

# ATTACKING ANDROID

WITH SECURE CODING PRACTICES  
IN JAVA



[WWW.DEVSECOPSGUIDES.COM](http://WWW.DEVSECOPSGUIDES.COM)

# Attacking Android

• Mar 11, 2024 •  35 min read

## Table of contents

### ContentProvider Management in Android Applications

- › Noncompliant Code Example:
- › Proof of Concept:
- › Compliant Solution:
- › Risk Assessment:

### Protecting Exported Services with Strong Permissions in Android Applications

- › Background:
- › Noncompliant Code Example:
- › Compliant Solutions:
- › Usage in Other Applications:
- › Risk Assessment:

### Protecting Against Directory Traversal Vulnerabilities in Android ContentProviders

- › Background:
- › Noncompliant Code Example 1:
- › Noncompliant Code Example 2:
- › Proof of Concept:
- › Compliant Solution:
- › Applicability:
- › Risk Assessment:

### Preventing Unauthorized Access to Sensitive Activities in Android Applications

- › Background:

- › Noncompliant Code Example:
- › Compliant Solution (Do not export activity):
- › Compliant Solution (Twicca):
- › Risk Assessment:

## Avoid Storing Sensitive Information on External Storage (SD Card) Without Encryption

- › Background:
- › Noncompliant Code Example:
- › Compliant Solution #1 (Save a File on Internal Storage):
- › Risk Assessment:

## Logging Sensitive Information in Android

- › Background:
- › Logging Sensitive Information:
- › Noncompliant Code Example:
- › Proof of Concept (Obtaining Log Output):
- › Compliant Solution:
- › Risk Assessment:

## Securing Sensitive Data in Android

- › Background:
- › Noncompliant Code Example:
- › Compliant Solution:
- › Risk Assessment:

## Cache

- › Noncompliant Code Example:
- › Compliant Solution:
- › Risk Assessment:

## Do not use world readable or writeable to share files between apps

- › Noncompliant Code Example 1:

- › Noncompliant Code Example 2:
- › For OAuth, use an explicit intent method to deliver access tokens
- › Noncompliant Code Example:
- › Compliant Solution:

Do not broadcast sensitive information using an implicit intent

Do not allow WebView to access sensitive local resource through file scheme

WebView Security Concerns:

- › Noncompliant Code Example:
- › Compliant Solution:
- › Risk Assessment:

Do not provide addJavascriptInterface method access in a WebView which could contain untrusted content. (API level JELLY\_BEAN or below)

- › Noncompliant Code Example:
- › Compliant Solutions:
- › Applicability:
- › Risk Assessment:

Enable serialization compatibility during class evolution

- › Noncompliant Code Example:
- › Compliant Solutions:
- › Risk Assessment:

Do not deviate from the proper signatures of serialization methods

- › Serialization Methods:
- › Compliant Solution for writeObject() and readObject():
- › readResolve() and writeReplace() Methods:
- › Noncompliant Code Examples for readResolve() and writeReplace():
- › Compliant Solution for readResolve() and writeReplace():
- › Risk Assessment:

## Exclude unsanitized user input from format strings

- › Noncompliant Code Example:
- › Compliant Solution:
- › Risk Assessment:

## Sanitize untrusted data included in a regular expression

- › Risks of Regex Injection:
- › Vulnerable Constructs in Regex:
- › Noncompliant Code Example:
- › Compliant Solutions:
- › Risk Assessment:

## Define wrappers around native methods

- › Native Method Overview:
- › Noncompliant Code Example:
- › Compliant Solution:
- › Exceptions:
- › Risk Assessment:

## Do not allow exceptions to expose sensitive information

- › Risks of Exception Propagation:
- › Noncompliant Code Example 1: Leaks from Exception Message and Type
- › Risk:
- › Noncompliant Code Example 2: Wrapping and Rethrowing Sensitive Exception
- › Risk:
- › Noncompliant Code Example 3: Sanitized Exception
- › Risk:
- › Compliant Solution 1: Security Policy
- › Solution:
- › Compliant Solution 2: Restricted Input



- › Solution:
- › Considerations:
- › Risk Assessment:

#### Do not encode noncharacter data as a string

- › Noncompliant Code Example:
- › Compliant Solution:
  - › Using toString() and getBytes():
- › Using Base64 Encoding:
- › Risk Assessment:

#### Do not release apps that are debuggable

- › Noncompliant Code Example:
- › Compliant Solution:
- › Risk Assessment:

#### Consider privacy concerns when using Geolocation API

- › Noncompliant Code Example:
- › Compliant Solution #1:
- › Compliant Solution #2:
- › Risk Assessment:

#### Properly verify server certificate on SSL/TLS

- › Noncompliant Code Example:
- › Compliant Solution:
- › Risk Assessment:

#### Specify permissions when creating files via the NDK

- › Noncompliant Code Example:
- › Compliant Solution (Set Umask):
- › Compliant Solution (Specify File Permissions):
- › Risk Assessment:

- › Sensitive classes must not let themselves be copied
- › Noncompliant Code Example:
- › Malicious Subclass:
- › Compliant Solution (Final Class):
- › Compliant Solution (Final clone()):

Risk Assessment:

References

Show less ^

In this comprehensive guide, we delve into the world of Android security from an offensive perspective, shedding light on the various techniques and methodologies used by attackers to compromise Android devices and infiltrate their sensitive data. From exploiting common coding flaws to leveraging sophisticated social engineering tactics, we explore the full spectrum of attack surfaces present in Android environments.

## ContentProvider Management in Android Applications

The `ContentProvider` class in Android facilitates data sharing among applications. Proper access control is crucial to prevent unauthorized access to sensitive data. There are three primary ways to control access:

### 1. Public Access:

- Making a `ContentProvider` public allows other applications to access its data.
- Use the `android:exported` attribute in the `AndroidManifest.xml` file to specify whether a `ContentProvider` is public.
- Before API Level 16, a `ContentProvider` is public by default unless `android:exported="false"` is explicitly set.
- Example code in `AndroidManifest.xml`:

```
<provider
    android:exported="true"
    android:name="MyContentProvider"
    android:authorities="com.example.mycontentprovider" />
```

### Private Access:

- A `ContentProvider` can be made private to restrict access from other applications.
- From API Level 17 onwards, a `ContentProvider` is private by default if `android:exported` is not specified.
- Example code in `AndroidManifest.xml`:

```
<provider
    android:exported="false"
    android:name="MyContentProvider"
    android:authorities="com.example.mycontentprovider" />
```

COPY 

### 3. Restricted Access:

- More details are needed for implementing restricted access. This section requires further elaboration.

### Noncompliant Code Example:

The example of noncompliant code demonstrates a Twitter client application inadvertently exposing sensitive information through a public `ContentProvider`.

### Proof of Concept:

The provided code snippet illustrates how the vulnerability in the `ContentProvider` can be exploited to extract sensitive data from the Twitter client application.

### Compliant Solution:



The compliant solution involves making the `ContentProvider` private in the `AndroidManifest.xml` file to prevent unauthorized access to sensitive data.

COPY 

```
<provider
    android:name=".content.AccountProvider"
    android:exported="false"
    android:authorities="jp.co.vulnerable.accountprovider" />
```

### Risk Assessment:

Declaring a `ContentProvider` as public without proper access control can lead to leakage of sensitive information to malicious applications.

## Protecting Exported Services with Strong Permissions in Android Applications

### Background:

The guideline is derived from the work of Chin et al. [Chin 2011], highlighting the risk associated with exported services that lack proper protection. Unprotected services can be exploited by any application, potentially leading to data leaks or unauthorized activities.

### Noncompliant Code Example:

The noncompliant code example demonstrates an exported service without adequate protection, allowing arbitrary applications to access sensitive information:

COPY 

```
<activity android:exported="false" ... >
    <intent-filter > ... </intent-filter>
    ...
</activity>
```

### Compliant Solutions:

## 1. Removing `<intent-filter>`:

- By removing the `<intent-filter>`, access to the service is restricted to components within the same application or applications with the same user ID.

## 2. Using Custom Permissions:

- If the intention is to allow access from other applications, custom permissions should be used instead of relying on default permissions like "normal." This prevents unauthorized access to confidential data.


### Compliant Solution Code:

COPY 

```
<permission android:name="customPermission"
android:protectionLevel="dangerous" ...></permission>

<activity
    android:permission="customPermission"
    ... >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter >
        <action android:name="package_name.MyAction" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

### Usage in Other Applications:

COPY 

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

```
Intent in = new Intent();
in.setAction("package_name.MyAction");
in.addCategory("android.intent.category.DEFAULT");
startActivity(in);
```

### Risk Assessment:

Failure to protect exported services with strong permissions poses a high risk, potentially leading to sensitive data exposure or denial of service attacks.

## Protecting Against Directory Traversal Vulnerabilities in Android ContentProviders

### Background:

The guideline warns about potential directory traversal vulnerabilities that can arise when using the `ContentProvider.openFile()` method in Android applications. Directory traversal vulnerabilities can allow an attacker to access files outside the intended directory, leading to unauthorized access or data corruption.

### Noncompliant Code Example 1:

The first noncompliant code example attempts to access a file using `Uri.getLastPathSegment()`, which may be vulnerable to directory traversal attacks when the path is URL encoded:

```
private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();

public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
    throws FileNotFoundException {
    File file = new File(IMAGE_DIRECTORY,
paramUri.getLastPathSegment());
    return ParcelFileDescriptor.open(file,
ParcelFileDescriptor.MODE_READ_ONLY);
}
```

## Noncompliant Code Example 2:

The second noncompliant code example tries to fix the vulnerability by decoding the URI string, but it still remains vulnerable to double encoding attacks:

```
private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();

public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
    throws FileNotFoundException {
    File file = new File(IMAGE_DIRECTORY,
Uri.parse(paramUri.getLastPathSegment()).getLastPathSegment());
    return ParcelFileDescriptor.open(file,
ParcelFileDescriptor.MODE_READ_ONLY);
}
```

COPY 

## Proof of Concept:

Malicious code can exploit these vulnerabilities by supplying specially crafted URI strings to the content provider, leading to unauthorized file access or traversal.

## Compliant Solution:

The compliant solution ensures protection against directory traversal by decoding the URI string and canonicalizing the file path:

```
private static String IMAGE_DIRECTORY = localFile.getAbsolutePath();

public ParcelFileDescriptor openFile(Uri paramUri, String paramString)
    throws FileNotFoundException {
    String decodedUriString = Uri.decode(paramUri.toString());
    File file = new File(IMAGE_DIRECTORY,
Uri.parse(decodedUriString).getLastPathSegment());
    if
(!file.getCanonicalPath().startsWith(localFile.getCanonicalPath())) {
        throw new IllegalArgumentException("Path traversal attempt
detected!");
    }
}
```

COPY 

```
return ParcelFileDescriptor.open(file,  
ParcelFileDescriptor.MODE_READ_ONLY);  
}
```

### Applicability:

This guideline is applicable to any Android application that exchanges files through a ContentProvider, ensuring protection against directory traversal attacks.

### Risk Assessment:

Failure to properly decode and canonicalize file paths received by a ContentProvider may result in directory traversal vulnerabilities, leading to unauthorized access or corruption of sensitive data.

## Preventing Unauthorized Access to Sensitive Activities in Android Applications

### Background:

Declaring an intent filter for an activity in the AndroidManifest.xml file exposes the activity to other applications, potentially allowing unauthorized access. Malicious apps could exploit this vulnerability to misuse sensitive functionalities of the exposed activity.

### Noncompliant Code Example:

The noncompliant code example shows an AndroidManifest.xml file exporting an activity without restricting access:

```
<activity  
    android:configChanges="keyboard|keyboardHidden|orientation"  
    android:name=".media.yfrog.YfrogUploadDialog"  
    android:theme="@style/Vulnerable.Dialog"  
    android:windowSoftInputMode="stateAlwaysHidden">  
  
    <intent-filter android:icon="@drawable/yfrog_icon"
```

COPY 

```

android:label="@string/YFROG">
    <action android:name="jp.co.vulnerable.ACTION_UPLOAD" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="image/*" />
    <data android:mimeType="video/*" />
</intent-filter>
</activity>

```

### Compliant Solution (Do not export activity):

In the compliant solution, the activity is not exported, restricting it to only accept intents from within the same app or apps with the same user ID:

```

<activity
    android:configChanges="keyboard|keyboardHidden|orientation"
    android:name=".media.yfrog.YfrogUploadDialog"
    android:theme="@style/VulnerableTheme.Dialog"
    android:windowSoftInputMode="stateAlwaysHidden"
    android:exported="false">
</activity>

```

COPY 

### Compliant Solution (Twicca):

In Twicca's case, instead of declaring the activity as not exported, they implemented caller validation within the activity itself. This ensures that the activity only proceeds if called from the same package:

```

public void onCreate(Bundle arg5) {
    super.onCreate(arg5);
    ...
    ComponentName v0 = this.getCallingActivity();
    if(v0 == null) {
        this.finish();
    } else if(!jp.r246.twicca.equals(v0.getPackageName())) {
        this.finish();
    }
}

```

COPY 



```

    } else {
        this.a = this.getIntent().getData();
        if(this.a == null) {
            this.finish();
        }
        ...
    }
}

```

### Risk Assessment:

Failure to validate the caller's identity before acting on received intents may lead to sensitive data exposure or denial of service attacks.

## Avoid Storing Sensitive Information on External Storage (SD Card) Without Encryption

### Background:

Android provides external storage options like SD cards for storing application data. However, files stored on external storage are not guaranteed to be secure and can be accessed by other apps or users. Storing sensitive information without encryption on external storage poses significant security risks.

### Noncompliant Code Example:

The following code snippet demonstrates storing sensitive information directly to external storage without encryption:

```

private String filename = "myfile";
private String string = "sensitive data such as credit card number";
FileOutputStream fos = null;

try {
    File file = new File(getExternalFilesDir(TARGET_TYPE), filename);
    fos = new FileOutputStream(file, false);
    fos.write(string.getBytes());
}

```

COPY 

```

    } catch (FileNotFoundException e) {
        // handle FileNotFoundException
    } catch (IOException e) {
        // handle IOException
    } finally {
        if (fos != null) {
            try {
                fos.close();
            } catch (IOException e) {
                // handle error
            }
        }
    }
}

```

### Compliant Solution #1 (Save a File on Internal Storage):

The compliant solution stores sensitive information in the internal storage directory with permission set to `MODE_PRIVATE`, ensuring that other apps cannot access the file:

COPY 

```

private String filename = "myfile";
private String string = "sensitive data such as credit card number";
FileOutputStream fos = null;

try {
    fos = openFileOutput(filename, Context.MODE_PRIVATE);
    fos.write(string.getBytes());
} catch (FileNotFoundException e) {
    // handle FileNotFoundException
} catch (IOException e) {
    // handle IOException
} finally {
    if (fos != null) {
        try {
            fos.close();
        } catch (IOException e) {
            // handle error
        }
    }
}

```

```
}  
  
}
```

### Risk Assessment:

Storing sensitive information on external storage without encryption can lead to data leakage to malicious apps, compromising the confidentiality of the data. Implementing proper encryption or storing data in internal storage mitigates this risk.

## Logging Sensitive Information in Android

### Background:

Android provides built-in logging mechanisms through the `android.util.Log` class. Developers use various logging levels (`Log.d`, `Log.e`, `Log.i`, `Log.v`, `Log.w`) to output debugging information, errors, and other messages to the system log. However, logging sensitive information, such as access tokens or user location data, can pose security risks.

### Logging Sensitive Information:

Developers should avoid logging sensitive information using the `android.util.Log` class. Instead, they should be cautious and ensure that only non-sensitive data is logged for debugging purposes.

### Noncompliant Code Example:

```
// Logging sensitive information (access token) using Log.d  
Log.d("Facebook-authorize", "Login Success! access_token=" +  
    + getAccessToken() + " expires=" +  
    + getAccessExpires());
```

COPY 

### Proof of Concept (Obtaining Log Output):

```
// Example code to obtain log output from a vulnerable application
final StringBuilder slog = new StringBuilder();

try {
    Process mLogcatProc;
    mLogcatProc = Runtime.getRuntime().exec(new String[]
        {"logcat", "-d", "LoginAsyncTask:I APIClient:I method:V *:S" });

    BufferedReader reader = new BufferedReader(new InputStreamReader(
        mLogcatProc.getInputStream()));

    String line;
    String separator = System.getProperty("line.separator");

    while ((line = reader.readLine()) != null) {
        slog.append(line);
        slog.append(separator);
    }
    Toast.makeText(this, "Obtained log information",
        Toast.LENGTH_SHORT).show();

} catch (IOException e) {
    // handle error
}

TextView tView = (TextView) findViewById(R.id.logView);
tView.setText(slog);
```

### Compliant Solution:

Developers should ensure that sensitive information is not logged. They can achieve this by:

1. Reviewing code to identify and remove any logging of sensitive information.
2. Using custom logging mechanisms that automatically turn off logging in release builds.

3. Employing obfuscation tools like ProGuard to remove specific logging calls.

### Risk Assessment:

Logging sensitive information can lead to data leakage, compromising user privacy and potentially exposing sensitive data to malicious apps.

## Securing Sensitive Data in Android

### Background:

Android apps often handle sensitive data, such as user credentials or personal information. It's crucial to protect this data from unauthorized access by other apps or entities. Data security measures include using appropriate file modes when creating files or databases and employing encryption for added protection.

### Noncompliant Code Example:

```
// Creating a file that is world-readable (noncompliant)
openFileOutput("someFile", MODE_WORLD_READABLE);
```

COPY 

In this example, the file is created with the `MODE_WORLD_READABLE` flag, allowing any application to read its contents, which poses a security risk.

### Compliant Solution:

```
// Creating a file with MODE_PRIVATE (compliant)
openFileOutput("someFile", MODE_PRIVATE);
```

COPY 

By using `MODE_PRIVATE`, the file can only be accessed by the app that created it, ensuring data security.

### Risk Assessment:

Failure to secure sensitive data can lead to data leakage, compromising user privacy and potentially exposing confidential information to unauthorized apps or entities.

## Cache

Caching data can pose security risks as it may become accessible to other applications or unauthorized users if the device is lost or stolen. Here's a breakdown of the key points mentioned in the guideline:

1. **Caching web application data:** Storing data such as URL histories, HTTP headers, HTML form inputs, and cookies in the cache can expose sensitive information to other applications.
2. **Keyboard cache:** Words entered by the user via the keyboard are stored in the Android user dictionary for auto-correction. This data is accessible to any app without requiring permission, potentially leading to the leakage of sensitive information.
3. **Cached camera images:** Apps may cache images captured by the device's camera, which can remain accessible even after the app has finished. Storing such images without proper security measures can compromise user privacy.
4. **GUI objects caching:** Application screens retained in memory can enable access to transaction histories by anyone with device access. Caching sensitive information in memory increases the risk of data exposure.

To address these concerns, developers should:

- Avoid caching sensitive information whenever possible.
- Use methods like `clearCache()` to delete cached data, especially when accessing sensitive data with a `WebView`.
- Utilize server-side headers like `no-cache` to prevent caching of particular content by the application.

### Noncompliant Code Example:




```
// Caching web application data (noncompliant)
webView.getSettings().setCacheMode(WebSettings.LOAD_DEFAULT);
```

In this noncompliant code example, web application data is cached using the default cache mode, potentially revealing sensitive information such as URL histories, HTTP headers, and cookies.

### Compliant Solution:

```
// Avoid caching web application data (compliant)
webView.getSettings().setCacheMode(WebSettings.LOAD_NO_CACHE);
```

COPY 

By setting the cache mode to `LOAD_NO_CACHE`, caching of web application data is disabled, reducing the risk of sensitive information leakage.

### Risk Assessment:

Failing to prevent caching of sensitive data can lead to unauthorized access to user information, compromising privacy and security. It's important to implement measures to avoid caching where sensitive data is involved.

## Do not use world readable or writeable to share files between apps

This guideline warns against the use of constants `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` in Android applications, as they can create security vulnerabilities. Instead, developers are encouraged to utilize more formal mechanisms for interactions and communication between applications, such as `ContentProvider`, `BroadcastReceiver`, and `Service`. Here's an explanation of each mechanism:

1. **ContentProvider:** This component provides content to applications and is used to share data among multiple applications. It offers a structured and secure way to

expose data to other applications while maintaining control over access permissions.

2. **BroadcastReceiver:** Broadcasts are sent when an event of interest occurs, and applications can register to receive certain broadcasts. BroadcastReceiver serves as a messaging system among applications, allowing them to communicate and respond to system-wide events.
3. **Service:** Services allow applications to perform background tasks and expose some of their functionality to other applications. They enable long-running operations to be executed independently of the application's UI.

The noncompliant code examples demonstrate the use of `MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE` constants to share files between applications, which is discouraged due to its security implications. These constants allow other applications to read or write the file using the shared preference API, potentially leading to security holes.

#### Noncompliant Code Example 1:

```
String FILENAME = "example_file";
String string = "hello world!";


FileOutputStream outputStream = openFileOutput(FILENAME,
Context.MODE_WORLD_READABLE);
outputStream.write(string.getBytes());
outputStream.close();
```

COPY 

#### Noncompliant Code Example 2:

```
String FILENAME = "example_file";
String string = "hello world!";

FileOutputStream outputStream = openFileOutput(FILENAME,
Context.MODE_WORLD_WRITABLE);
```

COPY 

```
outputStream.write(string.getBytes());  
outputStream.close();
```

By avoiding the use of these constants and opting for more secure mechanisms like `ContentProvider`, `BroadcastReceiver`, and `Service`, developers can minimize the risk of security vulnerabilities in their Android applications.

### For OAuth, use an explicit intent method to deliver access tokens

This guideline emphasizes the importance of using explicit intents over implicit intents in Android applications to protect user information and prevent potential security risks. Explicit intents specify the target component explicitly, while implicit intents declare general actions that all applications can use, potentially exposing sensitive user actions.

#### Noncompliant Code Example:


```
protected void OnTokenAcquired(Bundle savedInstanceState) {  
    //[Code to construct an OAuth client request goes here]  
    Intent intent = new Intent(Intent.ACTION_VIEW,  
        Uri.parse(request.getLocationUri() + "&response_type=code"));  
    startActivity(intent);  
}
```

COPY 

In the noncompliant code example, an implicit intent is used to send access tokens by invoking a specific action without specifying the target component explicitly. This approach can lead to security vulnerabilities and expose user information to unintended recipients.

#### Compliant Solution:

```
protected void OnTokenAcquired(Bundle savedInstanceState) {  
    //[Code to construct an OAuth client request goes here]
```

COPY 

```

Intent intent = new Intent(this, YourOAuthActivity.class);
intent.setAction(Intent.ACTION_VIEW);
intent.setData(Uri.parse(request.getLocationUri() +
"&response_type=code"));
startActivity(intent);
}

```

In the compliant solution, an explicit intent is used to send access tokens by specifying the target component (YourOAuthActivity.class) explicitly. This approach ensures that the intent is directed only to the intended recipient, enhancing security and protecting user information.

## Do not broadcast sensitive information using an implicit intent


This guideline focuses on securing broadcast intents in Android applications to prevent sensitive information leakage or denial of service attacks. It emphasizes the importance of using explicit intents or other secure mechanisms to restrict the receivers of broadcast intents.

Noncompliant Code Example:

```

public class ServerService extends Service {
    // ...
    private void d() {
        // ...
        Intent v1 = new Intent();
        v1.setAction("com.sample.action.server_running");
        v1.putExtra("local_ip", v0.h);
        v1.putExtra("port", v0.i);
        v1.putExtra("code", v0.g);
        v1.putExtra("connected", v0.s);
        v1.putExtra("pwd_predefined", v0.r);
        if (!TextUtils.isEmpty(v0.t)) {
            v1.putExtra("connected_usr", v0.t);
        }
        this.sendBroadcast(v1);
    }
}

```

COPY 

```
}  
}
```

In this noncompliant code example, an implicit intent is used to broadcast sensitive information such as the device's IP address, port number, and password. This approach allows any application, including malicious ones, to receive the broadcast message, potentially leading to data leakage or denial of service attacks.

Proof of Concept:


```
public class BcReceiver extends BroadcastReceiver {  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        if  
(intent.getAction().equals("com.sample.action.server_running")) {  
            String pwd = intent.getStringExtra("connected");  
            String message = "Received sensitive data: [" + pwd + "];"  
            Toast.makeText(context, message,  
Toast.LENGTH_SHORT).show();  
        }  
    }  
}
```

COPY 

The proof of concept demonstrates how a malicious broadcast receiver can intercept the implicit intent and access sensitive data sent by the vulnerable application.

Compliant Solution:

```
Intent intent = new Intent("my-sensitive-event");  
intent.putExtra("event", "this is a test event");  
LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
```

COPY 

In the compliant solution, LocalBroadcastManager is used to send the broadcast intent. This ensures that the intent is only broadcast and received within the same application, preventing other apps from accessing sensitive information.

**Risk Assessment:** Using implicit intents for broadcasting sensitive information can expose the data to malicious apps or lead to denial of service attacks. Employing explicit intents, LocalBroadcastManager, or other secure mechanisms can mitigate these risks and enhance the security of the application.

## Do not allow WebView to access sensitive local resource through file scheme

This guideline focuses on security concerns and best practices when using the WebView class in Android applications. It provides insights into several methods within WebView that can introduce security vulnerabilities if not used correctly.

### WebView Security Concerns:

#### 1. **setJavaScriptEnabled():**

- Enables JavaScript execution within the WebView.
- Default: false.
- **Risk:** Allows JavaScript code execution, potentially leading to security breaches.

#### 2. **setPluginState():**

- Enables or disables plugins in the WebView.
- Options: ON (always load), ON\_DEMAND (load if plugin exists), OFF (disable all).
- Default: OFF.
- **Risk:** Malicious content may exploit enabled plugins, compromising security.

#### 3. **setAllowFileAccess():**

- Enables or disables file access within the WebView.



- Default: true.
- **Risk:** Allows or restricts access to local files, impacting security based on application requirements.

#### 4. **setAllowContentAccess():**

- Enables or disables content URL access within the WebView.
- Default: true.
- **Risk:** Controls access to content providers, affecting security based on application needs.

#### 5. **setAllowFileAccessFromFileURLs():**

- Controls JavaScript access to content from file scheme URLs.
- Default: true or false based on API level.
- **Risk:** Determines if JavaScript running in file scheme URLs can access other file scheme URLs, affecting security based on API level.

#### 6. **setAllowUniversalAccessFromFileURLs():**

- Controls JavaScript access to content from any origin from file scheme URLs.
- Default: true or false based on API level.
- **Risk:** Affects JavaScript access to content from file scheme URLs, impacting security based on API level.

### Noncompliant Code Example:

The provided code demonstrates an activity with a WebView component that enables JavaScript and processes any URI passed through an Intent without validation. This setup can lead to potential security vulnerabilities, especially if malicious content is loaded.

COPY 

```
// Noncompliant Code Example
public class MyBrowser extends Activity {
    @Override
```

```

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

    WebView webView = (WebView) findViewById(R.id.webview);

    // turn on javascript
    WebSettings settings = webView.getSettings();
    settings.setJavaScriptEnabled(true);

    String url = getIntent().getStringExtra("URL");
    webView.loadUrl(url);
}
}

// Proof of Concept
// Malicious application prepares some crafted HTML file,
// places it on a local storage, makes accessible from
// other applications. The following code sends an
// intent to a target application (jp.vulnerable.android.app)
// to make it access and process the malicious HTML file.

String pkg = "jp.vulnerable.android.app";
String cls = pkg + ".DummyLauncherActivity";
String uri = "file:///[crafted HTML file]";
Intent intent = new Intent();
intent.setClassName(pkg, cls);
intent.putExtra("url", uri);
this.startActivity(intent);

// Compliant Solution
public class MyBrowser extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        WebView webView = (WebView) findViewById(R.id.webview);

        String url = getIntent().getStringExtra("url");

```

```
        if (!url.startsWith("http")) { /* Note:
"https".startsWith("http") == true */
            url = "about:blank";
        }

        webView.loadUrl(url);
    }
}
```

The noncompliant code initializes a WebView object, enables JavaScript, and loads a URL received via Intent without proper validation. This can lead to security vulnerabilities if malicious content is loaded.

The proof of concept demonstrates how a malicious application can exploit this vulnerability by sending an intent to a target application with a crafted HTML file URI.

The compliant solution validates the received URL to ensure it starts with "http" or "https" before loading it into the WebView. This helps mitigate the risk of loading potentially malicious local content.

### **Compliant Solution:**

A compliant solution validates URIs received via Intent before rendering them with WebView. It checks that the URI starts with "http" to ensure that only trusted URLs are loaded. This approach helps mitigate the risk of loading malicious local content.

### **Risk Assessment:**

Failure to implement proper security measures in WebView usage can result in information leaks and other security breaches. It's essential to validate input and configure WebView settings carefully to minimize these risks.

**Do not provide addJavascriptInterface method access in a WebView which could contain untrusted content. (API level JELLY\_BEAN or below)**

coding guideline regarding the usage of the `addJavaScriptInterface` method in Android applications, specifically targeting API levels JELLY\_BEAN and below. This method, when used with untrusted content in a WebView, can expose the application to scripting attacks, potentially compromising sensitive data and app control.

### Noncompliant Code Example:

The noncompliant code demonstrates an insecure usage of the `addJavaScriptInterface` method:

```
WebView webView = new WebView(this);
setContentView(webView);

class JsObject {
    private String sensitiveInformation;

    public String toString() {
        return sensitiveInformation;
    }
}

webView.addJavaScriptInterface(new JsObject(), "injectedObject");
webView.loadData("", "text/html", null);
webView.loadUrl("http://www.example.com");
```

COPY 

By allowing JavaScript to control the host, this code opens up avenues for potential scripting attacks, utilizing Java reflection to access public methods of injected objects and thereby gaining access to sensitive information.

### Compliant Solutions:

1. **Refrain from using `addJavaScriptInterface`:** In this approach, the code refrains from using the `addJavaScriptInterface` method altogether.

```
WebView webView = new WebView(this);  
setContentView(webView);
```

**Specify minimum SDK version in manifest:** Another compliant solution involves specifying in the app's manifest that it's only intended for API levels JELLY\_BEAN\_MR1 and above, where only public methods annotated with `JavascriptInterface` can be accessed from JavaScript.

```
<manifest>  
    <uses-sdk android:minSdkVersion="17" />  
    ...  
</manifest>
```

COPY 

#### Applicability:

- **Android Version Applicability:** Applies to Android API versions 16 (JELLY\_BEAN) and below.

#### Risk Assessment:

- **Risk Level:** High
- **Probability:** Probable
- **Impact:** Medium
- **Priority:** P12
- **Likelihood:** L1

## Enable serialization compatibility during class evolution

coding guideline regarding the serialization of classes in Java, emphasizing the importance of maintaining compatibility across different versions of a class.

#### Noncompliant Code Example:

The provided noncompliant code demonstrates a class `GameWeapon` that implements `Serializable` without specifying a `serialVersionUID`, thus relying on the default serialized form. Any changes to the internal structure of the class may break compatibility with previously serialized objects.

COPY 

```
class GameWeapon implements Serializable {
    int numOfWeapons = 10;

    public String toString() {
        return String.valueOf(numOfWeapons);
    }
}
```

## Compliant Solutions:

1. **Using `serialVersionUID`:** In this solution, the class declares a `private static final long serialVersionUID`, providing a unique identifier for the class version. This allows the JVM to deserialize objects even if the class definition has changed, as long as the version ID remains the same.

COPY 

```
class GameWeapon implements Serializable {
    private static final long serialVersionUID = 24L;

    int numOfWeapons = 10;

    public String toString() {
        return String.valueOf(numOfWeapons);
    }
}
```

**Using `serialPersistentFields`:** Another compliant approach involves using custom serialization with `serialPersistentFields`. This allows more flexibility in the serialization process by specifying which fields should be serialized. It also separates



the serialized fields from the class implementation, making it easier to evolve the class without breaking compatibility.

COPY 

```
class WeaponStore implements Serializable {
    int numOfWeapons = 10; // Total number of weapons
}

public class GameWeapon implements Serializable {
    WeaponStore ws = new WeaponStore();
    private static final ObjectStreamField[] serialPersistentFields
        = {new ObjectStreamField("ws", WeaponStore.class)};

    private void readObject(ObjectInputStream ois)
        throws IOException, ClassNotFoundException {
        ObjectInputStream.GetField gf = ois.readFields();
        this.ws = (WeaponStore) gf.get("ws", ws);
    }

    private void writeObject(ObjectOutputStream oos) throws IOException
    {
        ObjectOutputStream.PutField pf = oos.putFields();
        pf.put("ws", ws);
        oos.writeFields();
    }

    public String toString() {
        return String.valueOf(ws);
    }
}
```

#### Risk Assessment:

- **Risk Level:** Low
- **Probability:** Probable
- **Impact:** High

- **Priority:** P2
- **Likelihood:** L3

## Do not deviate from the proper signatures of serialization methods

coding guidelines for implementing special methods required for object serialization and deserialization in Java, emphasizing the importance of adhering to specific method signatures and access specifiers to ensure proper functionality and security.

### Serialization Methods:

1. `writeObject()` and `readObject()`: These methods are responsible for custom serialization and deserialization of objects. They must have the following signatures:

```
private void writeObject(java.io.ObjectOutputStream  
out) throws IOException;  
private void readObject(java.io.ObjectInputStream in) throws  
IOException, ClassNotFoundException;
```

COPY 

### Compliant Solution for `writeObject()` and `readObject()`:

Ensure the methods are private and non-static to limit accessibility and adhere to required signatures.

```
private void writeObject(final  
ObjectOutputStream stream) throws IOException {  
    stream.defaultWriteObject();  
}  
  
private void readObject(final ObjectInputStream stream) throws  
IOException, ClassNotFoundException {
```

COPY 

```
stream.defaultReadObject();  
}
```

### `readResolve()` and `writeReplace()` Methods:

These methods allow classes to control the types and instances of objects being serialized and deserialized. They can be used to replace or resolve objects before they are returned or written to the stream. While it's possible to add any access specifier to these methods, it's recommended to avoid making them private or static to ensure proper functionality and extensibility.

### Noncompliant Code Examples for `readResolve()` and `writeReplace()`:

```
// Private declaration  
class Extendable implements Serializable {  
    private Object readResolve() {  
        // ...  
    }  
  
    private Object writeReplace() {  
        // ...  
    }  
}  
  
// Static declaration  
class Extendable implements Serializable {  
    protected static Object readResolve() {  
        // ...  
    }  
  
    protected static Object writeReplace() {  
        // ...  
    }  
}
```

COPY 

### Compliant Solution for `readResolve()` and `writeReplace()`:

To ensure proper inheritance and functionality, declare these methods as protected and non-static.

COPY 

```
class Extendable implements Serializable {  
    protected Object readResolve() {  
        // ...  
    }  
  
    protected Object writeReplace() {  
        // ...  
    }  
}
```

#### Risk Assessment:

- **Risk Level:** High
- **Probability:** Likely
- **Impact:** Low
- **Priority:** P27
- **Likelihood:** L1

## Exclude unsanitized user input from format strings

guidelines regarding the usage of the `format()` and `printf()` methods in the `PrintStream` class from the `java.io` package. These methods are used for formatting output, but they pose security risks if untrusted data is incorporated into the format string.

#### Noncompliant Code Example:

The noncompliant code incorporates untrusted data into a format string, potentially leaking sensitive information or allowing a denial-of-service attack.


```

class Format {
    static Calendar c = new GregorianCalendar(1995,
GregorianCalendar.MAY, 23);
    public static void main(String[] args) {
        // args[0] should contain the credit card expiration date
        // but might contain %1$tm, %1$te or %1$tY format specifiers
        System.out.format(
            args[0] + " did not match! HINT: It was issued on %1$terd of
some month", c
        );
    }
}

```

### Compliant Solution:

The compliant solution avoids incorporating untrusted user input into the format string, rendering any format specifiers inert.

COPY 

```

class Format {
    static Calendar c = new GregorianCalendar(1995,
GregorianCalendar.MAY, 23);
    public static void main(String[] args) {
        // args[0] is the credit card expiration date
        // Perform comparison with c,
        // if it doesn't match, print the following line
        System.out.format(
            "%s did not match! HINT: It was issued on %terd of some month",
            args[0], c
        );
    }
}

```

### Risk Assessment:

- **Risk Level:** Medium

- **Probability:** Unlikely
- **Impact:** Medium
- **Priority:** P4
- **Likelihood:** L3

## Sanitize untrusted data included in a regular expression

discusses the risks associated with regular expression (regex) injection and provides compliant solutions to mitigate these risks. It emphasizes the importance of sanitizing untrusted input used in regex patterns to prevent potential security vulnerabilities.

### Risks of Regex Injection:

Regex injection occurs when untrusted input is incorporated into a regex pattern, allowing attackers to modify the original pattern in a way that deviates from the program's intended behavior. This can lead to control flow manipulation, information leaks, or denial-of-service (DoS) vulnerabilities.


### Vulnerable Constructs in Regex:

1. **Matching flags:** Untrusted inputs may override matching options passed to the `Pattern.compile()` method.
2. **Greediness:** Injection attempts may change the regex to match as much of the string as possible, potentially exposing sensitive information.
3. **Grouping:** Attackers may manipulate regex groupings by supplying untrusted input.

### Noncompliant Code Example:

The provided code dynamically constructs a regex pattern using untrusted user input, making it vulnerable to regex injection attacks.

```
public static void FindLogEntry(String search) {  
    // Construct regex dynamically from user string  
    String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";
```

COPY 

```
// ...  
}
```

## Compliant Solutions:

1. **Whitelisting:** Sanitize search terms by filtering out non-alphanumeric characters.

```
public static void FindLogEntry(String search) {  
    // Sanitize search string  
    StringBuilder sb = new StringBuilder(search.length());  
    for (int i = 0; i < search.length(); ++i) {  
        char ch = search.charAt(i);  
        if (Character.isLetterOrDigit(ch) || ch == ' ' || ch == '\\')  
    {  
        sb.append(ch);  
    }  
    }  
    search = sb.toString();  
    // Construct regex dynamically from sanitized user string  
    String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";  
    // ...  
}
```

COPY 

2. **Using** `Pattern.quote()`: Escape any potentially malicious characters in the search string.

```
public static void FindLogEntry(String search) {  
    // Sanitize search string  
    search = Pattern.quote(search);  
    // Construct regex dynamically from sanitized user string  
    String regex = "(.*? +public\\[\\d+\\] +.*" + search + ".*)";  
    // ...  
}
```

COPY 

## Risk Assessment:

- **Risk Level:** Medium
- **Probability:** Unlikely
- **Impact:** Medium
- **Priority:** P4
- **Likelihood:** L3

## Define wrappers around native methods

highlights the importance of defining wrapper methods for native methods in Java, written in languages like C and C++, to ensure security and maintainability. It emphasizes the necessity of validating inputs, performing security checks, and defensive copying to prevent vulnerabilities such as buffer overflows.

### Native Method Overview:

Native methods provide extensibility to Java by allowing integration with code written in languages like C and C++. However, using native methods can compromise portability and flexibility due to deviation from Java's policies.

### Noncompliant Code Example:

The provided code exposes a native method `nativeOperation()` publicly, allowing untrusted callers to invoke it directly, bypassing security checks.

```
public final class NativeMethod {  
  
    // Public native method  
    public native void nativeOperation(byte[] data, int offset, int  
len);  
  
    // Wrapper method that lacks security checks and input validation  
    public void doOperation(byte[] data, int offset, int len) {  
        nativeOperation(data, offset, len);  
    }  
}
```


COPY 



```
static {  
    // Load native library in static initializer of class  
    System.loadLibrary("NativeMethodLib");  
}  
}
```

### Compliant Solution:

The compliant solution addresses security concerns by declaring the native method as private and implementing a `doOperation()` wrapper method with proper security checks, input validation, and defensive copying.

COPY 

```
public final class NativeMethodWrapper {  
  
    // Private native method  
    private native void nativeOperation(byte[] data, int offset, int  
len);  
  
    // Wrapper method performs SecurityManager and input validation  
checks  
    public void doOperation(byte[] data, int offset, int len) {  
        // Permission needed to invoke native method  
        securityManagerCheck();  
  
        if (data == null) {  
            throw new NullPointerException();  
        }  
  
        // Copy mutable input  
        data = data.clone();  
  
        // Validate input  
        if ((offset < 0) || (len < 0) || (offset > (data.length - len))) {  
            throw new IllegalArgumentException();  
        }  
    }  
}
```

```

        nativeOperation(data, offset, len);
    }

    static {
        // Load native library in static initializer of class
        System.loadLibrary("NativeMethodLib");
    }
}

```

## Exceptions:

- **JN100-J-EX0:** Native methods like `int rand(void)` that don't require security manager checks, argument validation, or defensive copying do not need wrapping.

## Risk Assessment:

- **Risk Level:** Medium
- **Probability:** Probable
- **Impact:** High
- **Priority:** P4
- **Likelihood:** L3

## Do not allow exceptions to expose sensitive information

importance of filtering sensitive information when propagating exceptions in Java programs to prevent information leaks that could aid attackers in developing exploits. It discusses the risks associated with exposing exception messages and types, providing examples of problematic exceptions and noncompliant code.

Exception Name	Description of Information Leak or Threat
<code>java.io.FileNotFoundException</code>	Underlying file system structure, user name enumeration
<code>java.sql.SQLException</code>	Database structure, user name enumeration


Exception Name	Description of Information Leak or Threat
<code>java.net.BindException</code>	Enumeration of open ports when untrusted client can choose server port
<code>java.util.ConcurrentModificationException</code>	May provide information about thread-unsafe code
<code>javax.naming.InsufficientResourcesException</code>	Insufficient server resources (may aid DoS)
<code>java.util.MissingResourceException</code>	Resource enumeration
<code>java.util.jar.JarException</code>	Underlying file system structure
<code>java.security.acl.NotOwnerException</code>	Owner enumeration
<code>java.lang.OutOfMemoryError</code>	DoS
<code>java.lang.StackOverflowError</code>	DoS

Printing the stack trace can also result in unintentionally leaking information about the structure and state of the process to an attacker. When a Java program that is run within a console terminates because of an uncaught exception, the exception's message and stack trace are displayed on the console; the stack trace may itself contain sensitive information about the program's internal structure. Consequently, any program that may be run on a console accessible to an untrusted user must never abort due to an uncaught exception.

Risks of Exception Propagation:

- Information Leaks:** Exceptions can reveal sensitive details about the application's internal structure, system configuration, or user environment.
- Denial of Service (DoS):** Attackers can exploit exceptions to gather information for potential DoS attacks or exploit vulnerabilities.

Noncompliant Code Example 1: Leaks from Exception Message and Type

COPY 

```
class ExceptionExample {
    public static void main(String[] args) throws FileNotFoundException
    {
```

```

        FileInputStream fis =
            new FileInputStream(System.getenv("APPDATA") + args[0]);
    }
}

```

#### Risk:

- Exposes sensitive information about the file system layout to attackers.
- Allows attackers to reconstruct the underlying file system by passing fictitious path names.

### Noncompliant Code Example 2: Wrapping and Rethrowing Sensitive Exception

```

try {
    FileInputStream fis =
        new FileInputStream(System.getenv("APPDATA") + args[0]);
} catch (FileNotFoundException e) {
    // Log the exception
    throw new IOException("Unable to retrieve file", e);
}

```

COPY 

#### Risk:

- Even when the logged exception is not directly accessible to the user, the original exception can still provide information about the file system layout to attackers.

### Noncompliant Code Example 3: Sanitized Exception

```

class SecurityIOException extends IOException { /* ... */ };

try {
    FileInputStream fis =
        new FileInputStream(System.getenv("APPDATA") + args[0]);
} catch (FileNotFoundException e) {

```

COPY 

```
// Log the exception
throw new SecurityIOException();
}
```

## Risk:

- Although this approach is less likely to leak useful information, it still reveals that the specified file cannot be read, enabling attackers to infer sensitive details about the file system.

## Compliant Solution 1: Security Policy

COPY 

```
class ExceptionExample {
    public static void main(String[] args) {

        File file = null;
        try {
            file = new File(System.getenv("APPDATA") +
                args[0]).getCanonicalFile();
            if (!file.getPath().startsWith("c:\\\\homepath")) {
                System.out.println("Invalid file");
                return;
            }
        } catch (IOException x) {
            System.out.println("Invalid file");
            return;
        }

        try {
            FileInputStream fis = new FileInputStream(file);
        } catch (FileNotFoundException x) {
            System.out.println("Invalid file");
            return;
        }
    }
}
```

## Solution:

- Implements a security policy that restricts file access to a specific directory ( `c:\homepath` ).
- Provides a generic error message to conceal information about the file system layout outside the permitted directory.

## Compliant Solution 2: Restricted Input

COPY 

```
class ExceptionExample {
    public static void main(String[] args) {
        FileInputStream fis = null;
        try {
            switch(Integer.valueOf(args[0])) {
                case 1:
                    fis = new FileInputStream("c:\\homepath\\file1");
                    break;
                case 2:
                    fis = new FileInputStream("c:\\homepath\\file2");
                    break;
                //...
                default:
                    System.out.println("Invalid option");
                    break;
            }
        } catch (Throwable t) {
            MyExceptionReporter.report(t); // Sanitize
        }
    }
}
```

## Solution:

- Operates under the policy that only specific files ( `c:\homepath\file1` and `c:\homepath\file2` ) are permitted to be opened by the user.

- Uses a centralized exception reporting mechanism ( `MyExceptionReporter` ) to filter sensitive information from any resulting exceptions.

### Considerations:

- **Handling Security Exceptions:** Ensure that security-related exceptions are appropriately logged and sanitized.
- **Scalability:** Design solutions to handle a range of inputs efficiently, considering future scalability requirements.

### Risk Assessment:

- **Risk Level:** Medium
- **Probability:** Probable
- **Impact:** High
- **Priority:** P4
- **Likelihood:** L3

## Do not encode noncharacter data as a string

guideline addresses the issue of converting noncharacter data, such as numeric values, to strings and the potential loss of data integrity associated with such conversions. Let's break down the provided code examples and compliant solutions.

### Noncompliant Code Example:

This code attempts to convert a `BigInteger` value to a `String` and then back to a `BigInteger`. However, the process involves converting the `BigInteger` value to a byte array using `toByteArray()` method, then creating a string from the byte array using the `String(byte[] bytes)` constructor, and finally converting the string back to a byte array and then to a `BigInteger`. This approach risks data integrity issues because not all byte arrays can be safely converted to strings and back.

```
BigInteger x = new BigInteger("530500452766");
byte[] byteArray = x.toByteArray();
String s = new String(byteArray); // Risk of data corruption
byteArray = s.getBytes();
x = new BigInteger(byteArray); // Unlikely to reproduce the original
value
```

## Compliant Solution:

Using `toString()` and `getBytes()`:

This compliant solution ensures data integrity by first converting the `BigInteger` object to a string using the `toString()` method, which generates valid character data. Then, it converts the string to a byte array and back to a `BigInteger`.

COPY 

```
BigInteger x = new BigInteger("530500452766");
String s = x.toString(); // Valid character data
byte[] byteArray = s.getBytes();
String ns = new String(byteArray);
x = new BigInteger(ns);
```

## Using Base64 Encoding:

Another compliant solution involves using Base64 encoding to safely convert the `BigInteger` value to a string and back without corrupting the data. Java 8 introduced the `java.util.Base64` class, providing encoders and decoders for the Base64 encoding scheme.

COPY 

```
BigInteger x = new BigInteger("530500452766");
byte[] byteArray = x.toByteArray();
String s = Base64.getEncoder().encodeToString(byteArray);
byteArray = Base64.getDecoder().decode(s);
x = new BigInteger(byteArray);
```



## Risk Assessment:

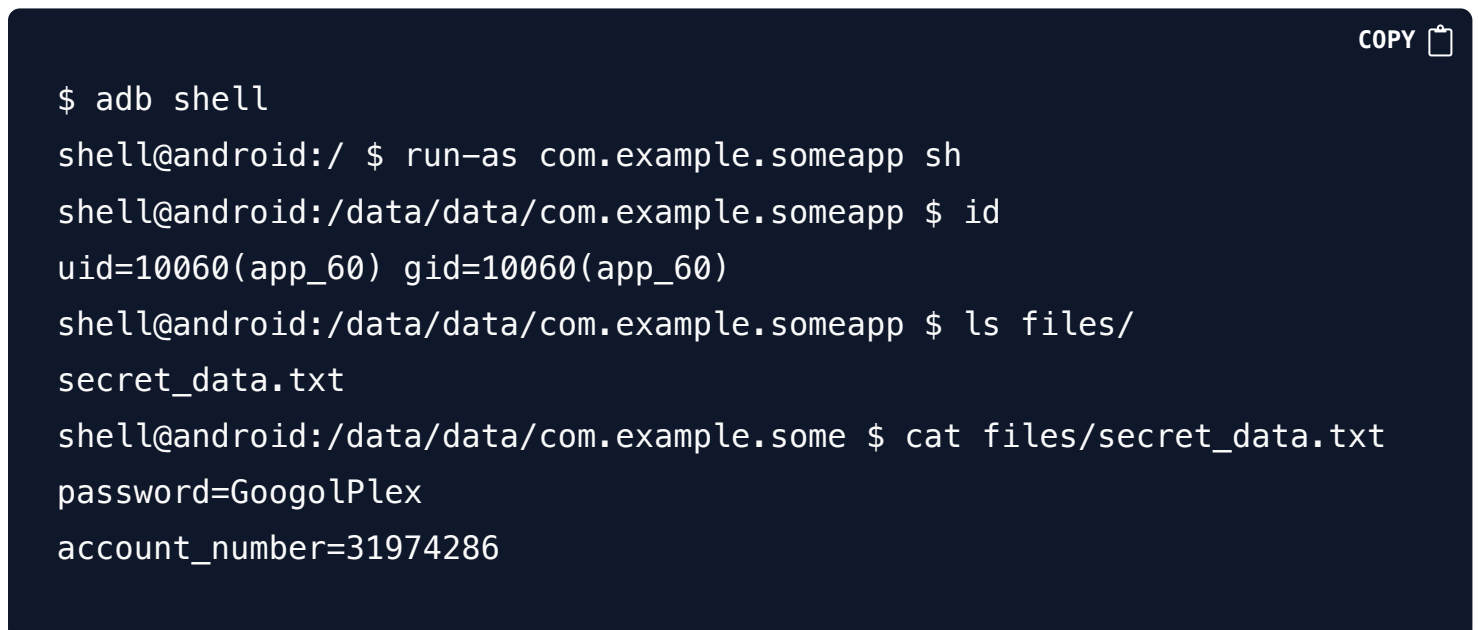
Encoding noncharacter data as a string can lead to a loss of data integrity. Therefore, it's crucial to use appropriate methods like `toString()` or encoding schemes like Base64 to ensure safe conversion without data corruption.

## Do not release apps that are debuggable

the `android:debuggable` attribute in Android app development, emphasizing the importance of ensuring that this attribute is set to false before releasing the app. Let's delve into the provided code examples and compliant solutions.

### Noncompliant Code Example:

This code example demonstrates an Android app with the `android:debuggable` attribute set to true, allowing access to sensitive data via debugging commands executed in the Android Debug Bridge (ADB) shell. Users can access the app's files and view sensitive information even without access to its source code.

A terminal window with a dark blue background and white text. In the top right corner, there is a 'COPY' button with a clipboard icon. The terminal shows a sequence of commands and their outputs. First, '\$ adb shell' is entered. Then, 'shell@android:/ \$ run-as com.example.someapp sh' is entered. Next, 'shell@android:/data/data/com.example.someapp \$ id' is entered, and the output is 'uid=10060(app\_60) gid=10060(app\_60)'. Then, 'shell@android:/data/data/com.example.someapp \$ ls files/' is entered, and the output is 'secret\_data.txt'. Finally, 'shell@android:/data/data/com.example.some \$ cat files/secret\_data.txt' is entered, and the output is 'password=GoogolPlex' and 'account\_number=31974286'.

```
$ adb shell
shell@android:/ $ run-as com.example.someapp sh
shell@android:/data/data/com.example.someapp $ id
uid=10060(app_60) gid=10060(app_60)
shell@android:/data/data/com.example.someapp $ ls files/
secret_data.txt
shell@android:/data/data/com.example.some $ cat files/secret_data.txt
password=GoogolPlex
account_number=31974286
```

With `android:debuggable` set to true, users can easily access sensitive data related to the app, posing a security risk.

### Compliant Solution:

To ensure app security, the `android:debuggable` attribute must be set to false before releasing the app. This prevents users from accessing sensitive information and

debugging the app without authorization.

COPY 

```
android:debuggable="false"
```

Some development environments automatically set `android:debuggable` to `true` for incremental or debugging builds but set it to `false` for release builds.

COPY 

```
<configuration>
  <compilation debug="true"/>
</configuration>
```

### Risk Assessment:

Releasing an app with `android:debuggable` set to `true` poses a high risk as it can leak sensitive information and make the app vulnerable to decompilation and alteration of its source code. Attackers can exploit this additional information to mount targeted attacks on the app's framework, database, or other resources.

## Consider privacy concerns when using Geolocation API

the Geolocation API, which allows web browsers to access the geographical location information of a user's device. It emphasizes obtaining the user's explicit permission before sending location information to websites to ensure privacy and security.

### Noncompliant Code Example:

The noncompliant code example overrides the `onGeolocationPermissionsShowPrompt()` method without presenting a user interface to ask for the user's permission. Instead, it unconditionally invokes the `callback` with the permission to access geolocation information, potentially leaking sensitive data without user consent.

```
public void onGeolocationPermissionsShowPrompt(String
origin, Callback callback){
    super.onGeolocationPermissionsShowPrompt(origin, callback);
    callback.invoke(origin, true, false);
}
```

### Compliant Solution #1:


This compliant solution presents a user interface to ask for the user's consent before transmitting geolocation data. Depending on the user's response, the application can control the transmission of geolocation information.

COPY 

```
public void onGeolocationPermissionsShowPrompt(String
origin, Callback callback) {
    super.onGeolocationPermissionsShowPrompt(origin, callback);
    // Ask for user's permission
    // When the user disallows, do not send the geolocation
information
}
```

### Compliant Solution #2:

In this compliant solution, the code checks a user setting to determine whether to prompt the user for permission to access geolocation information. If the setting is enabled, it shows a screen to ask for the user's permission. If the setting is disabled, it does not transmit the geolocation data.

COPY 

```
public void onGeolocationPermissionsShowPrompt(String
origin, GeolocationPermissions$Callback callback) {
    super.onGeolocationPermissionsShowPrompt(origin, callback);

    if(MyPreferences.getBoolean("SECURITY_ENABLE_GEOLOCATION_INFORMATION",
true)) {
        WebViewHolder.a(this.a).permissionShowPrompt(origin,
```

```
callback);  
    }  
    else {  
        callback.invoke(origin, false, false);  
    }  
}
```

### Risk Assessment:

Sending a user's geolocation information without obtaining the user's permission violates the security and privacy considerations of the Geolocation API, potentially leaking sensitive information. Therefore, it's crucial to implement proper user interface mechanisms to ask for consent before accessing and transmitting geolocation data.

## Properly verify server certificate on SSL/TLS

the importance of properly verifying server certificates when using SSL/TLS protocols for secure communication in Android applications. Failure to verify server certificates can lead to vulnerabilities where sensitive user data may leak through insecure SSL communication channels.

### Noncompliant Code Example:

The noncompliant code example demonstrates a custom `MySSLSocketFactory` class that extends `SSLSocketFactory` but fails to implement proper certificate verification. The `checkClientTrusted()` and `checkServerTrusted()` methods are overridden with blank implementations, effectively disabling SSL certificate verification. Additionally, the code sets the `sAllowAllSSL` flag to `true`, enabling the use of `ALLOW_ALL_HOSTNAME_VERIFIER`, which disables hostname verification.

COPY 

```
public class MySSLSocketFactory extends SSLSocketFactory {  
    SSLContext sslContext;  
  
    public MySSLSocketFactory(KeyStore truststore) throws  
        NoSuchAlgorithmException, KeyManagementException,  
        KeyStoreException, UnrecoverableKeyException {
```

```

        super(truststore);
        this.sslContext = SSLContext.getInstance("TLS");
        this.sslContext.init(null, new TrustManager[] {new
X509TrustManager() {
            public void checkClientTrusted(X509Certificate[] chain,
String authType) throws CertificateException {}
            public void checkServerTrusted(X509Certificate[] chain,
String authType) throws CertificateException {}
            public X509Certificate[] getAcceptedIssuers() { return
null; }
        }}, null);
    }

    public Socket createSocket() throws IOException {
        return this.sslContext.getSocketFactory().createSocket();
    }

    public Socket createSocket(Socket socket, String host, int port,
boolean autoClose)
        throws IOException, UnknownHostException {
        return this.sslContext.getSocketFactory().createSocket(socket,
host, port, autoClose);
    }
}

public static HttpClient getNewHttpClient() {
    DefaultHttpClient httpClient;
    try {
        KeyStore trustStore =
KeyStore.getInstance(KeyStore.getDefaultType());
        trustStore.load(null, null);
        MySSLSocketFactory mySSLSocketFactory = new
MySSLSocketFactory(trustStore);
        if(DefineRelease.sAllowAllSSL) {

((SSLSocketFactory)mySSLSocketFactory).setHostnameVerifier(SSLSocketFa
ctory.ALLOW_ALL_HOSTNAME_VERIFIER);
        }
        // Rest of the code
    }
}

```

```
    catch(Exception e) {  
        // Handle exception  
    }  
    return httpClient;  
}
```

### Compliant Solution:

The compliant solution would involve implementing proper SSL certificate verification mechanisms. Depending on the specific implementation requirements, this could include:

- Ensuring that `checkClientTrusted()` and `checkServerTrusted()` methods perform appropriate certificate validation.
- Enabling hostname verification to match the server's certificate with the intended hostname.
- Avoiding the use of `ALLOW_ALL_HOSTNAME_VERIFIER` to prevent bypassing hostname verification.

### Risk Assessment:

Failure to properly verify server certificates in SSL/TLS communication can lead to significant security risks, including man-in-the-middle attacks where an attacker intercepts and manipulates the communication between the client and the server. This can result in the exposure of sensitive user data, undermining the confidentiality and integrity of the application's communication channels.

## Specify permissions when creating files via the NDK

file permissions when creating files in Android applications, especially when using native code. Improper file permissions can lead to security vulnerabilities, such as exposing sensitive data or allowing unauthorized modification of files.

### Noncompliant Code Example:

The noncompliant code example demonstrates creating a text file using native C code without explicitly setting file permissions. This results in a file that is both world-readable and world-writable, potentially exposing it to unauthorized access or modification.

COPY 

```
FILE *fp = fopen("/data/data/com.mine.work/file.txt", "a");
fprintf(fp, "Don't alter this content.\n");
fclose(fp);
```

### Compliant Solution (Set Umask):

In this compliant solution, the user explicitly sets the process's umask to ensure that the created file's permissions match those of the Android SDK. By using the `umask()` C library call, the file permissions are restricted to prevent world-writable access

COPY 

```
umask(002);
FILE *fp = fopen("/data/data/com.mine.work/file.txt", "a");
fprintf(fp, "Don't corrupt this content.\n");
fclose(fp);
```

### Compliant Solution (Specify File Permissions):

Another compliant solution involves explicitly specifying the file permissions using the `open()` system call. By setting appropriate permissions using the `S_IRUSR`, `S_IWUSR`, `S_IRGRP`, and `S_IWGRP` flags, the file's access is restricted to user and group, preventing world access.

COPY 

```
const char *fn = "/data/data/com.mine.work/file.txt";
const char *content = "Don't corrupt this content.\n";
fd = open(fn, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP |
S_IWGRP);
```

```
err = write(fd, content, strlen(content));  
close(fd);
```

### Risk Assessment:

Failure to properly set file permissions when creating files in native code may result in files being accessible or modifiable by unauthorized users or applications. This can lead to the exposure or corruption of sensitive data, posing significant security risks to the application and its users.

### Sensitive classes must not let themselves be copied

importance of preventing the copying of classes containing private, confidential, or sensitive data. Failing to define proper copy mechanisms, such as a copy constructor, can lead to security vulnerabilities, including unauthorized data access or modification.

### Noncompliant Code Example:

In the noncompliant code example, a class `SensitiveClass` is defined, containing a character array for storing a file name and a Boolean variable for managing shared access. However, the class lacks a copy constructor, allowing potential vulnerabilities if the class is copied improperly.

COPY 

```
class SensitiveClass {  
    private char[] filename;  
    private Boolean shared = false;  
  
    SensitiveClass(String filename) {  
        this.filename = filename.toCharArray();  
    }  
  
    final void replace() {  
        if (!shared) {  
            for(int i = 0; i < filename.length; i++) {  
                filename[i]= 'x' ;}  
        }  
    }  
}
```



```

    }

    final String get() {
        if (!shared) {
            shared = true;
            return String.valueOf(filename);
        } else {
            throw new IllegalStateException("Failed to get instance");
        }
    }

    final void printFilename() {
        System.out.println(String.valueOf(filename));
    }
}

```

### Malicious Subclass:

A malicious subclass `MaliciousSubclass` is created, which extends `SensitiveClass` and overrides the `clone()` method. This subclass allows unauthorized access and modification of the sensitive data.

```

class MaliciousSubclass extends SensitiveClass implements Cloneable {
    protected MaliciousSubclass(String filename) {
        super(filename);
    }

    @Override public MaliciousSubclass clone() {
        MaliciousSubclass s = null;
        try {
            s = (MaliciousSubclass)super.clone();
        } catch (Exception e) {
            System.out.println("not cloneable");
        }
        return s;
    }
}

```

COPY 

```
// main method demonstrates exploitation  
}
```

### Compliant Solution (Final Class):

Declare `SensitiveClass` as `final` to prevent subclassing and ensure that the class cannot be copied.

```
final class SensitiveClass {  
    // ...  
}
```

COPY 

### Compliant Solution (Final clone()):

Prevent the cloning of instances by providing a `final` implementation of the `clone()` method that always throws a `CloneNotSupportedException`.

```
class SensitiveClass {  
    // ...  
    public final SensitiveClass clone() throws  
    CloneNotSupportedException {  
        throw new CloneNotSupportedException();  
    }  
}
```

COPY 

## Risk Assessment:

Failure to prevent the copying of sensitive classes can result in unauthorized data access or modification, leading to security vulnerabilities. Implementing proper copy mechanisms or making classes noncopyable mitigates these risks and ensures the integrity of sensitive data.

## References

- <https://wiki.sei.cmu.edu/confluence/display/android>
- Android Application Secure Design/Secure Coding Guidebook by Japan Smartphone Security Association (JSSEC)

Devops

DevSecOps

Android

Mobile Development

## MORE ARTICLES

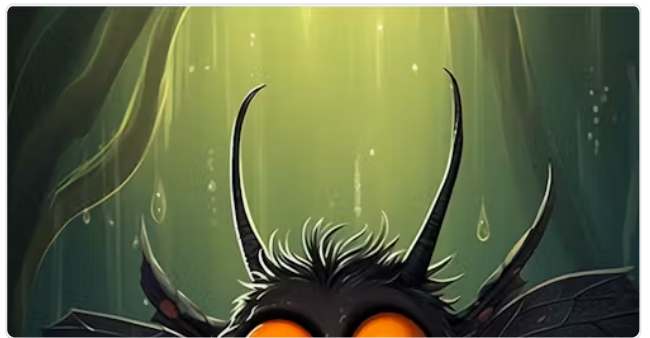
**RR** Reza Rashidi



### Attacking IOS

In this comprehensive guide, we delve into the world of iOS security from an offensive perspective, ...

**RR** Reza Rashidi



### Defending APIs

we embark on a journey to fortify our APIs against common vulnerabilities that lurk at every stage o...

**RR** Reza Rashidi



### Attacking APIs

APIs (Application Programming Interfaces) have become integral

components of modern software  
systems...