CISPA Summer School 2018
Attacking Android Apps

28 August 2018

1 Security Analysis of Chipper App

Please consult Appendix A for an introduction to the Chipper App and Service.
If you are not familiar with Android app programming, Appendix B explains the basics steps for setting up the attack app, create emulators, and running apps on them. Appendix C illustrates how Android apps can be decompiled and reverse engineered which is necessary to get you going with this exercise.

In this workshop, you will conduct a security analysis of the Chipper app to discover different vulnerabilities and develop proof-of-concept attack code. Each of the vulnerabilities of the Chipper app allows the attacker to either

- steal the user's credentials (username & password)
- steal the user's private messages
- send messages on the user's behalf, or
- a combination of those above

Attacking the Chipper App

The Chipper app has, in addition to the user's visible workflow, functions that prepare a future extension of the Chipper app with plugins (either in form of other apps or HTML code loaded). These plugin features together with the user-visible functions described in Appendix **??** have *seven confirmed vulnerabilities*. In the following, we will guide you through the confirmed vulnerabilities and give you hints on where to find them in the Chipper app and how to exploit them with your attacker app.

The vulnerabilities allow for:

1. JavaScript injection into the web content of the Chipper server
2. Local cross origin attacks from web content against the Chipper app (and other installed apps)
3. Privilege escalation for unauthorized Chipper plugin apps
4. File-system leaks of confidential information to other installed apps

If you are already familiar with Android app security, maybe try to figure them out without the help provided on the following pages ☺

For every vulnerability, we provide different levels of hints, so you can challenge yourself if you want:

- (**A**) Where in the code of the Chipper app the vulnerability is located
- (**B**) What the effect or cause of the vulnerability is
- (**C**) How it can be exploited

JavaScript Injection

There are two possibilities to inject JavaScript into the Chipper app. One is web-based through the server and another is through a parallel installed attack app on the same device.

Hints:

- (**A**) Search for the usage of the method *addJavascriptInterface(...)* through all app's classes and check the corresponding documentation[1] to understand how the methods of the javascript interface can be invoked.

  (**B**) Given that all posts are rendered in the WebView, a specially crafted post message could invoke one of methods defined in the javascript interface causing user's credentials (i.e., username and password) to be leaked.

  (**C**) One way to exploit this vulnerability is to post a message to the Chipper's server with a javascript content (e.g., *<script>some malicious js content</script>*) which, when viewed (i.e., rendered by the WebView) by other Chipper users, will invoke the vulnerable method of the javascript interface which will invoke the PluginService with a malicious payload that leaks user's credentials. To actually leak the credentials, you need to build an app (or extend the supplied attacker app) that listens on a specific channel and receives the credentials. This means, attacker's app should be installed on the same device as Chipper's to complete the attack.

- (**A**) Content providers enable app developers to store key-object tuples and, if permissions allow, share them between apps. The Chipper app uses this facility to cache the HTML content of the posts in a ContentProvider, from where it loads the HTML content in case the network isn't available. It uses a signature-level permission[2] to regulate access to the ContentProvider, such that only apps that are signed by the same key used to sign the Chipper app can access it. A good starting point to discover the vulnerability is the *AndroidManifest.xml* file. Also, check the implementation of the ContentResolver that is used to query entries of the ContentProvider.

  (**B**) This vulnerability can be used to inject a persistent malicious payload into the ContentProvider used for caching the posts. As a result, whenever the app is not connected to the internet, it loads posts from the cache triggering the attack. This vulnerability has a similar effect to the one described above (i.e., leak user's credentials).

  (**C**) Check the definition of the *<provider>* component in the *AndroidManifest.xml* file and understand the purpose of each used attribute[3]. There you will find two URIs for data repositories (see *android:pathPrefix* attribute) protected by a signature-level permission. However, after checking the functionality used for loading cached posts when there is no internet connection, you will find something wrong. To exploit this vulnerability, you need to build an app (or extend the supplied attacker's app) and implement a ContentResolver that injects a malicious payload into the ContentProvider of the Chipper app which, when rendered in the WebView, should leak user's credentials to the attacker's app.

Cross origin attack

A local cross-origin attack exists via specifically crafted links in messages. In a cross-origin attack, malicious web content manages to trigger local communication (e.g., Intent) from the app with a vulnerable WebView to other components of the same or another app (i.e., the Intent crosses the origin from the web to locally installed apps).

Hints:

(**A**) The Chipper app allows users to attach one link to each message which are then rendered together in the WebView. However, since not all links can be trusted, Android's framework enables developers to decided on the links that can be handled/loaded in the WebView by overriding the *shouldOverrideUrlLoading* API[4]. As a starting point to discover the vulnerability, check the implementation of the *shouldOverrideUrlLoading* API and think of a way to abuse it.

[1] https://developer.android.com/reference/android/webkit/WebView.html
[2] https://developer.android.com/guide/topics/manifest/permission-element#plevel
[3] https://developer.android.com/guide/topics/manifest/provider-element
[4] https://developer.android.com/reference/android/webkit/WebViewClient.html

(**B**) This vulnerability can be used to invoke privileged functionalities of the PluginService. For example, the attacker can leak user's credentials and post messages on behalf of other users who click on her malicious links.

(**C**) The attacker has to construct an URI and post it with the message to the Chipper app. When this URI is clicked by other users, the WebView handles the action and invokes the PluginService using Intents. We suggest that the malicious URI should post messages on behalf of anyone clicking on it. Check the method responsible for posting messages in the PluginService to extract the required parameters.

Privilege escalation for unauthorized plugin apps

The plugin functionality has an erroneous authentication allowing an attack app to gain protected privileges for the plugin functionality.

Further, an attack app can also use another plugin functionality with erroneous authorization to eavesdrop on sent messages. Both vulnerabilities originate from mistakenly handling package names as token of authority.

Hints:

- (**A**) Apps can request *PendingIntent*s for the PluginService[5] from a specific activity in Chipper's app using the *startActivityForResult*[6] API. However, since all the functionalities of the PluginService are sensitive, only some apps satisfying some requirements can gain full access to it and this is decided by the activity that issues the *PendingIntent*s.

  (**B**) By exploiting the erroneous authentication in the activity that issues the *PendingIntent*s, attackers can post new messages on behalf of the user or even leak her credentials as they would have full access to the functionalities of the PluginService.

  (**C**) The attacker has to build an app (or extend the supplied attacker's app) to circumvent the authorization check when retrieving the PendingIntent and then use these privileges to perform the actions described above.
- (**A**) This vulnerability has the same concept of the first one except it is exploited differently. As a starting point, trace the code used for posting a new message. The attacker should be able to retrieve all messages sent by the user of the Chipper app.

  (**B**) The attacker should be able to retrieve all messages posted by the Chipper user directly after they are posted. The Chipper app sends out notifications to selected receivers, however, those notifications are not properly protected, allowing an attacker app to receive them, thus leaking the messages.

  (**C**) The attacker has to build an app (or extend the supplied attacker's app) that listens for broadcasted Intents (i.e., BroadcastReceiver). To receive them, the attacker app just has to have the right package name and receiver component set, no further authentication is needed.

File-system leaks

The app has two leaks of sensitive information on the file-system, one unprotected storage and another via crypto misuse.

Hints:

- (**A**) Each app has a private directory that is inaccessible to other apps. However, there is also a shared directory that is accessible by all apps holding specific permissions (i.e., READ _EXTERNAL _STORAGE and WRITE _EXTERNAL _STORAGE). Given this information, search the code used for sharing messages for an exploit.

  (**B**) This vulnerability can be used to leak shared messages.

  (**C**) The attacker should build an app (or extend the supplied attacker's app) to observe the shared storage for new files and leak them.

[5] https://developer.android.com/reference/android/app/PendingIntent
[6] https://developer.android.com/training/basics/intents/result

– (**A**) The Chipper app enables users to backup their posts. The backup file is stored on the shared storage, but since posts are of sensitive nature, they are encrypted. Start discovering this exploit by checking the code used for creating backups.

(**B**) This vulnerability enables the attacker to read and decrypt the content of the backup. The encryption key is derived in a predictable fashion that any other app can easily reproduce.

(**C**) Build and app (or extend the supplied attacker's app) to leak the backup file by reading and decrypting it. The attacker simply has to apply the same code for key derivation as the Chipper app to retrieve the right decryption key.

A Introduction to Chipper App and Service

The goal of this workshop is to perform a security analysis of a simple Twitter-like application called *Chipper*. In the following, the general workflow from the user's perspective will be described as well as setting up the environment for your analysis.

Scenario

The general scenario for the Chipper app is depicted in Figure 1: The app, installed on the user's device, communicates over two different channels with the Chipper server. One channel is used for retrieving HTML content that is being displayed to the user, the other channel contacts the server's REST API for management purposes, such as sign up/out, posting messages, etc.

The app's functionality is limited and it offers the user to view public messages posted on Chipper, create an account and then view private messages (private = shared between only signed in users) and post public/private messages themselves, create backups of their message history, and share messages with other apps.

The server is configured under the domain `https://androidlecture.de:5000`.

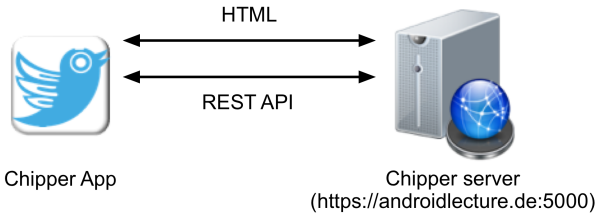

Figure 1: General Chipper scenario.

Workflow for User

The following is the workflow of interacting with the app from the user's perspective. Figure 2 depicts the home screen of the Chipper app. On this screen, all *public* postings by all users are shown. From the home screen, the user can navigate to different management interfaces through the main menu (Figure 3). For instance, the user can sign up to the Chipper service as shown in Figure 4 and also delete his account (Figure 10), which will result in all the user's message being deleted on the server.

As logged in user, the user can see all public and private messages (see Figure 6) and also only as signed in user it is possible to post messages. By pressing the red mail icon in the bottom right corner in, e.g., Figure 6, the user comes to the message authoring activity as depicted in Figure 5. Besides the message content, the user can also attach a link to the message and choose whether the message is publicly visible to anyone or only visible to logged in users ("private"). After posting messages, public messages are marked in grey while private messages, i.e., messages only visible to logged in users, are marked in blue (see Figure 6).

The user can share messages through the main menu, then select the message to share by long-click (in Figure 7) and then confirming the sharing of this message (see Figure 8). Lastly, the user can create a backup of the messages (Figure 9).

Setup

Chipper App

The Chipper app is complied against SDK version 27 (i.e., Android Nougat) with minimum SDK Version 23 (i.e., Android Marshmallow).

Android Emulator

We successfully tested the Chipper app on the Android's emulator (shipped with Android SDK on Android Studio) and Genymotion. You are free to test and analyse the app on your prefered device/emulator as long as it remains fully functional.

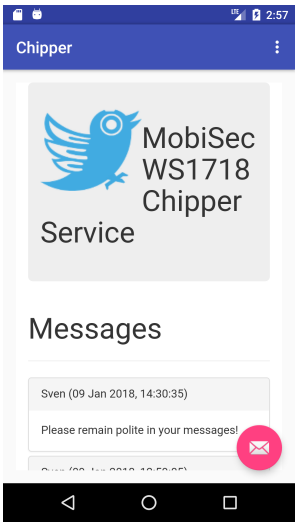

Figure 2: Home screen of Chipper app.

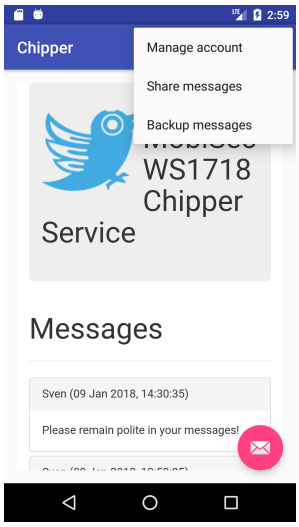

Figure 3: Main menu on home screen.

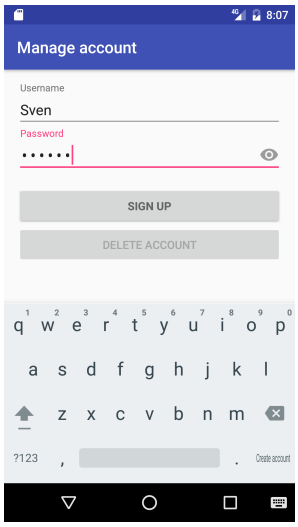

Figure 4: Creating a user account.

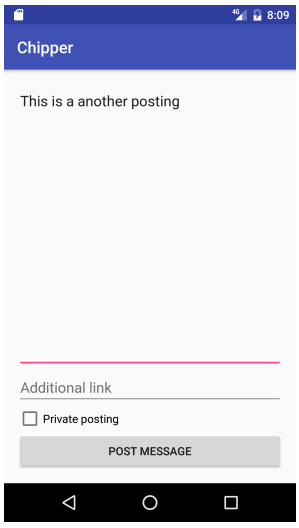

Figure 5: Posting a message.

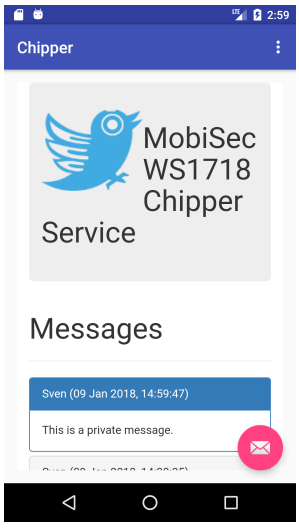

Figure 6: Home screen showing private and public messages.

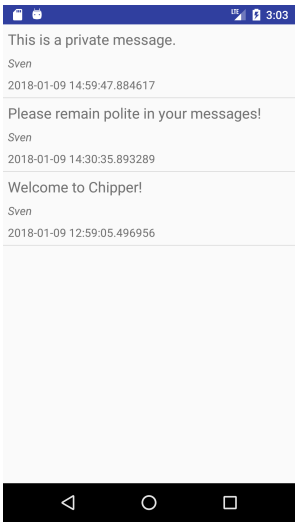

Figure 7: Selecting a message for sharing.

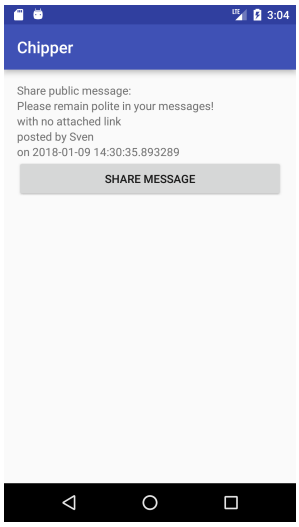

Figure 8: Confirming the message sharing.

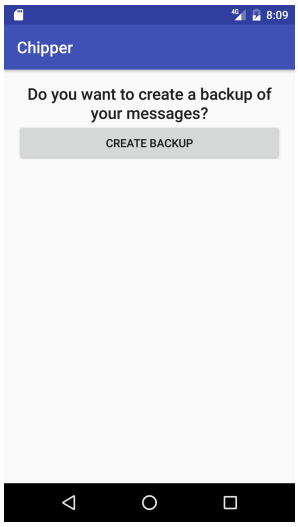

Figure 9: Creating a backup of the messages.

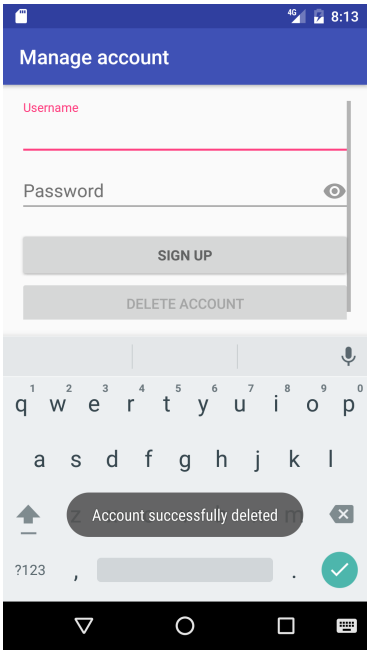

Figure 10: Deleting user account.

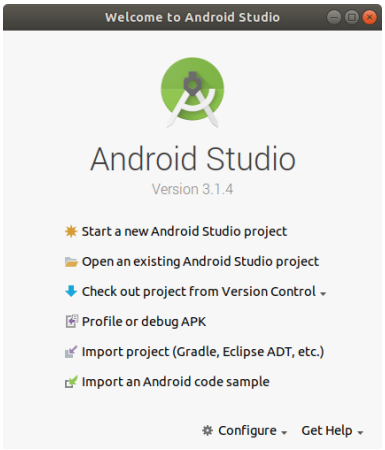

Figure 11: Import project with a fresh setup

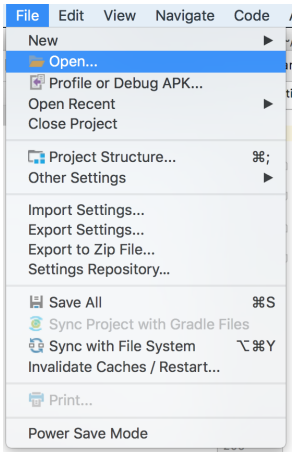

Figure 12: Import project while an old project is already opened

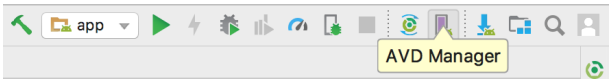

Figure 13: Create an emulator

Chipper Server

The Chipper server is running under `https://androidlecture.de:5000` and can only be accessed from within the Saarland University network. Your messages are readable by other students, be careful about your words (e.g., don't insult people, etc.).

B Introduction to Android Studio

In this section, we show how to import the attack app that can be used to launch/complete attacks against the Chipper app. We also present how to create an emulator that would run both the Chipper and the attack app. If you have experience in Android development, you might need to consider creating your own attack app from scratch and possibly skip this section.

Importing Attacker's Project

To import the attacker's app, download the Attacker's Android Studio project and place it somewhere that is easily reachable. Then open Android Studio. If the screen on Figure 11 is shown, then click on "Open an existing Android Studio project". Navigate to where you stored the project and select it. Then, press the OK button. If Android Studio opened with an already opened project, then you can import the Attacker's project by clicking on File, select open, navigate to project, and finally open it (see Figure 12).

Creating an Emulator

To test your attacks, you need to run both the Chipper and your Attacker's app on an emulator. To create a new emulator, open the AVD Manager (see Figure 13) and then, from the new window, click on "Create Virtual Device". Then choose *Nexus 5* and then click next. The next interface asks for specifying the system image (you might need to download the system image) which the emulator should use. Select Marshmallow and then click *Next*. From the new view, keep the *Emulated Performance - Graphics* unchanged and then click finish. If the emulator did not work (see instructions on the next section, then you might need to change the *Emulated Performance - Graphics* to *Software* instead of *Automatic*. At this point, you should have Attacker's project imported and a runnable emulator.

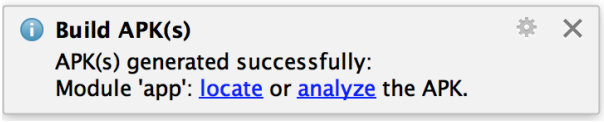

Figure 14: The result of successful app build

Running Apps on the Emulator

C Decompiling Android APK

Android apps are written in Java[7] and distributed as APKs that pack apps' resources and code. App's code (i.e., the *.DEX* file) is an intermediate representation called *Smali* which is compiled from Java. App's *DEX* file is supplied to Android Runtime (ART) which compiles it into native code and runs it on the device.

There are several tools that can be used to decompile and reverse engineer Android apps (i.e., APK files and included DEX files). The most prominent tools are, but not limited to, *apktool*[8] and *jadx*[9]. For those who are not familiar with these tools, we provide a brief description:

1. **apktool**: This tool extracts app's resources from the *APK* file and constructs app's *Smali* code from the *DEX* file into nearly its original form. At this point, changes can be applied to App's code and resources. Then, the same tool is used to build the *APK* and prepare it to run on devices/emulators. This process breaks app's original signature. As an immediate consequence, the same origin policy is violated resulting in this app not receiving further updates from the original developer, among other possible degradations in functionality in the same app or other apps that cooperate with it.

2. **jadx**: This tool decompiles App's *APK* or *DEX* files into Java. The output of this operation cannot immediately be compiled again and packed as a valid *APK* that would run on devices/emulators. Nevertheless, this tool is helpful for those who are not familiar with debugging and analyzing *Smali* code and prefer Java instead.

Bear in mind that neither of these tools deobfuscates the code. Also, you are free to use your preferable tool for decompiling/reverse engineering app's code. However, this should only be used to analyze, debug, and test app's code and *not* to modify the app to achieve your attacks.

Decompiling APKs using Jadx

Jadx provides command-line and GUI support to several platforms (i.e., Mac OS, Linux, and Windows). You can either download the binaries [10] or compile the project from the source[11]. In either case, you will have the necessary binaries used to decompile APKs under *./build/jadx/bin* when compiled from source, or under *./bin* otherwise.

To decompile the Chipper APK using command line, execute the following:

```
./jadx chipper.apk
```

The output directory that results from the execution of this command contains Chipper's obfuscated code, assists, and meta data files. Import the folder into your preferable text editor and get into the analysis.

[7] Android Apps can also include C/C++ code. However, this is out of the scope of this discussion and exercise.
[8] See https://ibotpeaches.github.io/Apktool/
[9] See https://github.com/skylot/jadx
[10] https://github.com/skylot/jadx/releases
[11] https://github.com/skylot/jadx