Google Play Protect

Android app vulnerability classes

A whirlwind overview of common security and privacy problems in Android apps

Introduction

How Google Play Protect educates developers to make millions of apps safer to use

Content of the presentation

- Overview of common Android app vulnerabilities reported through the Google Play Security Rewards Program
- Explicitly not an attempt at creating a complete audit guide
  - Focused only to vulnerabilities in scope for our bug bounty
- For each vulnerability present
  - Overview
  - Auditing tips
  - Remediation tips
  - CWE ID (Common Weakness Enumeration) and other resources

How to use this presentation

- You are an app developer:
  - Understand which severe vulnerabilities are common even in top apps by top developers
  - Learn how to find more information about how to identify and fix these vulnerabilities
- You are a security researcher:
  - Understand what common vulnerabilities are worth looking into
  - Learn how to find these vulnerabilities to earn your own bug bounties

Android app vulnerabilities

19 vulnerability classes across 5 themes

Insecure connections

Use of insecure network protocols

The problem: Loading data over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to replace, remove, and inject code.

Example code

```
URL url = new URL("http://www.google.com");
HttpURLConnection urlConnection =
  (HttpURLConnection) url.openConnection();
urlConnection.connect();
InputStream in = urlConnection.getInputStream();
```

See also: CWE-319 (Cleartext Transmission of Sensitive Information)

Use of insecure network protocols

The problem: Loading data over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to replace, remove, and inject code.

Auditing tips

- Review the app’s network security configuration (an Android P feature) to understand whether app is allowed to use insecure network protocols in the first place.
    - https://developer.android.com/training/articles/security-config (“Network security configuration”)
- Grep the app’s code for “http:”, “ftp”, “smtp:”, and URLs that indicate use of insecure network protocols.
- Understand common entry points into the network such as the URL class or the WebView class and check whether there are code flows that lead to insecure connections being made.

Use of insecure network protocols

The problem: Loading data over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to replace, remove, and inject code.

Remediation

- Apps targeting Android P and above are safe by default. If such apps still want to make insecure connections, they have to define a network security configuration that allows that.
- Use `javax.net.ssl.HttpsURLConnection` as much as possible.
- Presubmit checks to look for URLs for insecure protocols are easy to set up.
- The Android Lint check `InsecureBaseConfiguration` flags insecure HTTP connections.

Data transmission over insecure protocols

The problem: Sending sensitive data over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to intercept, modify, and steal the data.

Example code

```
URL url = new URL("http://www.mysite.com?lookupNumber=" +
phoneNumber);
HttpURLConnection urlConnection =
  (HttpURLConnection) url.openConnection();
urlConnection.connect();
InputStream in = urlConnection.getInputStream();
```

See also: CWE-319 (Cleartext Transmission of Sensitive Information)

Data transmission over insecure protocols

The problem: Sending sensitive data over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to intercept, modify, and steal the data.

Auditing tips

- In addition to the auditing tips for finding the use of insecure network protocols:
  - Understand which user data and application data is potentially sensitive
  - Audit whether any network connections over insecure protocols send potentially sensitive data to network endpoints

Data transmission over insecure protocols

The problem: Sending sensitive data over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to intercept, modify, and steal the data.

Remediation

- Exactly the same as for the use of insecure network connections

Authentication over insecure protocols

The problem: Sending login information over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to intercept, modify, and steal the data.

Example code

```
URL url = new
URL("http://www.mysite.com?user=bugdroid&pwd=androidnumber
one");
HttpURLConnection urlConnection =
  (HttpURLConnection) url.openConnection();
urlConnection.connect();
InputStream in = urlConnection.getInputStream();
```

See also: CWE-319 (Cleartext Transmission of Sensitive Information)

Authentication over insecure protocols

The problem: Sending login information over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to intercept, modify, and steal the data.

Auditing tips

- In addition to the auditing tips for finding the use of insecure network protocols:
  - Understand which login information the app is working with
  - Audit whether any network connections over insecure protocols send login information to network endpoints

Authentication over insecure protocols

The problem: Sending login information over insecure networks such as HTTP allows attackers that control the network (for example a local WiFi network) to intercept, modify, and steal the data.

Remediation

- In general the same as for the use of insecure network connections.
- Authentication should probably be handled through a common, security-vetted method such as OAuth. Custom authentication schemes are likely to get more wrong than just insecure network protocols.

Cryptography and authentication

Embedded third-party secrets

The problem: Apps that embed third-party secrets such as Twitter API keys or AWS authentication tokens can trivially have these secrets extracted and abused by attackers.

Example code

```
ParseTwitterUtils.initialize(
  "CONSUMER KEY", "CONSUMER SECRET");
```

See also: CWE-798 (Use of Hard-coded Credentials)

Embedded third-party secrets

The problem: Apps that embed third-party secrets such as Twitter API keys or AWS authentication tokens can trivially have these secrets extracted and abused by attackers.

Auditing tips

- Compile a list of interesting public APIs and create regular expressions to find keys or secrets through grep.
- Compile a list of interesting public APIs and look for their package names in apps.
- Grep for “key”, “password”, “login”, “secret” and such.

Embedded third-party secrets

The problem: Apps that embed third-party secrets such as Twitter API keys or AWS authentication tokens can trivially have these secrets extracted and abused by attackers.

Remediation

- Follow best practices for the used API as documented in their help center.
  - Some APIs encourage you to embed their secrets in the app as a secret compromise will have limited negative effects.
- Some third-party services offer more secure alternatives to embedding credentials in the app (such as Amazon and Google) while others like Twitter consider embedding the third-party secrets in the code as best practice.
  - For services which don't provide a per-user authentication service, it's possible to spin up your own server that handles this.
- For services that recommend embedding third-party secrets in the app, expect the secrets to be extracted and abused by attackers. The goal for such situations is to mitigate the negative effects of the abuse.

Embedded cryptography secrets

The problem: Applications that use embedded crypto secrets are susceptible to simple data decryption attacks.

Example code

```
private static String SECRET_KEY = "MySecretAESKey99";

private static byte[] encrypt(String inputText) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE,
        new SecretKeySpec(SECRET_KEY.getBytes(), "AES"));

    return cipher.doFinal(inputText.getBytes("UTF-8"));
}
```

See also: CWE-321 (Use of Hard-coded Cryptographic Key)

Embedded cryptography secrets

The problem: Applications that use embedded crypto secrets are susceptible to simple data decryption attacks.

Auditing tips

- Look for all uses of APIs in the javax.crypto package
- Look for all embedded popular cryptography libraries
- Audit initialization and use of discovered cryptography APIs

Embedded cryptography secrets

The problem: Applications that use embedded crypto secrets are susceptible to simple data decryption attacks.

Remediation

- Understand what you are trying to protect from whom
    - In many cases using embedded secrets is actually fine
- Avoid rolling your own crypto by abstracting what you're trying to do and using a ready-made library for your purpose

Leaking OAuth tokens

The problem: OAuth Implicit Grant and Authz (without PKCE) flows expose tokens that can be used to create fraudulent requests.

Example code

```
String url = "https://accounts.google.com/o/oauth2/v2/auth";
url += "?client_id=XXXX";
url += "&redirect_url=com%2Emy%2Eapp%3A%2F%2Foauth";
url += "&response_type=token";  // Use Implicit Grant =(
url += "&scope=email";
Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse(url));
startActivity(i);
```

See also: CWE-??? (???)

Leaking OAuth tokens

The problem: OAuth Implicit Grant and Authz (without PKCE) flows expose tokens that can be used to create fraudulent requests.

Auditing tips

- Search for OAuth endpoint URLs being sent to the browser and check what parameters are being passed
    - Implicit grant is always bad
    - If using Authz flow, check for PKCE use
    - If using PKCE, check that it is not predictable

Leaking OAuth tokens

The problem: OAuth Implicit Grant and Authz (without PKCE) flows expose tokens that can be used to create fraudulent requests.

Remediation

- Use a platform-integrated authentication flow rather than the browser.
  - Much more foolproof.
- If using OAuth in the browser, use the Authz flow with PKCE
  - Make sure you aren't using a hardcoded secret or nonce

Private file access

Private data sharing

The problem: Applications may leak private data to other apps or attackers in obvious and subtle ways.

Example code

```
public class FileUploader extends Activity {

    protected void onCreate(Bundle savedInstanceState) {
      String path = getIntent().getExtras().getString(“path”);
      String url = getIntent().getExtras().getString(“url”);

      uploadFile(path, url);
    }
}
```

See also: CWE-359 (Exposure of Private Information ('Privacy Violation'))

Private data sharing

The problem: Applications may leak private data to other apps or attackers in obvious and subtle ways.

Auditing tips

- Look for files stored on external storage which is world-readable by default.
- Look for files stored in internal storage whose access attributes are changed to make them world-accessible.
- Look for unsanitized file or path names that are supplied by potential attackers.

Private data sharing

The problem: Applications may leak private data to other apps or attackers in obvious and subtle ways.

Remediation

- Store files in internal storage with default access attribute unless you want to intentionally share them with other apps.
- Limit the ability of users to supply their own path or file names.
- Verify that the canonical path of a file or path points to an expected path.
    - Watch out for TOCTOU issues. Symlinks can change at any time.

Private data overwrite due to path traversal

The problem: Applications that don't sanitize attacker-provided directory paths may be susceptible to overwrite their internal files with attacker-provided files.

Example code

```
OutputStream os =
    new FileOutputStream(PUBLIC_DIRECTORY + "/Download/" + filename);
```

See also: CWE Category 21 (Pathname Traversal and Equivalence Errors)

Private data overwrite due to path traversal

The problem: Applications that don’t sanitize attacker-provided directory paths may be susceptible to overwrite their internal files with attacker-provided files.

Auditing tips

- Find any kind of path or file operations with variable input and see if they rely on user-provided input.

Private data overwrite due to path traversal

The problem: Applications that don’t sanitize attacker-provided directory paths may be susceptible to overwrite their internal files with attacker-provided files.

Remediation

- Limit the ability of users to supply their own path or file names.
- Verify that the canonical path of a file or path points to an expected path right before doing file operations on the file or path.

Private data overwrite due to ZIP file traversal

The problem: Applications that unzip archive files without sanitizing the target file paths of files inside the archives are susceptible to overwriting their internal files with attacker-provided files.

Example code

```
InputStream is = new FileInputStream(zipFilePath);
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(is));
ZipEntry ze;

while ((ze = zis.getNextEntry()) != null) {
    outputFileName = ze.getName();
    writeToFile(outputFileName, zis);
}
```

See also: CWE ??? (???)

Private data overwrite due to ZIP file traversal

The problem: Applications that unzip archive files without sanitizing the target file paths of files inside the archives are susceptible to overwriting their internal files with attacker-provided files.

Auditing tips

- Find all uses of APIs in the java.util.zip.* package or third-party zipping libraries.
- Understand whether user-provided ZIP files are being unzipped by the code.

Private data overwrite due to ZIP file traversal

The problem: Applications that unzip archive files without sanitizing the target file paths of files inside the archives are susceptible to overwriting their internal files with attacker-provided files.

Remediation

- Validate that the canonical path of unzipped files points to the real directory to unzip to.
  - A Google search for “android zip path traversal” will lead to examples.

Unprotected app parts

Unprotected activities

The problem: Exported activities can be started by other apps on the device which may break user workflow assumptions, including ways that break security boundaries.

Example code

```
<activity android:name="com.mypackage.SomeActivity">
...
  <intent-filter>
    <action android:name="com.mypackage.OPEN" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
...
</activity>
```

See also: CWE-926 (Improper Export of Android Application Components)

Unprotected activities

The problem: Exported activities can be started by other apps on the device which may break user workflow assumptions, including ways that break security boundaries.

Auditing tips

- Understand what conditions can lead to activities being exported
  - android:exported attribute
  - Presence of intent filters auto-exports activities
- Go through list of activities declared in manifest XML file
  - Find those that are exported

Unprotected activities

The problem: Exported activities can be started by other apps on the device which may break user workflow assumptions, including ways that break security boundaries.

Remediation

- Explicitly mark components with android:exported="false" in the application manifest
- Use android:protectionLevel="signature" in the xml manifest to restrict access to applications signed by you

Unprotected services

The problem: Applications that export services allow malicious apps on the device to start those services.

Example code

```
<service android:name="com.myapp.service">
...
  <intent-filter>
    <action android:name="com.myapp.START_SERVICE" />
  </intent-filter>
...
</service>
```

See also: CWE-926 (Improper Export of Android Application Components)

Unprotected services

The problem: Applications that export services allow malicious apps on the device to start those services.

Auditing tips

- Same as for unprotected activities

Unprotected services

The problem: Applications that export services allow malicious apps on the device to start those services.

Remediation

- Same as for unprotected activities

Typos in custom permissions

The problem: When used custom permissions don’t match declared custom permissions, Android defaults to silently failing to enforce the permission.

Example code

```
<permission
android:label="@string/write_contact_permission"
android:name="my.app.permission.write" />

<provider android:label="@string/contacts"
android:name="my.app.ContactsProvider"
android:writePermission="my.app.permissions.write"/>
```

See also: CWE-??? (???)

Typos in custom permissions

The problem: When used custom permissions don't match declared custom permissions, Android defaults to silently failing to enforce the permission.

Auditing tips

- For stand-alone applications, check that declared and used permissions match in the manifest file.
    - For sets of apps, check that declared and used permissions match across the whole set of apps.

Typos in custom permissions

The problem: When used custom permissions don't match declared custom permissions, Android defaults to silently failing to enforce the permission.

Remediation

- If you are using custom permissions you can set up a presubmit check to see whether any declared permissions are not used or any used permissions are undeclared.
    - Doesn't quite work if you have multiple interacting apps where not all apps declare and use the same permissions. Maybe a whitelist in the presubmit check is possible.

Intent redirection

The problem: Apps that accept and launch arbitrary intents from external sources may allow malware to start internal components indirectly or access protected content:// URIs.

Example code

```
protected void onHandleIntent(Intent intent) {
  if (intent != null) {
    Intent privateIntent = new Intent(this, parseIntentClass(intent));
    startActivity(privateIntent);
  }
}
```

See also: CWE-925 (Improper Verification of Intent by Broadcast Receiver)

Intent redirection

The problem: Apps that accept and launch arbitrary intents from external sources may allow malware to start internal components indirectly or access protected content:// URIs.

Auditing tips

- Find calls to startActivity and verify that Intent components are constructed from trusted data.
- Find calls Intent::getExtras where returned values are cast to Intent and verify that they are properly checked before being used.
    - It isn't enough to check the target class name. Malware can reuse your class names and force your app to send an Intent that grants content:// URI access to their app.

Intent redirection

The problem: Apps that accept and launch arbitrary intents from external sources may allow malware to start internal components indirectly or access protected content:// URIs.

Remediation

- Whenever sending an Intent created from untrusted data
  - Check the class name **and** package name of the target component or...
  - Check the signing key of the app that owns the target component
  - Verify that Intents don't contain FLAG_GRANT_READ_URI_PERMISSION or FLAG_GRANT_WRITE_URI_PERMISSION

Implicit broadcasts (sending)

The problem: Applications that send broadcasts without specifying the broadcast target may have these broadcasts intercepted by malicious apps on the same device.

Example code

```
android.content.Intent intent = new android.content.Intent();
intent.putExtra("SMS", smsMessage);
intent.setAction("my.app.sms");
~~intent.setClass(context, my.app.SmsHandler);~~
sendBroadcast(intent);
```

See also: CWE-927 (Use of Implicit Intent for Sensitive Communication)

Implicit broadcasts (sending)

The problem: Applications that send broadcasts without specifying the broadcast target may have these broadcasts intercepted by malicious apps on the same device.

Auditing tips

- Look for all intent broadcasts that do not have a target set
    - Called implicit broadcasts
    - Lacks `setComponent`, `setClass`, `setClassName` or an explicit constructor

Implicit broadcasts (sending)

The problem: Applications that send broadcasts without specifying the broadcast target may have these broadcasts intercepted by malicious apps on the same device.

Remediation

- Turn as many implicit intents into explicit intents.
- For the intents that cannot be made explicit understand that they can be intercepted
  - Think through the worst case scenarios.
  - Document why these intents are implicit.

Implicit broadcasts (receiving)

The problem: Applications that accept broadcasts without checking the sender may accept maliciously crafted broadcasts sent from malicious apps on the same device.

Example code

```
final class Receiver extends BroadcastReceiver {
    public final void onReceive(Context ctx, Intent intent) {
        if (verifyPackageName(getCallingActivity())) {
            doSomethingSecret();
        }
    }
}
```

Don’t actually know what verifyPackageName would look like.

See also: CWE-925 (Improper Verification of Intent by Broadcast Receiver)

Google Play Protect

Google Play

Implicit broadcasts (receiving)

The problem: Applications that accept broadcasts without checking the sender may accept maliciously crafted broadcasts sent from malicious apps on the same device.

Auditing tips

- Find all receivers in the app
  - Look at manifest file and for `Context.registerReceiver` calls.
  - Non-exported receivers are safe as they can only receive broadcasts from inside the app.

Implicit broadcasts (receiving)

The problem: Applications that accept broadcasts without checking the sender may accept maliciously crafted broadcasts sent from malicious apps on the same device.

Remediation

- Make as many receivers as possible non-exported.
- For receivers that have to stay exported, treat all inputs as untrusted.

Other

Everything else that doesn’t fit anywhere

Incorrect URL verification

The problem: Apps that rely on URL parsing to verify that a given URL is pointing to a trusted server may be susceptible to many different ways to get URL parsing and verification wrong.

Example code

```
// Just one of many ways to get URL verification wrong

if (uri.getHost().endsWith("mywebsite.com")) {
     webView.loadUrl(uri.toString());
}
```

See also: CWE-939 (Improper Authorization in Handler for Custom URL Scheme)

Incorrect URL verification

The problem: Apps that rely on URL parsing to verify that a given URL is pointing to a trusted server may be susceptible to many different ways to get URL parsing and verification wrong.

Auditing tips

- Find every instance of URLs being used in an app
- Narrow it down to those that work on user-controlled input
- Narrow it down to those that are used to branch between different code paths
  - Could be any part of the URL, not just the host name

Incorrect URL verification

The problem: Apps that rely on URL parsing to verify that a given URL is pointing to a trusted server may be susceptible to many different ways to get URL parsing and verification wrong.

Remediation

- Read and understand https://hackerone.com/reports/431002
    - "Golden techniques to bypass host validations in Android apps"
- Create a library that follows best practices and always use that when trying to verify URLs

Cross-app scripting

The problem: Apps that load untrusted URLs from other apps into their WebViews that match a certain form (e.g., javascript: or file:///path/to/private) allow malicious JavaScript code execution.

Example code

```
Intent intent = getIntent();
String url = intent.getStringExtra(“url”);
WebView myWebview = (WebView) findViewById(R.id.foo);

// url points to the cookies database, is a javascript:
// url, or is a phishing url
myWebview.loadUrl(url);
```

See also: CWE-925 (Improper Verification of Intent by Broadcast Receiver)

Cross-app scripting

The problem: Apps that load untrusted URLs from other apps into their WebViews that match a certain form (e.g., javascript: or file:///path/to/private) allow malicious JavaScript code execution.

Auditing tips

- Find calls to loadUrl and evaluateJavascript in your app.
- Verify that inputs to these functions are trusted or sanitized (or both).
- Check for dangerous WebView settings like setAllowFileAccess(true)
  - Files can contain *both* malicious scripts and sensitive user data (the cookies database).

Cross-app scripting

The problem: Apps that load untrusted URLs from other apps into their WebViews that match a certain form (e.g., javascript: or file:///path/to/private) allow malicious JavaScript code execution.

Remediation

- Disable WebView file access unless it is needed.
  - Use setAllowFileAccess(false)
- Target up-to-date SDK versions to prevent javascript: URLs
- Make sure that app-private Activities aren't exported by accident
  - Creating an intent-filter exports the component by default!
- Verify that URLs from external sources are safe to load before calling loadUrl()
  - URL validation can be tricky as you just learned.
  - Think carefully about what URLs are acceptable. Same-origin-policy does not protect against phishing.

Incorrect sandboxing of scripting language

The problem: Embedding a scripting language or interpreter in an app can lead to exposure of app internals if the security boundaries of the interpreter are not well understood.

Example code

```
PythonInterpreter interpreter = new PythonInterpreter();
interpreter.exec(
    "from shutil import copyfile\n
     copyfile(\"passwords.bin\", \"/sdcard/passwords.bin\"
);
```

See also: CWE-266 (Incorrect Privilege Assignment)

Incorrect sandboxing of scripting language

The problem: Embedding a scripting language or interpreter in an app can lead to exposure of app internals if the security boundaries of the interpreter are not well understood.

Auditing tips

- Look for indicators of scripting languages
  - Common interpreters such as Python inside the app.
  - Interpreters may be hidden in file formats like PDF (Javascript).
  - Look for Android and Java sandboxing libraries.
- Check whether these interpreters allow untrusted input.

Incorrect sandboxing of scripting language

The problem: Embedding a scripting language or interpreter in an app can lead to exposure of app internals if the security boundaries of the interpreter are not well understood.

Remediation

- Disable user-supplied input if possible.
- Build your own limited-scope interpreter or interface to more permissive interpreters.
- Otherwise sanitize scripts, disable scripting, or restrict permissions and abilities of interpreter engine.

Unprotected data on server

The problem: An app connects to a remote web server, database, or API that does not sufficiently protect sensitive user data. Any attacker can access this data.

Example code

```
Not applicable since the problem is not in the application
code.
```

See also: CWE-306 (Missing Authentication for Critical Function)

Unprotected data on server

The problem: An app connects to a remote web server, database, or API that does not sufficiently protect sensitive user data. Any attacker can access this data.

Auditing tips

- Find all instances of an app connecting out to a remote server, either in the main app or through SDKs.
- Understand default and custom configuration options for remote endpoints.

Bug Bounty through the Google Play Security Reward Program: $0 (not in scope)

Unprotected data on server

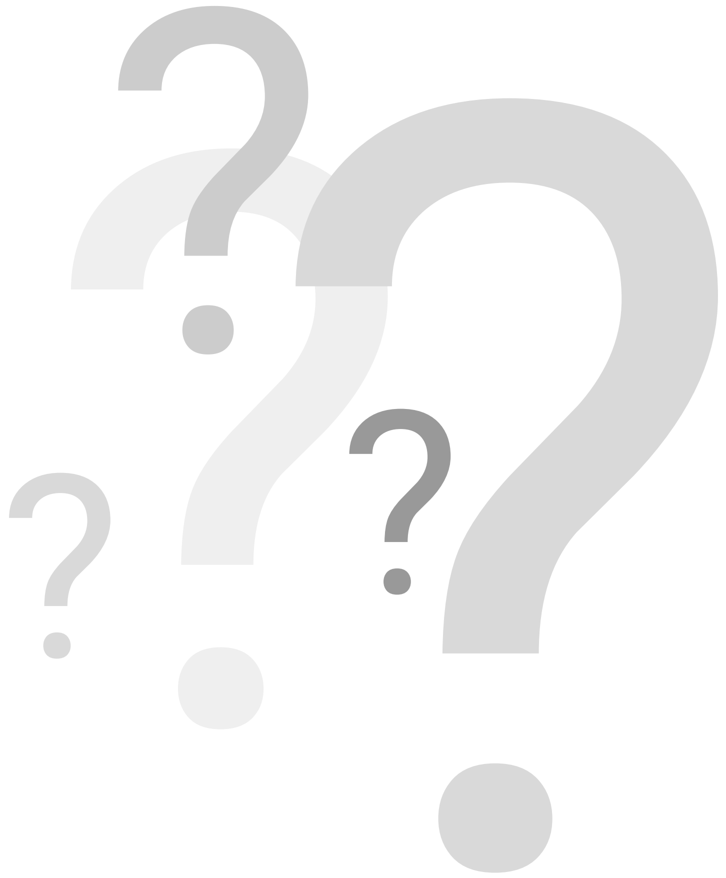
The problem: An app connects to a remote web server, database, or API that does not sufficiently protect sensitive user data. Any attacker can access this data.

Remediation

- Follow best practices for your particular web host or API provider.

Q&A

Questions?
Want to learn more about the Google Play Security Reward Program?
Visit:
https://www.google.com/about/appsecurity/play-rewards/.



THANK YOU