Surreptitious Sharing on Android

Dominik Schürmann[1] Lars Wolf[1]

Abstract: Many email and messaging applications on Android utilize the Intent API for sharing images, videos, and documents. Android standardizes Intents for sending and Intent Filters for receiving content. Instead of sending entire files, such as videos, via this API, only URIs are exchanged pointing to the actual storage position. In this paper we evaluate applications regarding a security vulnerability allowing privilege escalation and data leakage, which is related to the handling of URIs using the *file* scheme. We analyze a vulnerability called *Surreptitious Sharing* and present two scenarios showing how it can be exploited in practice. Based on these scenarios, 4 email and 8 messaging applications have been analyzed in detail. We found that 8 out of 12 applications are vulnerable. Guidelines how to properly handle file access on Android and a fix for the discussed vulnerability are attached.

Keywords: Android, sharing, vulnerability, Intent, API.

1 Motivation

Android includes a rich API for communication and interaction between applications. It introduced the concept of Intents with a set of standardized actions to facilitate sharing of data. This allows applications to concentrate on its core functionality and rely on others to deal with extended use cases. For example, there is no need to support recording of video when implementing an email client, instead a video recorder can directly share a finished recording with the email application. In this example, it is crucial to only share the intended video—other recordings must be protected against unauthorized access. Android provides a variety of security mechanisms to implement access control. These include sandboxing on file system layer via Unix UIDs, as well as Android permissions and URI permissions on the API layer. Even though these mechanisms are already sophisticated in comparison to traditional desktop operating systems, several vulnerabilities have been discovered and discussed in the research community [EMM12, Mu14b, Mu15, Ba15a, Mi15]. These often emerge from edge cases in inter-application communication, which were not considered in the application developer's or Android's security model.

The main contribution of this paper is the presentation and evaluation of a security vulnerability we call *Surreptitious Sharing* related to content sharing via *file* schemes. So far this vulnerability has been neglected in literature and to the best of the authors' knowledge a related vulnerability exploiting *file* URIs has only been described in a security audit by Cure53 [He15]. We provide a detailed explanation of the vulnerability and analyze popular applications in regards to it. Finally, we will discuss best practices of securing Android against this issue and what developers can do to protect their users.

[1] TU Braunschweig, Institute for Operating Systems and Computer Networks, Mühlenpfordtstr. 23, 38106 Braunschweig, {schuermann, wolf}@ibr.cs.tu-bs.de

Since Android's inception and widespread adoption, a huge amount of research papers has been published analyzing certain security mechanisms. A great overview of Android's model and especially URI handling can be found in [EMM12]. An example of recent work in this area are publications of Bagheri et al. In [Ba15a], they developed a formal model to describe and evaluate Android's permission model to discover issues, e.g., privilege escalation attacks. They found a previously undiscovered flaw in Android's handling of custom permissions, which are still heavily based on installation order. They also found an URI permission flaw, where applications were able to access certain URIs because URI permissions are not properly revoked when the associated content provider has been uninstalled. In a different paper, they presented COVERT [Ba15b], a tool to evaluate the security of inter-application communication. Using formal models, the interactions of multiple applications can be checked based on the applications' control-flows. They showed their ability to find privilege escalation vulnerabilities using an experimental evaluation with applications from Google Play, F-Droid, MalGenome, and Bazaar. Still, it is not clear if the vulnerability presented in this paper could have been found using their model, which does not consider the underlying file system permissions.

While a vast amount of research has been published about static and dynamic analysis tools for Android, we will focus on specific discovered vulnerabilities related to the one presented in this paper. Mark Murphy discusses permission and URI permission issues in several blog posts. He describes a problem where an application can gain access to protected functionality by being installed before the targeted application, declaring the same custom permission as the targeted application while at the same time requesting the permission via `<uses-permission>` [Mu14b]. This vulnerability no longer works on Android 5; the installation of a second application requesting the same permission now fails with `INSTALL_FAILED_DUPLICATE_PERMISSION` [Mu14a]. In [Mu15], he discusses the limits of URI permissions granted via Content Providers.

A different way to gain access to data is by tricking the users to input their private information into user interfaces controlled by an attacker. Qi Alfred Chen et al. were able to launch malicious user interfaces from background processes to hijack login screens, i.e., launch a similar looking screen where the user enters her credentials [CQM14]. They were able to determine precise timings by detecting UI state changes via side channels of Android's shared memory [CQM14]. The authors of [NE13] tried to prevent other accidental data leakages on Android. They present Aquifer, a policy framework that developers can use to define restrictions and to protect their user interfaces in addition to Android's security model. They implemented an example using K-9 Mail, which has also been analyzed in this paper.

Other paper looking at application specific issues have been published. In [Fa13], 21 password manager have been evaluated. Besides complete failures in regards to cryptographic implementations, the main problem is that most password managers pasted passwords into Android's clipboard, which can be accessed without additional permissions. They propose fixes that require changes in the Android operating system, as well as fixes that can be deployed by developers. A more automated process of finding misused cryptographic principles has been investigated in [Eg13]. The authors found that 88% of all considered applications violated at least one posed rule. An evaluation of the security of nine popular messaging tools has been presented in "Guess Who's texting You?" [Sc12]. It focuses on several aspects, such as deployed authentication techniques, registration mechanisms, and privacy leakages. A detailed analysis of WhatsApp can be found in [Th13].

A vulnerability similar to the one discussed in this paper has been discovered by Rob Miller of MWR Labs [Mi15]. He was able to bypass Android's file system permission model using Google's Android Admin application by opening a specially crafted webpage with this application. This webpage is opened inside the application's context and includes a *file* URI, which can be used to gain access to the private storage.

2 IPC and Access Control on Android

Before discussing the vulnerability in detail, we introduce fundamentals of Android's Inter Process Communication (IPC) and Access Control mechanisms. Android's IPC is based on the Binder mechanism. It is not based on traditional System V components, but has been implemented due to performance and security reasons as a patch for the Linux kernel. Memory can be allocated and shared between processes without making actual copies of the data. Binder's security is based on a combination of the usage of Unix UIDs and Binder capabilities. It is differentiated between direct capabilities, which are used to control access to a particular interface, and indirect capabilities, which are implemented by sharing tokens between applications. Android Permissions are checked by verifying that a permission that is associated to a Binder transaction is granted to the participating application. UIDs are used as an identifier in this process.

The Binder mechanism is normally not directly used inside the Android Open Source Project (AOSP) and Android applications. Instead, more easily usable IPC primitives exist that are based on Binder allowing communication between processes. These are Activity/Broadcast Intents, Messenger, and Services. While these primitives can be used to share data, they are only suitable for up to 1 MB of actual payload. Instead of sharing the data directly as a copy, it is advised to provide it via Content Providers and only share a Uniform Resource Identifier (URI) pointing to the payload. Google provides the class `FileProvider`[3] shipped with their support library to ease and secure the implementation of file sharing between applications. A file that is intended to be shared is temporarily stored inside the application's private cache directory and all metadata inside `FileProvider`'s database. A unique URI is created, and this URI is shared, for example via an Intent, to a different application. URIs served via `FileProvider` are using the *content* scheme, e.g., `content://com.google.android.apps.docs.files/exposed_content/6jn9cnzdJbDy`. Access to these files can be granted using two different methods. Calling `Context.grantUriPermission(package, Uri, mode_flags)` allows another package to retrieve the file until `revokeUriPermission()` is executed. Possible flags are `FLAG_GRANT_READ_URI_PERMISSION` and `FLAG_GRANT_WRITE_URI_PERMISSION`. If the URI is shared by Intent only, the flags can also be directly assigned via `Intent.setFlags()` which means that

[3] `http://developer.android.com/reference/android/support/v4/content/FileProvider.html`

access is granted to the called Activity until the corresponding stack is finished. Besides these *content*-URIs, Android supports a legacy way of sharing files, which is discouraged in recent documentations: URIs starting with the *file* scheme can point to an actual file in the file system, e.g., `file:///sdcard/paper.pdf`. The security of *file* URIs is based on file system permissions only, i.e., Unix UIDs. For the default case of accessing files on the external storage, access control is handled differently on Android versions: Since SDK 4 `WRITE_EXTERNAL_STORAGE` permission is required to access the card. In SDK 16, `READ_EXTERNAL_STORAGE` has been introduced for read operations only. Since SDK 19, actual external storages (not internal ones, which strangely also fall under the broader definition of "external storage") are no longer readable or writable. In SDK 21, the Storage Access Framework has been introduced which allows to access external storages via a new API, complementing the original Java `File` API. Since SDK 23, runtime permissions are introduced. `READ_EXTERNAL_STORAGE` permission must be granted during runtime to access *file* based URIs pointing to the SD card.

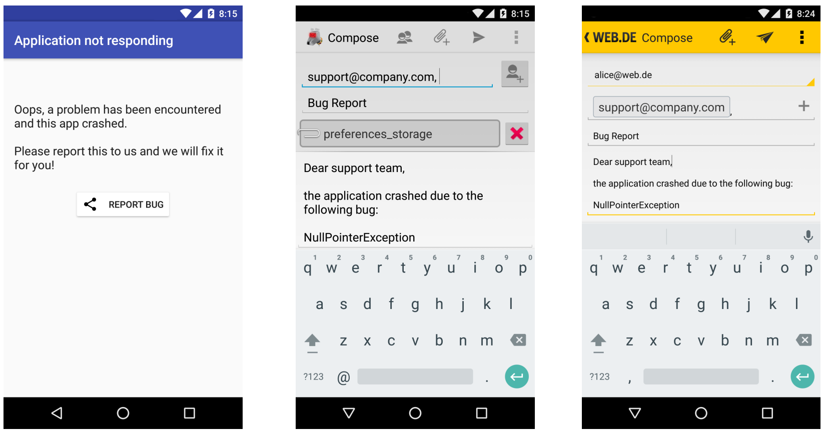
On the receiving side, these URIs are opened via `ContentResolver.openInputStream (Uri uri)`. This method supports the schemes *content*, *file*, and *android.resource* (used to open resources, e.g., `android.resource://package_name/id_number`).

In addition to storing on external mediums, applications can save private data using `Context.openFileOutput(String name, int mode)`. This datum is saved in `/data/data/com.example.app/` and can only be access by the application itself, not other applications or the user (in contrast to storing on SD cards)[4]. These methods utilize the standard File API and streaming classes of Java. Each application on Android has its own file system UID that protects these data directories against unauthorized access. In effect, all Android applications are separated from one another on the file system layer.

2.1 Sharing API

Using the described IPC methods, Android provides great ways to allow sharing of data between applications. One of the most simple standardized ways of sharing content between applications is by using share buttons or menus implemented in a variety of applications. By constructing an Intent with the `android.intent.action.SEND` action, a sender can transfer text or other content. The actual payload is included via so called extras, which are serialized using Android's Parcelable methods. For `SEND`, the following extras exist (all prefixed with `android.intent.extra.`): `TEXT`, `HTML_TEXT`, `STREAM`. While `TEXT` and `HTML_TEXT` contain the whole String, `STREAM` contains an URI, which points to the actual content via *content* or *file* schemes. Other standardized extras are `EMAIL`, `CC`, `BCC`, `SUBJECT`. Applications can introduce new extras, as Intent APIs are not fixed like traditional IPC interfaces. There exists a range of other Intent actions with similar semantics, such as `SEND_MULTIPLE`. It is basically the same mechanism as `SEND`, but allows sharing of multiple URIs using a list.

[4] `http://developer.android.com/guide/topics/data/data-storage.html#filesInternal`


Application not responding

Oops, a problem has been encountered and this app crashed.

Please report this to us and we will fix it for you!

support@company.com,

Bug Report

preferences_storage

Dear support team,

the application crashed due to the following bug:

NullPointerException

alice@web.de

support@company.com

Bug Report

Dear support team,

the application crashed due to the following bug:

NullPointerException

(a) Pretended crash encouraging a bug report (b) K-9 Mail (c) WEB.DE

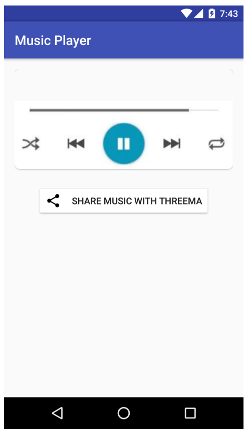
Fig. 1: Surreptitious Sharing of IMAP passwords stored in email clients (Scenario 1)

To receive data sent by these Intents, Android allows to specify Intent Filters inside the `AndroidManifest.xml`. Intent Filters consist of an XML structure defining the receiving action, the accepted MIME types, and filters which URIs are accepted[5]. It is important to note that explicitly declaring MIME types when constructing an Intent helps the receiving application to determine the type of content, but without any checks whatsoever that the actual payload is of that type. No *magic bytes* are checked by the Android OS.

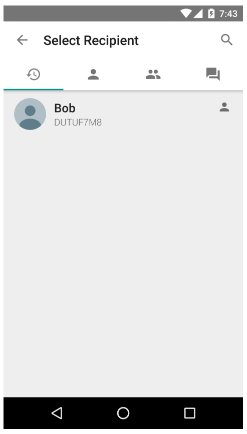
3 Surreptitious Sharing

The central issue presented in this paper is related to the security of URIs using the *file* scheme. As discussed previously, access control to URIs based on this scheme is handled via traditional Unix file system permissions. The introduced permissions `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` are designed as additional barriers for accessing SD cards. The main issue lies in the fact that applications cannot only access their private data directories using `Context.openFileOutput(String name, int mode)`, but also using *file* URIs. While these URIs are normally used to access files on the SD card, via `file:///sdcard/paper.pdf` for example, they can also point to private files, e.g., `file:///data/data/com.example.app/files/paper.pdf`. If an application registers Intent Filters to support Android's sharing API or defines custom Intents accepting URIs, they are potentially accepting *file* URIs that could also point to their own private files. For applications facilitating communication, like email or messaging applications, this leads to what we call *Surreptitious Sharing*. To our knowledge, this issue has first been documented as vulnerability OKC-01-010 in Cure53's security audit of the OpenPGP application OpenKeychain [He15]. While their report applies this issue to the file

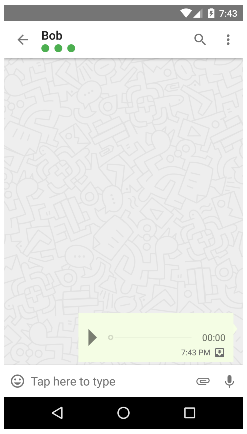
[5] `http://developer.android.com/guide/topics/manifest/intent-filter-element.html`



(a) Music player encouraging music sharing



(b) Threema: Select recipient



(c) Threema: Shared database shown as audio recording

Fig. 2: Surreptitious Sharing of messaging databases (Scenario 2)

encryption API in OpenKeychain, which accepts *file* URIs, we apply it in a broader context to communication applications. Investigating the AOSP source code reveals that support for *file* URIs using `Context.openFileOutput(String name, int mode)` (similar checks are present in `openAssetFileDescriptor`) was planned to be removed [6]. To demonstrate the impact of this security vulnerability, we consider two attack scenarios: Scenario 1) “Fake Bug Report” and Scenario 2) “Music Sharing”.

Scenario 1 is intended to surreptitiously share IMAP passwords of email clients to an attacker. Following Figure 1, interaction with Scenario 1 consists of a sequence of actions. A malicious application shows a screen indicating that a problem has occurred urging the user to report the bug to the developers. Touching the button starts a malicious Intent specially crafted for a particular email application with an URI pointing to a private file of this email application, containing the IMAP password. Generally, this Intent uses the `SEND` action and a set of extras to prefill the recipient for example in addition to the malicious *file* URI inside the `STREAM` extra. The Intent is explicitly defined using the package name and the exact Activity. The private file is attached to the email shown in the compose screen. The user finally needs to send this email to expose her sensitive information.

Scenario 2 is designed to exploit messaging applications and share their databases to obtain message histories, for example. Instead of faking a crash followed by a bug report, it consists of a functional music player also featuring a button to share music with friends via installed messengers (cf. Figure 2). The button is prominently placed to encourage sharing. Based on the targeted messaging application, the user interfaces following the sharing action, differ.

[6] see inline comments in `openInputStream` method in `https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/content/ContentResolver.java`

Application	Version	Component	Open source	Displays filename	Ignores MIME type	File pre-processing	Security check	Secure
K-9 Mail	5.006	MessageCompose	●	●	●	○	○	○
AOSP Mail[a]	5.1.1-ecfa8c7973	ComposeActivityEmail	●	●	●	○	◐	○
GMail	5.8.107203005	ComposeActivityGmail	○	●	●	○	◐	○
WEB.DE	3.6	MailComposeActivity	○	●	●	○	○	○

[a] AOSP Mail from CyanogenMod 12.1

Tab. 1: Scenario 1 using SEND Intents with *file* URI

Even if it is not known beforehand which applications are installed, Android's API allows to query for available components via `PackageManager.queryIntentActivities()`. Thus, it is easy to search for targeted applications and then use a custom malicious Intent. In the following, we will investigate several applications in regards to this vulnerability. The applications have been evaluated on Android 6.0 (SDK 23), except AOSP Mail which has been tested on CyanogenMod 12.1 (Android 5.1, SDK 22). However, the vulnerability itself is present on all Android versions.

3.1 Scenario 1: Fake Bug Report

We will first look into two of the most popular open source email applications, K-9 Mail and AOSP Mail, and analyze their security. Due to the availability of their source code, it is possible to analyze the actual implementation of opening URIs more precisely. Afterwards, GMail and WEB.DE apps are tested. All results are listed in a concise form in Table 1.

K-9 Mail The crafted exploit (cf. Appendix 7.1) works as intended. As typical for an email client that accepts all MIME types (filters for "*/*"), no pre-processing is done for attached files and explicitly declared MIME types via `Intent.setType()` are ignored. The file `preference_storage`[7] containing the IMAP passwords is shared and displayed in K-9 Mail (cf. Figure 1b). At least, the attached file is displayed upfront to the user.

AOSP Mail Google has stopped introducing new features to the AOSP Mail client since the GMail application supports external IMAP accounts starting from version 5.0. Still, the AOSP Mail client is used as the code base for many derived proprietary clients and is still maintained by custom roms such as CyanogenMod. Here, we tested a version from CyanogenMod 12.1.
AOSP Mail has a security check in place to prevent attachments from data directories

[7] `file:///data/data/com.fsck.k9/databases/preferences_storage`

except from directories of the calling application, which shows "Permission denied for the attachment."[8]. By creating a world-readable hard link from the malicious application to the AOSP Mail's `EmailProvider.db`[9] file we were able to work around this security check and successfully attach the database containing the IMAP password (cf. Appendix 7.1). This also allows to hide the actual filename as only the name of the hard link is shown in the displayed attachments.

GMail GMail also has a similar security check in place resulting in "Permission denied for the attachment". Again, we were able to circumvent this by using a hard link. However, we were not able to exploit GMail on Android 6, maybe due to the new runtime permissions; this has not been investigated further. Retrieving the Google password is not possible due to the usage of OAuth in combination with the AccountManager API. Still, stored emails can be retrieved by retrieving the Google account name via `AccountManager.getAccounts()` and then sharing a URI pointing to `mailstore.example@gmail.com.db`[10].

WEB.DE The WEB.DE Android client is based on K-9 Mail and behaves similarly, except that the attachment is initially hidden behind the keyboard making it more difficult for the user to notice it (cf. Figure 1c).

3.2 Scenario 2: Music Sharing

Due to the nature of messaging applications, these are rarely used for bug reports. Thus, to exploit these, Scenario 2 is used. Table 2 gives an overview of the analyzed applications and their properties. We looked at five popular messaging applications from Google Play and three selected privacy-focused ones. In contrast to email applications, most messaging applications do not ignore the given MIME type and often rely on it instead of detecting the type itself.

WhatsApp We were not able to exploit WhatsApp. According to our analysis, only image, audio, and video types are supported. Sharing the message database[11] as an image MIME type resulted in "The file you picked was not an image file.", which seems to be detected during pre-processing of the image. Audio and video files were displayed in the message history, but were not properly uploaded; the retry button did not work.

Hangouts Hangouts behaves similarly to WhatsApp. Sending the database[12] as an image results in "Media format not supported".

[8] `https://android.googlesource.com/platform/packages/apps/UnifiedEmail/+/adea2c809c1a97d3948b131a6f36ee9b88039a45`
`https://android.googlesource.com/platform/packages/apps/UnifiedEmail/+/24ed2941ab132e4156bd38f0ab734c81dae8fc2e`

[9] `/data/data/com.android.email/databases/EmailProvider.db`

[10] `file:///data/data/com.google.android.gm/databases/mailstore.example@gmail.com.db`

[11] `file:///data/data/com.whatsapp/databases/msgstore.db`

[12] `file:///data/data/com.google.android.talk/databases/message_store.db`

Application	Version	Component	Open source	Displays filename	Ignores MIME type	File pre-processing	Security check	Secure
WhatsApp	2.12.365	ContactPicker	○	○	○[a]	●	○	●
Hangouts	6.0.107278502	ShareIntentActivity	○	○	○[a]	●	○	●
Facebook Messenger	50.0.0.11.67	ShareIntentHandler	○	◐	●[a]	◐	●	●[a]
Skype	6.12.0.585	SplashActivity	○	◐	●	◐	○	○
Snapchat	9.20.2.0	LandingPageActivity	○	○	●	●	○	●[a]
Threema	2.532	RecipientListActivity	○	◐	○	◐	○	○
Signal	3.5.2	ShareActivity	●	○	○	◐	○	○
Telegram	3.2.6	LaunchActivity	●	◐	○	◐	○	○

[a] Indefinite results, difficult to verify because application is not open source

Tab. 2: Scenario 2 using `SEND`, `SEND_MULTIPLE` Intents with *file* URIs

Facebook Messenger Facebook messenger registers the Intent for audio, video, image, text/plain and we were able to share a test file from the SD card, which was properly been sent and displayed in message history with its filename. However, we were not able to exploit the application. Sharing a malicious URI[13] from the internal storage resulted in “Sorry, Messenger was unable to process the file”.

Skype Skype allows sharing of any MIME type and we were easily able to execute the exploit to retrieve `offlinestorage.db`[14]. Pre-processing is done for images and videos; sharing the malicious URI with these MIME types results in a frozen program.

Snapchat We were not able to retrieve private files from Snapchat, because it is limited to image files which are pre-processed before sending.

Threema As depicted in Figure 2, we were able to retrieve Threema’s database[15] shown as an audio recording by setting the MIME type to “audio/mp4”. This is possible because Threema does not pre-process these files before sending. Obviously, the audio cannot be played. The file can still be saved successfully by the receiving attacker and opened as a database file. Threema supports encrypted databases, which, according to [DLP13], are encrypted using a key saved beside the database. Using `SEND_MULTIPLE`, we were able to retrieve both the database and the key[16]. However, explicit MIME types are ignored here, thus the files cannot be hidden as audio recordings.

[13] `file:///data/data/com.facebook.orca/databases/threads_db2`
[14] `file:///data/data/com.skype.raider/files/offlinestorage.db`
[15] `file:///data/data/ch.threema.app/databases/threema.db`
[16] `file:///data/data/ch.threema.app/files/key.dat`

Signal Signal was in the same way vulnerable as Threema, i.e., its database[17] has been shared as an audio recording. Sending it using "image/png" resulted in a crash due to pre-processing in `AttachmentDatabase.getThumbnailStream()`. Because the fake image has been cached in Signal's database, Signal now crashes on each start.

Telegram Telegram was exploitable[18] but we were not able to hide the filename or type with any of the discussed tricks. However, using the hard link can help to make the user less suspicious.

4 Countermeasures

Upcoming Android versions should incorporate a security check into `ContentResolver .openInputStream()` that prevents opening of *file* URIs where the files are solely owned by the application's UID (cf. Appendix 7.2). In addition, Google's support library should be extended. While it contains a `FileProvider`[19] to provide files to other applications based on *content* URIs avoiding the *file* scheme, it should additionally include the method from Appendix 7.2 to securely open streams from URIs.

Furthermore, we recommend that application developers follow best practices to prevent Surreptitious Sharing: Content shared from other applications should be considered as unverified input that needs to be explicitly acknowledged by the user. Even after fixing the vulnerability, applications could still share different content than what has been shown to the user before, e.g., files from an SD card instead of the displayed image. Thus, we propose to pre-process content if possible as well as display the content and filename before actually sending it.

5 Conclusion

In this paper, we analyzed 12 applications in respect to a security vulnerability called Surreptitious Sharing. Our evaluation showed that 8 applications were exploitable and security checks implemented in GMail and AOSP Mail could be bypassed. Unfortunately, especially the privacy-focused messaging applications were easily exploitable. Hiding the private files by setting an explicit MIME type has been shown to work in Signal and Threema. Besides fixing the vulnerability, we recommend best practices for application developers how to handle shared files.

6 Acknowledgments

We thank Cure53 for discovering the original vulnerability OKC-01-010 [He15] and discussing possible attack vectors via hard links. We thank all vendors in advance for fixing the vulnerabilities.

[17] `file:///data/data/org.thoughtcrime.securesms/databases/messages.db`
[18] `file:///data/data/org.telegram.messenger/files/cache4.db`
[19] `http://developer.android.com/reference/android/support/v4/content/FileProvider.html`

References

[Ba15a] Bagheri, Hamid; Kang, Eunsuk; Malek, Sam; Jackson, Daniel: Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification. In (Bjørner, Nikolaj; de Boer, Frank, Hrsg.): FM 2015: Formal Methods, Jgg. 9109 in Lecture Notes in Computer Science, S. 73–89. Springer, 2015.

[Ba15b] Bagheri, Hamid; Sadeghi, Alireza; Garcia, Joshua; Malek, Sam: COVERT: Compositional Analysis of Android Inter-App Permission Leakage. IEEE Transactions on Software Engineering, 41(9):866–886, Sept 2015.

[CQM14] Chen, Qi Alfred; Qian, Zhiyun; Mao, Z Morley: Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In: Proceedings of the 23rd USENIX Security Symposium. S. 1037–1052, 2014.

[DLP13] Dimitrov, Hristo; Laan, Jan; Pineda, Guido: Threema security assessment. In: Research project for Security of Systems and Networks. Dezember 2013.

[Eg13] Egele, Manuel; Brumley, David; Fratantonio, Yanick; Kruegel, Christopher: An Empirical Study of Cryptographic Misuse in Android Applications. In: Proceedings of the 2013 Conference on Computer and Communications Security (CCS). ACM, New York, NY, USA, S. 73–84, 2013.

[EMM12] Egners, Andre; Meyer, Ulrike; Marschollek, Björn: Messing with Android's Permission Model. In: 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). S. 505–514, June 2012.

[Fa13] Fahl, Sascha; Harbach, Marian; Oltrogge, Marten; Muders, Thomas; Smith, Matthew: Hey, You, Get Off of My Clipboard. In: Financial Cryptography and Data Security, Jgg. 7859 in Lecture Notes in Computer Science, S. 144–161. Springer, 2013.

[He15] Heiderich, Mario; Horn, Jann; Aranguren, Abraham; Magazinius, Jonas; Weißer, Dario: Pentest-Report OpenKeychain, August 2015. `https://cure53.de/pentest-report_openkeychain.pdf`.

[Mi15] Miller, Rob: Sandbox bypass through Google Admin WebView. MWR Labs, August 2015.

[Mu14a] Murphy, Mark: Custom Permission Vulnerability and the 'L' Developer Preview. Commonsware Blog, August 2014. `https://commonsware.com/blog/2014/08/04/custom-permission-vulnerability-l-developer-preview.html`.

[Mu14b] Murphy, Mark: Vulnerabilities with Custom Permissions. Commonsware Blog, Februar 2014. `https://commonsware.com/blog/2014/02/12/vulnerabilities-custom-permissions.html`.

[Mu15] Murphy, Mark: The Limits of ContentProvider Security. Commonsware Blog, Juli 2015. `https://commonsware.com/blog/2015/07/13/limits-contentprovider-security.html`.

[NE13] Nadkarni, Adwait; Enck, William: Preventing accidental data disclosure in modern operating systems. In: Proceedings of the 2013 Conference on Computer and Communications Security (CCS). ACM, S. 1029–1042, 2013.

[Sc12] Schrittwieser, Sebastian; Frühwirt, Peter; Kieseberg, Peter; Leithner, Manuel; Mulazzani, Martin; Huber, Markus; Weippl, Edgar R: Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In: 19th Annual Network & Distributed System Security Symposium (NDSS). 2012.

[Th13] Thakur, Neha S.: Forensic analysis of WhatsApp on Android smartphones, 2013.

7 Appendix

7.1 Example Exploit Targeting K-9 Mail

```
public void exploit() {
    Intent intent = new Intent();
    intent.setComponent(new ComponentName("com.fsck.k9", "com.fsck.k9.activity.
        MessageCompose"));
    intent.setAction(Intent.ACTION_SEND);
    String linkPath = createLink("/data/data/com.fsck.k9/databases/
        preferences_storage");
    Uri uri = Uri.parse("file://" + linkPath);
    intent.putExtra(Intent.EXTRA_STREAM, uri);
    intent.putExtra(Intent.EXTRA_EMAIL, new String[]{"support@company.com"});
    intent.putExtra(Intent.EXTRA_TEXT, "Dear support team,\n\nthe application
        crashed due to the following bug:\n\nNullPointerException");
    intent.putExtra(Intent.EXTRA_SUBJECT, "Bug Report");
    startActivity(intent);
}

private String createLink(String path) {
    File link = new File(getFilesDir().getPath() + "/bug_report");
    link.delete();
    try {
        Os.link(path, link.getAbsolutePath());
    } catch (ErrnoException e) {
        Log.e("SurreptitiousSharing", "hard link failed", e);
    }
    link.setReadable(true, false);
    return link.getAbsolutePath();
}
```

7.2 Secure Replacement for ContentResolver.openInputStream()

```
@TargetApi(VERSION_CODES.LOLLIPOP)
static InputStream openInputStreamSafe(ContentResolver resolver, Uri uri)
        throws FileNotFoundException {
    String scheme = uri.getScheme();
    if (ContentResolver.SCHEME_FILE.equals(scheme)) {
        ParcelFileDescriptor pfd = ParcelFileDescriptor.open(
                new File(uri.getPath()), ParcelFileDescriptor.parseMode("r"));
        try {
            final StructStat st = Os.fstat(pfd.getFileDescriptor());
            if (st.st_uid == android.os.Process.myUid()) {
                Log.e("SafeOpen", "File is owned by the application itself,
                    aborting!");
                throw new FileNotFoundException("Unable to create stream");
            }
        } catch (ErrnoException e) {
            Log.e("SafeOpen", "fstat() failed", e);
            throw new FileNotFoundException("fstat() failed");
        }

        AssetFileDescriptor fd = new AssetFileDescriptor(pfd, 0, -1);
        try {
            return fd.createInputStream();
        } catch (IOException e) {
            throw new FileNotFoundException("Unable to create stream");
        }
    } else {
        return resolver.openInputStream(uri);
    }
}
```