashMap 静态成员变量 mMethod 中。第 5 行实例化 File 对象，文件路径为/data/data/biz.bokhorst.xprivacy/meta.xml。10-13 行使用 XMLReader 解析 xml 文件，处理 XML 文件的类位 MetaHandler。MetaHandler 类代码如下：

```
1.  private static class MetaHandler extends DefaultHandler {
2.     @Override
3.     public void startElement(String uri, String localName, String qName, Attributes attributes)
    throws SAXException {
4.     if (qName.equals("Meta"))
5.        ;
6.     else if (qName.equals("Hook")) {
7.        // Get meta data
8.        String restrictionName = attributes.getValue("restriction");
9.        String methodName = attributes.getValue("method");
10.       String dangerous = attributes.getValue("dangerous");
11.       String permissions = attributes.getValue("permissions");
12.       int sdk = Integer.parseInt(attributes.getValue("sdk"));
13.       // Add meta data
14.       if (Build.VERSION.SDK_INT >= sdk) {
15.          boolean danger = (dangerous == null ? false : Boolean.parseBoolean(dangerous));
16.          String[] permission = (permissions == null ? null : permissions.split(","));
17.          MethodDescription md = new MethodDescription(methodName, danger, permission, sdk);
18.          if (!mMethod.containsKey(restrictionName))
19.             mMethod.put(restrictionName, new ArrayList<MethodDescription>());
20.         if (!mMethod.get(restrictionName).contains(methodName))
21.             mMethod.get(restrictionName).add(md);
22.          }
23.       }
24.       ......
```

```
25. }
```

上述代码中的 startElement 函数获取 restiction、method、dangerous、permissions 以及 sdk 的值，根据这些信息实例化 MethodDescription 对象，并最终存入到 mMethod 中。

/data/data/biz.bokhorst.xprivacy/meta.xml 文件又是如何生成的呢？XPrivacy 定义了一个 Content Provider，即 PrivacyProvider 类，在该类的 onCreate 函数中，会调用 writeMetaData 函数将 assets/meta.xml 文件写入到/data/data/biz.bokhorst.xprivacy/meta.xml 路径下。