WHITE PAPER

McAfee™
Together is power.

Secure Coding for Android Applications

Table of Contents

Author

This white paper was written by:
Naveen Rudrapp, Senior Security Consultant, McAfee® Foundstone® Professional Services

Secure Coding for Android Applications

Smartphones are more popular than ever. One of the reasons for this is the fact that the Google Android operating system (OS) is a platform that enables developers to write applications and distribute them for free in an open market. According to the latest studies, more than one billion Android devices have been activated with an astonishing growth of 1.4 million devices per day. With this sort of growth, it is absolutely necessary for developers to understand how to create secure Android applications. This white paper focuses on secure coding practices for Android applications.

Android Application Components

Before diving deeply into Android's security options, it is necessary to take a high-level overview of the major components of an Android application to understand how the different pieces fit together. Please refer to Android developer documentation for extensive information about various components discussed below.

Activity

In Android development terms, an "Activity" refers to single, focused window that interacts with a user and provides functionality. An activity forms the fundamental building blocks of the application.

An activity has the following states:

- **Active:** When an activity is interacting with a user, it is at the top of the activity stack and visible to the user. Android will kill any other services or activities on the stack to keep the active activity alive.
- **Paused:** This is a state where an activity is not in focus but is actually visible to user. For example, this state is reached when a pop-up appears when activity is running.
- **Stopped:** This is the state where activity is not visible to user, but resides in memory and retaining all data. This activity will be killed to save memory if needed for an active activity.
- **Inactive:** An activity just before launching, after it has been killed, is said to be in an inactive state.

Knowing about the states of an activity allows a developer to understand how data is handled and aids in implementing activities securely.

Connect With Us

Intents

An intent is commonly used to start an activity or service. Intents can be broadcast and received within the application itself and with other applications. This allows for great flexibility in application development, information sharing, and the ability to trigger operations in other applications.

There are two main types of intents:

- **Explicit intents:** Explicit intents specify the exact class that needs to be invoked to launch an activity within the application. These are limited to the application context in which they are run.
- **Implicit intents:** These are the intents that hold information about the type of operation to be performed. It's up to the OS to decide the best operation based on the information provided.

Service

A service represents a background operation or an operation that does not require user interaction and takes a lengthy amount of time to complete. These operations are performed without affecting the main application running on the front end. Service continues to run in background, even when application is not running.

Content Providers

Content providers store data persistently. They manage the storage of application data and interact with a number of local SQL databases. Content providers also provide the best means to share data between applications.

WebView

WebViews act like a web browser to display HTML content to the user. Android's WebKit engine is used to display web pages. Any vulnerability found in WebKit directly impacts the WebView. This component allows a user to navigate forward and backward through the history, zoom in and out, and perform text searches, just like Internet Explorer, Firefox, or any other browser.

Permissions

The core security of an application is defined by its permissions. The extent to which an application can perform an action is limited to the permissions defined in its `AndroidManifest.xml`. By default, every application is sandboxed by the OS and restricts access to the data of another application. At the time of application installation, the user is presented with the list of permissions that are required by the application. Once the user grants those permissions, only then the application will be installed. Granting of permissions dynamically at runtime is not supported.

Secure Coding Recommendations

So far, we have covered the various Android components used in an application. In this section, we will cover secure coding recommendations and the associated client-side application attacks.

Lock-down application permissions

It is necessary to follow the principle of least privilege when assigning permissions. Permissions should not be assigned unless they are required. The application should be granted only the minimum required permissions at the architecture level. For instance, READWRITE permissions should not be granted when only READ permissions are required. This is a common mistake made by developers due to a lack of understanding of the functionality at the application. Examples like these underscore the importance of strong application development planning and requirements documentation.

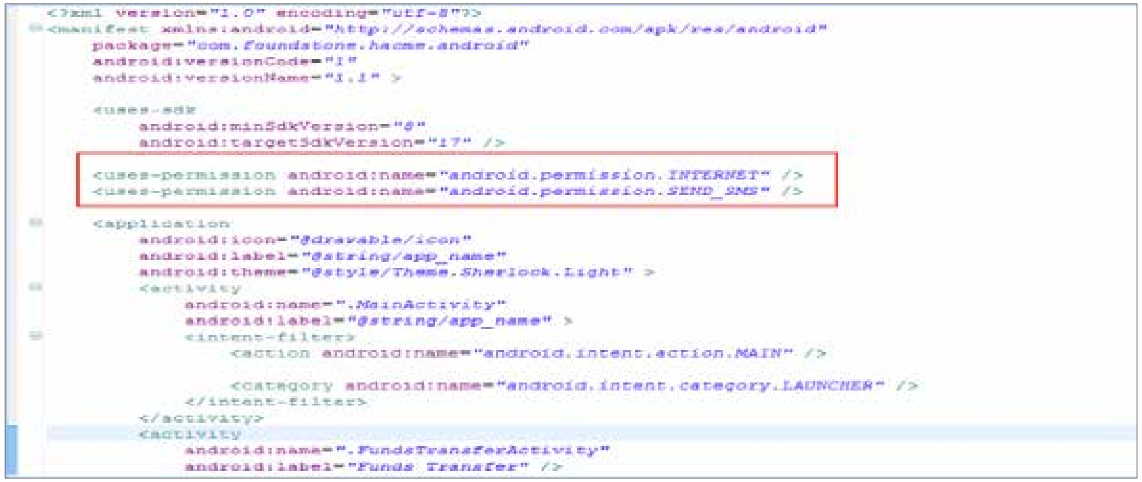
For example, the `Android:protectionLevel` element of the `AndroidManifest.xml` file defines the protection/risk level associated with the installed application. It also provides the procedure the OS should follow to determine whether the permission can be granted. When the value of the parameter is `dangerous`, the application, when installed, gains permission to access user data and to control the device. Developers should exercise extreme caution while assigning applications with high-risk permissions.

File permissions

File permissions apply to files stored on external storage. Any file created using `openFileOutput` is private to the application and cannot be accessed by other applications. Pay close attention before providing a file with the `MODE _ WORLD _ READABLE or MODE _ WORLD _ WRITABLE` permissions. This allows other applications to access the file. Do not provide the writable option until it is required to enforce the principle of least privilege. The standard way to share a file between applications is to use `Content Provider`.

Permissions can be defined at two levels: in the `AndroidManifest.xml` file and in program's code. Both cases need to be analyzed for security issues.

1. Permission defined in `AndroidManifest.xml` are shown in Figure 1.



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.foundstone.hacme.android"
    android:versionCode="1"
    android:versionName="1.1" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.SEND_SMS" />

    <application
        android:icon="@drawable/icon"
        android:label="@string/app_name"
        android:theme="@style/Theme.Sherlock.Light" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".FundsTransferActivity"
            android:label="Funds Transfer" />
```

Figure 1. Image shows permissions defined in AndroidManifest.XML file.

2. In code, permission defined for a file is as below:

```
String OUTPUT_FILE = "file.txt";
FileOutputStream fos = openFileOutput(OUTPUT_
FILE,
Context.MODE_WORLD_WRITEABLE);
```

Custom permissions

Another important consideration is the use of custom permissions. These permissions are described to the user via a permission description within the `Android:description` tag of AndroidManifest.xml. Care needs to be taken to ensure the description is adequate to provide the layperson user with information that details what permission is being granted.

Below is a code sample showing the description defined for custom permission used:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:Android="http://schemas.Android.com/apk/res/Android"
    package="com.testpackage.permissiontestclient"
    Android:versionCode="1"
    Android:versionName="1.0" >

    <uses-sdk Android:minSdkVersion="10" />
    <permission Android:name="com.testpaccourierkage.mypermission"
Android:label="my_permission" Android:protectionLevel="dangerous"
Android:description="@string/detonate_description"></permission>
<application>
   <activity
            Android:permission="com.testpackage.mypermission"
            Android:name=".PermissionTestClientActivity"
            Android:label="@string/app_name" >
  </activity>
</application>
</manifest>
```

Handle Broadcast Messages carefully

To handle event-driven tasks, an application can register a broadcast receiver that executes a function once it's passed an intent that matches specified criteria. Applications can send broadcast intents, which allow another application to receive it and process its data.

There are two ways to register a broadcast receiver: within the application code and within the AndroidManifest.xml:

Registering a broadcast receiver within the application code:

```
registerReceiver(new BroadcastReceiver(){

@Override
public void onReceive(Context context, Intent
intent) {…}
}, null);
```

Registering a broadcast receiver within the AndroidManifest.xml:

```
<receiver Android:name="receiver" >
<intent-filter>
<action Android:name="com.Myapplciation.
Android.mybroadcast" />
</intent-filter>
</receiver>
```

Once the class that receives broadcast intents has been identified, it must be thoroughly reviewed to ensure sound development practices.

Null values passed to broadcast receivers

One common area of concern involves variables containing null values sent within a broadcast intent. Application developers commonly forget to consider this possibility which may result in a denial-of-service (DoS) condition.

```
public class MyBroadcastReceiver extends BroadcastReceiver {
@Override
 public void onReceive(Context context, Intent intent) {
 //Code implemented here
 //check if intent and variables value in it is null before using it
 //and proper security exceptions are in place to handle it
 //without crashing the application
  }
 }
```

It is important to check intent and to make sure that variables in intent are for `null` in both of the cases discussed above before performing any operation to ensure `null` protection is in place. Any object or data received from the broadcast should be checked for invalid data or exceptions before using it in an application.

Broadcast Messages for Inter-Process Communication (IPC)

The nature of broadcast messages permits any application to receive a broadcasted Intent. If broadcast messages are used for IPC, then a malicious application may be able to gain access to another application's data. This is often a common occurrence that demonstrates the lack of understanding around broadcast messages.

The following functions send broadcast messages and are vulnerable to IPC sniffing:

```
sendBroadcast(intent);
sendStickyBroadcast(intent);
```

To mitigate this issue, use intents with assigned permissions. This can be achieved in two ways, shown below. A malicious application cannot receive an intent if it is not granted the required permissions.

Permission defined in code:

```
String requiredPermission =
"com.Foundstone.MY _ BROADCAST _
PERMISSION _ Defined";
sendBroadcast(intent,
requiredPermission);
sendOrderedBroadcast(intent,
requiredPermission);
```

Permission defined in AndroidManifest.xml file:

```
<receiver
Android:name="com.Foundstone.
NameForOrderedReceiver"
Android:permission="com.Foundstone.MY_
BROADCAST_PERMISSION_Defined">
  <intent-filter>
  <action Android:name="com.Foundstone.
  action.ACTION_CLASS_FOR_ORDERED_BROADCAST"
  />
  </intent-filter>
</receiver>
```

Using this method, an intent will be delivered to those registered receivers who are granted the required permission.

Use the local broadcast manager if the intent needs to be broadcast locally. The local broadcast manager allows intents to be broadcast locally within the application so no other application can gain unauthorized access to application data.

The code below shows how a local broadcast is sent:

```
LocalBroadcastManager lbm =
LocalBroadcastManager.getInstance(this);
lbm.sendBroadcast(new Intent(ANY _
ACTION _ TO _ BE _ PERFORMED));
```

Insecure storage

Application data should be stored in two folders:

- `/data/data/<package name of application>`
- `/sdcard`

Data stored in both of these directories behave differently from a security standpoint. Data that is stored in `/data/data/<package name of application>` cannot be accessed by another application unless the application explicitly provides permissions or if the Android device is rooted. Data that is stored in `/sdcard` can be accessed by any application without the need for any special permission or rooting. Hence, it is common for malicious applications to access data in `/sdcard`.

When evaluating an application's storage usage, ensure that both online and offline functionality of the application is evaluated. Look specifically for code that allows storage of the data locally and ensure that no sensitive data is stored on the client side. At the architecture level, try to minimize the sensitive data that needs to be stored on the device. If any data needs to be stored, then encrypt it using a strong algorithm prior to storage.

WebView Caching

WebView class allows HTML data gets cached locally. If sensitive information is requested by WebView, then consider using `clearCache()`to delete any sensitive data stored locally. Also use "`no-cache`" to instruct the Android WebKit not to cache data.

`PreferenceActivity` storage

`PreferenceActivity` allows a user to store data locally that can be read by all the classes in the application. When this activity is used, an XML file is created in `/data/data/<Package name of application>`. This data persists even after the application is closed. If the mobile device is stolen, then data stored within the XML file can be retrieved after rooting the device.

Insecure storage in process memory

Data processed by the application may be stored within memory longer than necessary, which makes it more susceptible to attack. An attacker with access to the phone may be able to dump the memory of the process to gain access to sensitive information such as usernames, passwords, and other data.

Analyze the classes that take username, password, and account number as input. Try to determine if the values are cleared in the memory after use. If not, the application may expose sensitive information if a memory dump can be obtained by an attacker.

The `Application` class is another important class to be re-initialized with junk value once the application is closed. This class is shared across all the classes of same application and remains active even after application is closed. Hence, any data stored in this class can be obtained via dumping memory.

To manually validate that sensitive information is cleared from memory, log into the application, execute any operation that takes in sensitive data as input, and perform a memory dump to view if there is any sensitive present in it.

Steps to obtain a memory dump and analysis:

1. Obtain the memory dump of the target application using the `ddms.bat` tool after logging out of the application. This tool is present in the tools directory of the Android SDK.
2. Convert it to readable format using `hprof-conv.exe`. This tool is present in the tools directory of the Android SDK.

   Command: `hprof-conv.exe source dest`
3. Open the file obtained in the Eclipse memory analyzer.
4. Click on Open Dominator for entIre heap.
5. Click on "Group result by Group by package".
6. Navigate to the class that holds sensitive data, and see if the data is still available in clear text and if it can be retrieved.

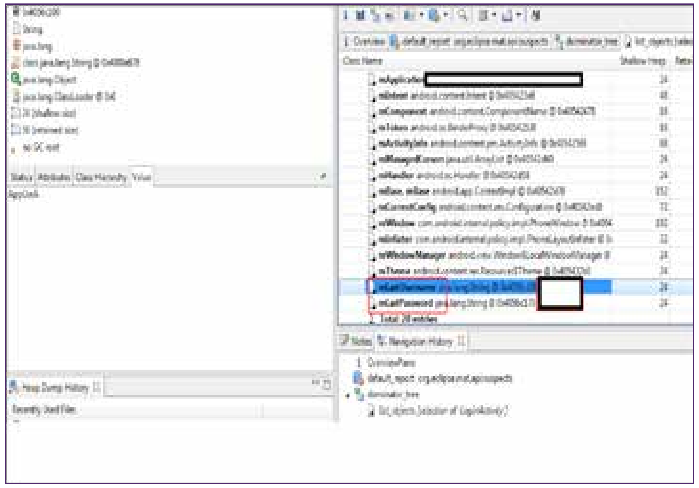


Figure 2. Username and password stored in variables and retrieved via memory dumping once user has logged out.

The Dalvik runtime allows garbage collection, but this does not allow a developer not to consider memory management. It is never advisable for any variable to hold sensitive information in it even when the user is logged in. This is especially true when the user logs off the application. At that point, all variables holding sensitive information should be cleared by initializing them to some junk value.

Protect pending intents

The pending intents function allows the intent in your application to be invoked by another application. Just invoking the intent is not the issue. The issue is that the application that invokes the intent also executes at the same permission level as that of the application that had the pending intent to be invoked.

By allowing other applications to invoke `PendingIntent` in our application, we are allowing the other applications to execute at the same privilege as that of our application. Hence, we should be careful about trust and operation performed by third-party applications using `PendingIntent`.

Look for the code snippets as shown below while checking for secure usage of pending intent—you can identify that intent is being handed over to the `alarmManager` application:

```
Intent myIntent = new
Intent(AndroidAlarmService.this,
MyAlarmService.class);
pendingIntent = PendingIntent.
getService(AndroidAlarmService.this, 0,
myIntent, 0);
AlarmManager alarmManager =
(AlarmManager)getSystemService(ALARM_
SERVICE);
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.
currentTimeMillis());
calendar.add(Calendar.SECOND, 10);
alarmManager.set(AlarmManager.RTC_WAKEUP,
calendar.getTimeInMillis(), pendingIntent);
```

Once identified, check to see if the intent is handed over to a third-party application. Now try to figure out what the third-party application does with the pending intent (that is, `alaramManager`). Here it is necessary to understand that the trust is based on the third-party application to determine if any security issues exist.

`PendingIntents` can be created using three different functions; therefore, four types of code need to be analyzed:

- `getActivity(Context, int, Intent, int)`
- `getActivities(Context, int, Intent[], int)`
- `getBroadcast(Context, int, Intent, int)`
- `getService(Context, int, Intent, int)`

The best and simplest approach is to find an alternative for the pending intent which is as good as eliminating risk. If a pending intent is an application requirement, make sure that only a trusted application receives it. This leaves no room for that intent to be used by an untrusted application.

Improper usage of WebView

The `WebView` class is one of the most powerful classes, and it renders web pages inside a normal browser. It also allows applications to interact with WebView by adding a hook, monitoring changes being made, add JavaScript, and more. Even though this seems like a great feature, it brings in security loopholes if not used with caution. Since `WebView` can be customized, it creates the opportunity to break out of the sandbox and bypass the same origin policy.

WebView allows sandbox bypass in two different scenarios:

1. JavaScript can invoke Java code.
2. Java code can invoke JavaScript.

Sample code to Invoke Java from JavaScript:

```
wv.addJavascriptInterface(new FileUtils(), "file");
<script>
filename = '/data/data/com.Foudnstone/data.txt';
file.write(filename, data, false);
</script>
```

Sample code to invoke JavaScript from Java:

```
String javascr = "javascript: var newscript=document.
createElement(\"script\");";
javascr += "newscript.src=\"http://www.foundstone.com\";";
javascr += "document.body.appendChild(newscript);";
myWebView.loadUrl(javascr);
```

Now consider a scenario where there was link in the `WebView` that redirected a user to a malicious website or to load malicious JavaScript from another website. Attacker-controlled code can use legitimate sandbox bypass code similar to the examples above and access all application data.

For the secure usage of the WebView, look for the action that is being performed while the loadurl event is triggered. There are two scenarios where the security issue can be found.

1. Loadurl event is not overridden.
2. Loadurl event is overridden and the user is not restricted to the base URL of the application in Loadurl event.

To avoid security issues from the WebView, always restrict users to the application domain using the code shown below to prevents `WebView` security issues.

```
WebViewclient wvclient = New WebViewClient() {
// override the "shouldOverrideUrlLoading" hook.
public boolean shouldOverrideUrlLoading(WebView view,String url){
if(!url.startsWith("http://clientlocation.com")){
Intent i = new Intent("Android,intent.action.VIEW",Uri.parse(url));
startActivity(i);
// override the "onPageFinished" hook.
public void onPageFinished(WebView view, String url) { ...}
}
webView.setWebViewClient(wvclient);
// override the "onPageFinished" hook.
public void onPageFinished(WebView view, String url) { ...}
}
webView.setWebViewClient(wvclient);
```

Other best practices while coding with WebView are listed as below:

- Do not call setJavaScriptEnabled() for WebView until there is need for processing JavaScript. If JavaScript is not enabled, then attacks like XSS, defacing, and others will be eliminated.
- Compile the application against Android API level equal to or more than 17. This API forces the developer to add @JavascriptInterface to any method that will be exposed to JavaScript. This also prevents access to OS commands (via java.lang.Runtime).
- Send all traffic over SSL. Any traffic is easy to sniff and manipulate using a man-in-the-middle attack. A hacker cannot inject script via MITM and cannot break the sandbox of WebView.

Secure usage of service

Services perform long-term operations that run in the background—such as continuously monitoring to determine whether Internet connectivity is there or not. When a service is declared in `AndroidManifext.xml`, it cannot be invoked by other applications because it is not exported (this is the default behavior). However, while declaring service—if intent filters are added—they are exported by default. Below is an example:

```
<service Android:name=".Search"
    Android:enabled="true"
    Android:label="@string/app_name">
    <intent-filter>
        <action Android:name="Foudnstone.
        intent.action.DO_SOME_ACTION" />
        <category Android:name="Android.
intent.category.DEFAULT" />
    </intent-filter>
</service>
```

Refer to the `AndroidManifest.xml` file to see if there have been any services defined with intent filters and if the code snippet is similar to the example provided above which is a security bug. Make sure proper permission is required to access the service. If no permission is required, then the service is not secure and is exported so that it can be used by other applications.

It is always safe to explicitly declare the `Android:exported` attribute so that we know how the service behaves. Along with this, define the `Android:permission` attribute as an extra measure so that any other application that needs to access the service should have the corresponding `<uses-permission>` declared in its manifest xml file.

Below is the snippet of code that shows how a service can be declared with all security attributes:

```
<service Android:enabled=["true" | "false"]
         Android:exported=["true" | "false"]
         Android:icon="drawable resource"
         Android:isolatedProcess=["true" | "false"]
         Android:label="string resource"
         Android:name="string"
         Android:permission="string"
         Android:process="string" >
    . . .
</service>
```

Content providers

This provides the mechanism that is used to share persistent data across applications. By default, `ContentProviders` can be invoked and accessed by any application if it is not set with permission. If permission is set on `ContentProvider` in AndroidManifest.xml file, then only those applications that are granted permission can access that content provider.

Always define both read and write permission as needed. This is shown in the declaration below:

```
<provider
              android:name="MyContactsProvider"
              android:authorities="com.permission.test"
              android:readPermission="android.permission.permRead"
              android:writePermission="android.permission.permWrite"/>
```

Improper use of implicit intents

There are scenarios where an activity needs to be started by Android OS at run-time and the developer does not explicitly specify in code the activity to be run. At run-time, Android OS will decide on the activity to be run based on the best match of intent filters. This is the scenario where implicit intents are used. In the code below, Android OS will display users with all those applications that can perform dial-up functionality. If there is just one application, then it is invoked. If there are multiple applications, then Android OS requests the user to choose which application he wants to invoke.

Intent intent = new Intent(Intent.ACTION_DIAL, Uri.parse("tel:666-2890"));

The security issue here is that the application will not be able to launch the required activity if it is not present in any of the applications installed. Sometimes, it will crash if proper exception management has not been implemented. Also, proper security measures need to be implemented so that the correct application is invoked and a malicious application has not installed on the Android device.

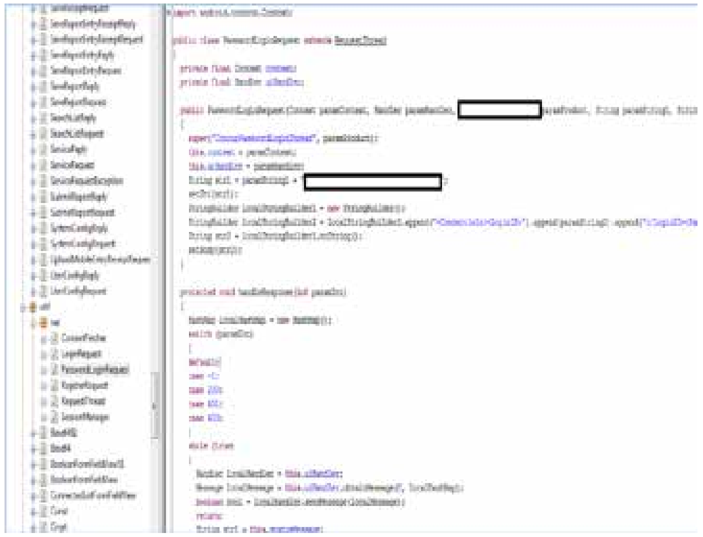
For secure usage of implicit intents, always check to see if the intent will resolve using the code snippet below, and install the proper application as desired instead of forcing the user to choose one from Google Play store.

```
if (something needs to be checked)
 {
  //create the implicit intent
  String url = "http://www.test.com";
  Intent i = new Intent(Intent.ACTION_VIEW);
  i.setData(Uri.parse(url));
  // Now check if there is an activity which can perform the action as
//desired by implicit intent
  PackageManager manager = getPackageManager();
  ComponentName comp = intent.resolveActivity(manager);
  if (comp == null) {

  // if there is no activity available to perform operation.
  // Downlaod required application from google play store and install
  // Upon succesfull installation start the activity
  }
  Else
{  // check for the safe known applications are present to launch
application
  //provide user with list of safe application for the action to be
performed
  startActivity(intent);
}
}
```

Code obfuscation

The Dalvik byte code can be easily reversed to obtain Java code that is very close to the original Java code. This aids the attacker in understanding application logic and also gain deeper understanding of the application. dex2jar and JD-Gui are two free tools that can be used to reverse engineer Android applications. The JD-Gui screenshot below shows a reverse-engineered application and its Java source.



[illegible]

Figure 3. None of the function names, variable names, or class names are obfuscated.

Code obfuscation is a method that involves mangling code during the build process. The generated code is difficult for humans to understand and increases the amount of work required for reverse engineering.

Progaurd is a free Java obfuscator. It renames fields, methods, and class with a name that is meaningless.

Excessive logging

Client-side data logging performed by Android applications has not garnered much attention from a security standpoint. However, during Android application review, we often see sensitive user data like user names, passwords, and account numbers written to application logs. This information can be easily retrieved by an attacker if he is able to gain access to the device.

Look for catch blocks that hold the key. Make sure logging is performed in all the catch blocks, but no sensitive information is written. Also, look for `log4j` properties and a debug logging level in an application which typically indicate an issue.

Perform proper exception management, and always perform logging only to the extent required. Sensitive data like account numbers and passwords should not be logged.

The data items below should be logged:

- User name
- Time
- Action performed
- Application name

You should perform additional logging according to the application and business requirements.

Use strong passwords and not PINs

Android applications usually require a PIN for authentication instead of a password. The reason for this is to make applications easier to use, as it is difficult to enter a long password from a mobile device. However, it is important to point out that PINs are easier to bruteforce than complex passwords and can negatively impact the overall security of the target application.

For better security, consider using passwords for your applications instead of PINs. Best practices for creating secure passwords are summarized below::

- Use a password and not a PIN.
- Choose a password that uses alphanumeric characters and special characters.
- Choose a password that is at least seven characters in length.
- Ensure that a proper password expiration policy is implemented on server side.
- Do not allow users to reuse any of their last six passwords.

Perform data validation

Data validation issues in Android are usually not considered as serious during penetration testing or while performing a code review. However, this is a mistake. WebView becomes vulnerable to all browser attacks because WebView itself is a browser instance and has all the capabilities of a browser.

An Android application can be coded in Java or native code, which is C++. When Java is used, many of the data validation issues like buffer overflow, format string issues, and others are eliminated, as the language itself is not vulnerable. When using native code, special care needs to be taken when data is read from an untrusted source because it is vulnerable to issues like buffer overflow, format string issues, and more.

When performing data validation code review, it is necessary to identify the source and the sink. Source refers to the place where the data is received. Sink refers to the place where data is sent back to user. Once complete flow from the source to sink is understood, we can easily identify what kinds of issues there are, for example, XSS, SQL injection, buffer overflow, and many more data validation-related issues.

For more information on data validation issues, refer to **https://www.owasp.org**.

Common mistakes

Other common issues that occur when Android applications are being developed and that are similar to the web application penetration testing are mentioned below:

- Logout of a user in an Android application is usually done only on the client side by moving him to new screen. The application should also send an explicit logout request to the server to terminate server-side sessions.
- Session ID or sensitive data should not be sent in request URLs.
- Inadequate error-handling leads to sensitive information disclosure.

- Data entered via mobile applications, including Android applications, is often persisted in the backend servers without validation. The persisted data is then accessed via the web application, resulting in security issues like XSS, malicious URL injection, and others.
- Android applications are also vulnerable to web-related attacks like XSS, CSRF, XFS, and others when WebView is used.
- Session ID timeout is usually very long or sessions do not expire. Invalidate session IDs on both the client and server side after 30 minutes of inactivity.

References

- Professional-Android-4-Application-Development by Reto Meier
- **http://developer.Android.com/**
- **http://stackoverflow.com/questions/7295604/how-to-set-an-alarm-in-android-java**
- **http://proguard.sourceforge.net**
- **http://www.cis.syr.edu/~wedu/Research/paper/webview_acsac2011.pdf**
- **http://developer.Android.com/training/articles/security-tips.html**
- **http://stackoverflow.com/questions/4062838/intent-filter-within-a-service**
- **http://blog.opensecurityresearch.com/2012/04/acquiring-volatile-memory-from-Android.html**
- **http://bgr.com/2013/03/13/android-activation-growth-analysis-373572/**

About the Author

Naveen Rudrappa is a senior security consultant at McAfee Foundstone Professional Services. Rudrappa has more than seven years of experience in information security. He has also completed certificates such CEH and SCJP. Rudrappa focuses on web application penetration testing, thick client testing, mobile application testing, web services testing, code review, threat modeling, external network penetration testing, and other service lines.

About McAfee Foundstone Professional Services

McAfee Foundstone Professional Services, a division of McAfee, offers expert services and education to help organizations continuously and measurably protect their most important assets from the most critical threats. Through a strategic approach to security, McAfee Foundstone identifies and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively. The company's professional services team consists of recognized security experts and authors with broad security experience with multinational corporations, the public sector, and the US military.

www.foundstone.com.

About McAfee

McAfee is one of the world's leading independent cybersecurity companies. Inspired by the power of working together, McAfee creates business and consumer solutions that make the world a safer place. By building solutions that work with other companies' products, McAfee helps businesses orchestrate cyber environments that are truly integrated, where protection, detection and correction of threats happen simultaneously and collaboratively. By protecting consumers across all their devices, McAfee secures their digital lifestyle at home and away. By working with other security players, McAfee is leading the effort to unite against cybercriminals for the benefit of all.

www.mcafee.com.

McAfee™
Together is power.

2821 Mission College Blvd.
Santa Clara, CA 95054
888.847.8766
www.mcafee.com