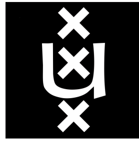

University of Amsterdam

MSc System and Network Engineering

Subverting Android 6.0 fingerprint authentication

Authors:

Thom Does
Thom.Does@os3.nl

Mike Maarse
Mike.Maarse@os3.nl

February 7, 2016

Abstract

Fingerprint recognition is becoming a standard means of authentication on mobile devices. The total number of mobile devices incorporating a fingerprint scanner is expected to reach 990 million in 2017, and over 50% of all smartphones are expected to have a fingerprint sensor by 2019.

With the introduction of Android 6.0, a new fingerprint authentication API has been added to the operating system. Application developers can use the native API to secure sensitive data and authorise certain actions, such as financial transactions.

In this paper, we describe two methods to circumvent fingerprint authentication in Android 6.0 resulting in false positive recognition. The first method involves replacing a critical software component to always provide a positive authentication result. In the second method, we intercept and alter Inter-Process Communication. This also enabled us to bypass fingerprint authentication, but is less detectable for end-users than the former method.

Additionally we also perform a replay attack on Android's *Keystore*, forcing the "release" of authentication-gated cryptographic keys.

Please note that all findings required root access and can only be practically exploited when having physical access.

Acknowledgements

First of all we would like to thank KPMG for providing us with all the necessary facilities and required equipment to conduct our research.

We would like to thank Paul van Iterson for proposing the research topic and for continuously supplying us with feedback on our work and overall progress.

Rick van Galen also has our gratitude, for his central role in organising our research project and providing feedback.

Finally, we would like to thank Jordi van den Breekel for his additional supervision.

Contents

1 Introduction **1**
1.1 Contribution . . . 1
1.2 Research question . . . 1
1.3 Related work . . . 2
1.4 Report structure . . . 2

2 Authentication system **3**
2.1 Software components . . . 3
2.2 Fingerprint HAL . . . 4
2.3 Hardware-backed Keystore . . . 4
2.4 Inter-process Communication . . . 5

3 Methodology **6**
3.1 Scope . . . 6
3.2 Test environment . . . 6
3.3 Research outline . . . 7

4 Results **8**
4.1 False positive recognition . . . 8
4.1.1 Fingerprint ID verification . . . 8
4.1.2 Replacing fingerprintd . . . 9
4.1.3 Modified IPC . . . 10
4.2 Replaying authentication tokens . . . 13
4.2.1 AuthToken table . . . 13
4.2.2 Challenge-response . . . 13
4.2.3 Retrieving the AuthToken . . . 14
4.2.4 Replaying the AuthToken . . . 15
4.2.5 Exploitability . . . 15
4.3 Case studies . . . 16
4.3.1 FingerprintDialog . . . 16
4.3.2 bol.com . . . 17

5 Discussion **18**
5.1 Vendor-dependent findings . . . 18
5.2 Feasibility of attacks . . . 18

6 Conclusion **20**

7 Recommendations **21**
7.1 OS developers . . . 21
7.1.1 dm-verity . . . 21
7.1.2 Unambiguous implementer references . . . 21
7.1.3 Erase AuthTokens after use . . . 21
7.1.4 Only offer secure authentication methods . . . 22
7.1.5 Protect Binder parcels' integrity . . . 22

7.2 App developers 22
7.2.1 Distrust rooted devices 22
7.2.2 Use the Keystore 22
7.3 End users 23

8 Future work **24**
8.1 Keystore 24
8.2 Gatekeeper 24
8.3 Other fingerprint functions 24
8.4 Fingerprint authentication resources 24
8.5 Vendor specific (HAL) libraries 25
8.6 Cross-reference findings 25

A Security enhancements **29**
A.1 SELinux 29
A.2 dm-verity 29
A.3 Trusted Execution Environment 30
A.4 Authentication-gated keys 30

B Binder IPC **31**
B.1 Communication model 31
B.2 Interfaces 31
B.3 Serialisation and transmission 31

C AuthTokens **33**

D Authentication flow **35**

E AuthToken challenges **36**

F Replay AuthToken code **37**

Acronyms

ADB Android Debugging Bridge.

AOSP Android Open Source Project.

API Application Programming Interface.

dm-verity device-mapper-verity.

HAL Hardware Abstraction Layer.

HMAC Hash Message Authentication Code.

IPC Inter-Process Communication.

MAC Mandatory Access Control.

OS Operating System.

RPC Remote Procedure Call.

SELinux Security-Enhanced Linux.

TEE Trusted Execution Environment.

CHAPTER 1

Introduction

An emerging trend in the field of authentication on mobile devices is biometrics. The release of Apple's iPhone 5S, with its implementation of *Touch ID*, strongly contributed to the adoption of fingerprint authentication by the general public. By March 2015 over 18 million of these phones have been sold to end users [1]. The total number of mobile devices incorporating a fingerprint scanner is expected to reach 990 million in 2017 [2], and over 50% of all smartphones are expected to have a fingerprint sensor by 2019 [3].

The adoption of the fingerprint as a standard means of authentication on mobile devices, can be attributed to the fact that (on average) users prefer this method over traditional PIN [4, 5]. It is suggested that fingerprints should be used in mobile payments instead of PINs [6]. This approach is now incorporated by Google Play, in which users can use their fingerprint to authorise payments [7].

As the adoption and capabilities of fingerprint authentication increases, security becomes an increasing factor as well. Prior to the release of Android 6.0, hardware vendors used their own fingerprint recognition implementation, this development has led to several security vulnerabilities [8]. In an effort to improve security, Android 6.0 standardises support for fingerprint authentication. Its specification outlines requirements which compliant devices must adhere to [9].

1.1 Contribution

In this paper we introduce a method of reviewing a new implementation of fingerprint authentication on mobile devices. We identify the components involved and use existing research provided by security experts and the scientific community to perform a thorough analysis of component interaction. Additionally we show methods to manipulate the authentication system's behaviour.

1.2 Research question

Our main research question is: *Is it possible to bypass Android 6.0's fingerprint authentication, by modifying its vendor-independent software components, or by tampering with their interprocess communication?*

To answer this question we investigate which software components play a key role in fingerprint authentication. Subsequently, we research how genuine software components can be replaced with malicious counterparts.

We will also investigate the components' Inter-Process Communication (IPC) and explore methods of manipulating the IPC data.

1.3 Related work

There is a considerable amount of research regarding fingerprints and attacks against fingerprint authentication. Demonstrated attacks range from hardware attacks, such as faking fingerprints [10, 11], to software-based attacks.

An example of a study on software-based attacks can be found in Y. Zhang *et al.* [8], where multiple vendor implementations of fingerprint authentication systems on Android prior to version 6.0 are assessed. Several vulnerabilities were discovered, some compromised fingerprints and enabled the researchers to perform authorised actions without authenticating.

As will be elaborated in the next chapter, the fingerprint authentication system consists of multiple components which use IPC to communicate. Software-based attacks on the IPC architecture of the Android operating system are described by Artenstein & Revivo [12]. They investigate on intercepting IPC traffic by injecting a library into an arbitrary process. A significant part of our research builds on their findings.

Despite the amount of research on fingerprint authentication, no comparable research related to Android 6.0's native fingerprint authentication has been published. Our research aims to contribute to this knowledge by investigating the implementation of fingerprint authentication on devices running Android 6.0.

1.4 Report structure

This first chapter of this report is the introduction, where the relevance of this subject and the research question are elaborated. In Chapter 2, we describe the software components involved in fingerprint authentication and introduce some essential concepts to the Android OS. Chapter 3 elaborates on the scope of this research and describes an outline of the research in general. In the fourth chapter we describe the findings of our research and the circumstances in which they were obtained. In Chapter 5 we will discuses the limitations of our results.

A summary of the most important limitations and an answer to our research question is provided in Chapter 6. In the seventh chapter we make several recommendations on how the risks of our proposed attacks can be mitigated. Finally, Chapter 8 and in the last chapter we will propose some future work.

We aimed to write a concise report on a comprehensive topic (Android 6.0 fingerprint authentication). Therefore, throughout this document, we refer to several essential concepts elaborated upon in the appendices. Readers unfamiliar with Android concepts such as Binder IPC and the *Keystore* may refer to the Appendix for more information.

CHAPTER 2

Authentication system

In this chapter we elaborate on the components involved in fingerprint authentication. Some important concepts, such as IPC in Android, are also briefly explained.

2.1 Software components

The fingerprint authentication system consists of multiple components. Components close to third-party applications (apps) have a higher abstraction level than components closer to the hardware. The relation between these components is shown in *Figure 2.1: Fingerprint authentication system [13]*.

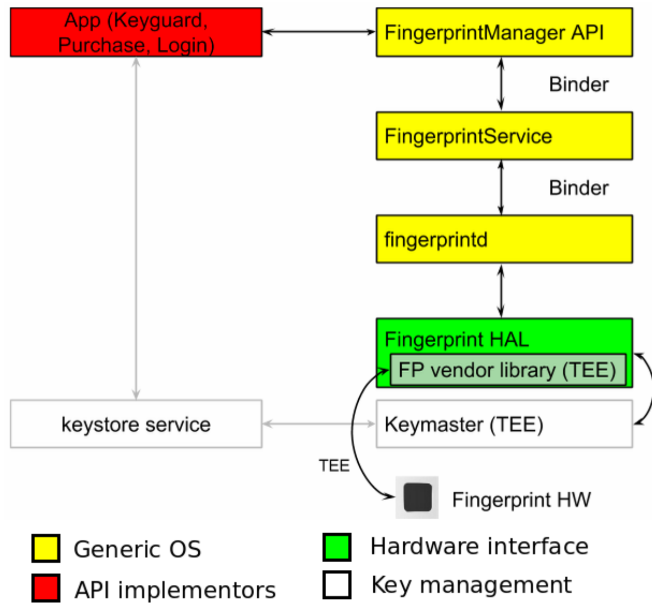

Figure 2.1: Fingerprint authentication system [13]

The component shown in red represent applications developed by third-party developers, these are the so-called "Apps". Shown in green is the Fingerprint Hardware Abstraction Layer (HAL), which acts as an interface to the hardware. The components shown in white are part of Android's *Keystore* and are used for key management. The components highlighted in yellow (`FingerprintManager API`, `FingerprintService` and `fingerprintd`) are part of the Android Open Source Project (AOSP) and are therefore independent of hardware vendors and third-party application developers.

FingerprintManager API Third-party developers can implement *FingerprintManager* to incorporate fingerprint authentication into their Apps. The API can then be used to authenticate users before performing certain actions.

Applications can also utilise the API in conjunction with the *Keystore* to authenticate cryptographic operations [14], this is described in detail in the appendix (*section A.3: Trusted Execution Environment*).

The *FingerprintManager* API wraps the functionality of *FingerprintService*.

FingerprintService This is a singleton service that runs as part of the system process [13]. Its main purpose is to handle communication with the fingerprint daemon (*fingerprintd*).

It also implements logic for additional abstractions. For instance, it ensures that third-party applications cannot distinguish individual fingerprints. This is a typical abstraction that is part of the Android 6.0 specifications [9].

fingerprintd The fingerprintd daemon runs as a separate process and communicates with the *Fingerprint HAL*. It is therefore closest to the hardware of all generic OS software components.

fingerprintd makes calls through the *Fingerprint HAL* to the vendor-specific library to perform operations including fingerprint enrolment, removal and authentication.

2.2 Fingerprint HAL

The *Fingerprint HAL* is a hardware vendor-specific implementation. It is used to directly communicate with the fingerprint hardware (sensor).

The HAL defines a set of 9 functions that can be used to perform operations on fingerprint data stored in the Trusted Execution Environment (TEE) [13]. These functions can be used to enrol, remove and authenticate fingerprints.

Upon enrolment, fingerprints (or its minutiae templates) are stored in the TEE. Hence, they are stored separately from the Android OS. This prevents users with elevated privileges (such as the *root* user) from compromising fingerprint data.

2.3 Hardware-backed Keystore

The availability of a TEE offers an opportunity for Android devices to provide a hardware-backed *Keystore* service to the Android OS [15]. The *Keystore* consists of multiple components, two of which are [16]:

1. Keymaster (TEE)
2. Keystore service

As described in the appendix (*section A.4: Authentication-gated keys*), fingerprint authentication can be used to restrict the access of cryptographic keys to their authentic users. This feature is used to prevent users with elevated privileges (i.e. the *root* user) from accessing other users' private keys.

The *Keymaster* is a component of the *Keystore* that resides in the TEE. It is used in authentication-gated cryptography. For applications to communicate with the *Keymaster*, a communication channel from Android OS to the TEE is required, *Keystore service* is used for this purpose.

The *Keystore* uses authentication tokens (*AuthTokens*) to make key release decision, this means it may or may not allow an application to use certain cryptographic keys. *AuthTokens* are created when a user successfully authenticates utilising the TEE. *AuthTokens* are an important feature of the *Keystore* and subsequently an important topic of this research. An elaborate description of what *AuthTokens* are and how they are used to base a key release decision can be found in *Appendix C: AuthTokens*.

2.4 Inter-process Communication

Binder is a framework that is used for Inter-Process Communication (IPC) on Android. Between *FingerprintManager* and *fingerprintd* data messages are exchanged utilising Binder IPC.

It is also used to interact with the *Keystore* which makes use a special extension of the Binder library [16].

Basic knowledge of Binder is essential to understand a significant part our research. Readers unfamiliar with Binder may refer to the appendix. We provide a basic introduction to Binder in *Appendix B: Binder IPC*.

CHAPTER 3

Methodology

This chapter describes how the results were obtained. We elaborate upon the scope of this project, the test environment, and general outline of this research.

3.1 Scope

The Android 6.0 fingerprint authentication specification encompasses both hardware and software components [9], some being vendor-specific. To infer conclusions on fingerprint authentication in the Android 6.0 OS, we aimed to keep vendor-dependent components out-of-scope.

However, since all components are part of a coherent system, behaviour of one component can propagate to another. Therefore, vendor-dependent components could not entirely be ignored.

To accommodate for this, we performed our research on a Nexus device which is offered by the official Google store. Devices in the Nexus line can be considered Google's flagship Android products. We expect other vendors to use these devices as a reference model.

3.2 Test environment

The research was conducted on a rooted LG Nexus 5X running Android 6.0 "bullhead", build MDA89E with a Linux 3.10.73-gea58e70 kernel. Both the kernel and build are factory defaults for the Nexus 5X.

Rooting the device was done in a systemless manner. This means that no modifications were made to the `/system` partition during the rooting procedure. Moreover, the systemless approach keeps Security-Enhanced Linux (SELinux) running in enforcing mode [17]. More information on SELinux and how it is used to protect the Android OS can be found in the appendix (*section A.1: SELinux*).

Throughout our research, we used Android Debugging Bridge (ADB) to interact with the device. ADB can be used to invoke shell commands on the device.

We used Android's *Repo* tool [18] to obtain the Android 6.0 source code for build MDA89E, which resides in the *android-6.0.0_r12* source tree branch [19]. We used this code to identify critical fingerprint authentication parameters, and to build modified software components.

3.3 Research outline

We started this research by determining the role of each component with respect to fingerprint authentication. From this research we found that *fingerprintd* implements all logic on which *FingerprintService* and *FingerprintManager* are built, the latter two can be considered abstraction layers of *fingerprintd*. Therefore, we concluded that *fingerprintd* should be at the core of our research.

After determining we would focus on *fingerprintd*, an investigation was launched to establish which function parameters are critical for accepting or rejecting fingerprints as authentic. This required analysis of how values and messages are propagated through the fingerprint authentication system. Most of this research was done by reviewing AOSP source code and adding logging statements to authentication components.

In the final stage of our research, we made targeted modifications to `fingerprintd` and its IPC in order to circumvent fingerprint authentication and unlock encryptions keys from the *Keystore*.

CHAPTER 4

Results

During this research, we were able to fake a successful authentication attempt, while supplying a fingerprint that was not enrolled into the device. We also discovered that it is possible to perform replay attacks against the *Keymaster* in the TEE.

In this chapter we will present and elaborate on our results. We also present a case study to test our results on two distinct implementations of fingerprint authentication; one utilising the *Keystore* and one that does not.

4.1 False positive recognition

We found that fingerprint authentication can be circumvented by returning a non-zero fingerprint ID to *FingerprintService*. In this section we will elaborate on how this was accomplished.

4.1.1 Fingerprint ID verification

Whenever a fingerprint is recognised in an authentication attempt, the corresponding ID is sent to *fingerprintd* from the TEE.

Each fingerprint ID is uniquely associated with a fingerprint on enrolment. Observed fingerprint IDs did not show predictable patterns and appeared to be randomly generated. Enrolled fingerprints can have any ID except '0'. Fingerprint ID '0' is reserved, it is returned by the hardware to indicate that the fingerprint could not be recognised.

The Android OS has no knowledge of which fingerprints are enrolled into the TEE, nor which identifiers are associated with them. However, the OS needs some way of knowing whether a fingerprint was enrolled into the TEE or not.

Recall that returned fingerprint IDs are equal to '0' when a fingerprint could not be recognised. This implies that a fingerprint ID that is not equal to '0' may indicate a recognised fingerprint and thus a successful authentication attempt.

Source code analysis revealed that *FingerprintService* performs a check on fingerprint IDs received from *fingerprintd*. The check compares the received fingerprint ID to '0', if it matches the fingerprint was not recognised and authentication fails. However, if it did not match '0' it assumes that the fingerprint was recognised and authentication succeeds.

Finally, the authentication result is forwarded to the instance of *FingerprintManager* that initiated the authentication process. The application implementing *FingerprintManager* consequently performs an action based on the authentication result. For instance, Android's *LockScreen* will unlock the screen when a fingerprint is recognised and show an error message when it is not.

The activities performed during fingerprint authentication across the different components are visualised in *Appendix D: Authentication flow*.

4.1.2 Replacing fingerprintd

We can leverage the check in *FingerprintService* to fake a successful authentication by running a modified *fingerprintd*. In our experiment, the source code of *fingerprintd* was modified to always send a fingerprint ID that not equal to '0'. Consequently, in all authentication attempts *FingerprintService* assumes the user successfully authenticated and forwards the result to the target application.

Source modifications The fingerprint ID returned to *FingerprintService* can be altered by modifying the source code of the *fingerprintd* binary, which is open source and provided by AOSP.

In the `hal_notify_callback` function we substituted the variable holding the fingerprint ID (as returned by the TEE) with a literal value not equal to zero, for instance '42' as illustrated in *Listing 4.1: Callback to* FingerprintService.

```
callback->onAuthenticated(device,
    42, // msg->data.authenticated.finger.fid
    msg->data.authenticated.finger.gid);
```

Listing 4.1: Callback to *FingerprintService*

Installation After building the modified *fingerprintd* binary, we replaced the genuine binary (located at `/system/bin/fingerprintd`) with the modified version. Also, as Android is based on Linux, it we attributed the binary with the same privileges and owner as its genuine counterpart. The OS would automatically run the modified binary in its attempt to start the genuine *fingerprintd*.

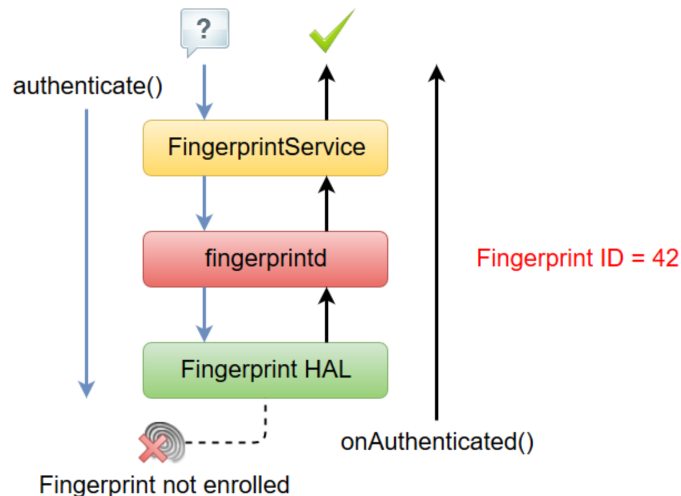

Figure 4.1: Modified fingerprint daemon

Exploitation The modified binary always returns the fingerprint ID '42' to *FingerprintService*. Since this is a non-zero value, authentication appears to applications to always succeed, even when the fingerprint was not enrolled into the TEE.

The effect of the modified binary is shown in *Figure 4.1: Modified fingerprint daemon*. In the diagram an application (implementing *FingerprintManager*) initiates an `authenticate()` call to *FingerprintService*, it is then propagated through *fingerprintd* and finally to *Fingerprint HAL* which interfaces to the TEE. The TEE would find that the fingerprint was not enrolled into the device and returns the fingerprint ID '0'. However, when the callback (`onAuthenticated()`) passes *fingerprintd* this value will be altered to something other than '0', in this case '42'. The *FingerprintService* would then indicate to the application that fingerprint authentication succeeded.

User warning As described in further detail in the Appendix (*section A.2: dm-verity*), dm-verity detects changes made on the `/system` partition. When such changes have occurred, dm-verity presents a warning message to the user when booting the device. The message will automatically disappear after five seconds.

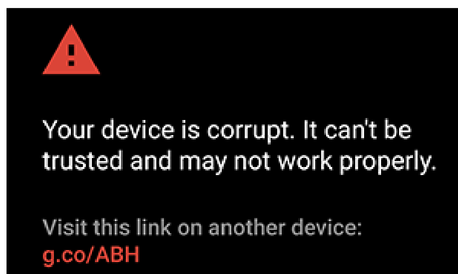

Figure 4.2: dm-verity warning [20]

Since the *fingerprintd* binary is located on the `/system` partition, dm-verity will detect that "the device is corrupt" and show the warning in *Figure 4.2: dm-verity warning [20]* during booting accordingly. Therefore, this attack is easily detectable. Security aware users may recognise this warning and understand that it might be caused by malicious intent.

4.1.3 Modified IPC

As described in the previous subsection fingerprint IDs are propagated between different components before reaching the target application. Since these components also run in separate processes they need a way to communicate, which is provided for by the Binder framework (see *section 2.4: Inter-process Communication*).

By manipulating the IPC between *fingerprintd* and *FingerprintService*, we were (again) able to fake a successful authentication attempt. Binder IPC is described in more detail in *Appendix B: Binder IPC*. Readers unfamiliar with its concepts may refer to it, as it introduces essential knowledge of understand some important results.

As discussed in the appendix (*section B.2: Interfaces*), Binder interfaces are the connecting element between two communicating processes. Source code analysis of *FingerprintService* and *fingerprintd* revealed that two separate interfaces are being used, one for authentication calls and one for its callbacks.

Capturing data Binder uses *Parcels* to exchange data among different processes. *Parcels* contain the values of arguments that were passed in the Remote Procedure Call (RPC). Since Binder IPC is used to communicate data from *fingerprintd* to *FingerprintService*, we targeted *Parcels* being transacted between these two processes.

Applications using Binder IPC implement the core Binder function library (`libbinder.so`). To intercept the *Parcels* being sent to *FingerprintService* (containing the fingerprint ID) we modified the source code of the `libbinder.so` library. Similar to *fingerprintd* the source code of the Binder library is also open source and provided for by AOSP.

We modified the Binder library's source code to hook on its `ioctl()` system call. The `ioctl()` system call is used to send and receive *Parcels* to and from the Binder driver. The hook function processes the buffer containing transaction data when sending and receiving *Parcels*. Data flowing to the driver (send) contain commands prefixed with "BC" (Binder Command) while data flowing from the driver (receive) is prefixed with "BR" (Binder Return Command) [12].

Since the fingerprint ID is contained in traffic sent to the Binder driver by *fingerprintd* to *FingerprintService*, we focussed on capturing data in `BC_TRANSACTION` commands.

Library injection After compiling the modified Binder library, we placed the modified library in a location readable (and executable) by the *system* user, which runs the *fingerprintd* process. In our experiment we used the *system* user's home folder at `/data/system/`.

Library injection was then staged by setting the `LD_PRELOAD` environment variable in a *system* user shell to include the modified library file. Through this shell a new *fingerprintd* process was started (by running the genuine binary) which then loads the modified Binder library (`libbinder.so`).

Afterwards, we were successful in intercepting all Binder IPC traffic flowing from and to *fingerprintd*.

From an attackers perspective, a benefit of this method is that files on the system partition are not modified. Therefore, the user will not be warned when booting the device, as was the case in the previous attack where we replaced the genuine *fingerprintd* binary.

Manipulating payload The target *Parcel* containing the fingerprint ID was located by applying a filter on the IPC data being transacted. We further extended the hook function in the modified Binder library to filter *Parcels*. The following filter conditions were used:

- Function code
- Interface descriptor
- Fingerprint ID

In our attempt to obtain the target *Parcel*, we filtered the *Interface Descriptor* and the *Function Code*. If the *Function Code* and descriptor matched the filter settings of the *Parcel* containing the fingerprint ID, we manipulated its argument values (i.e. the fingerprint ID).

The function that is called when the *Parcel* arrives is shown in *Listing 4.2: onAuthenticated function prototype*. This is the callback function to the *FingerprintService* that contains the fingerprint ID (`fingerId`).

```
status_t onAuthenticated(int64_t devId,
int32_t fingerId, int32_t groupId);
```

Listing 4.2: onAuthenticated function prototype

Within the *Parcel* data structure, the argument values follow the *Interface Descriptor*. By calculating the memory address of the `fingerId` variable in the *Parcel* we were able to write to its memory location. This enabled us to modify the fingerprint ID being sent to *FingerprintService*.

A high-level overview of Parcel data manipulation is shown in *Figure 4.3: Malicious IPC data*.

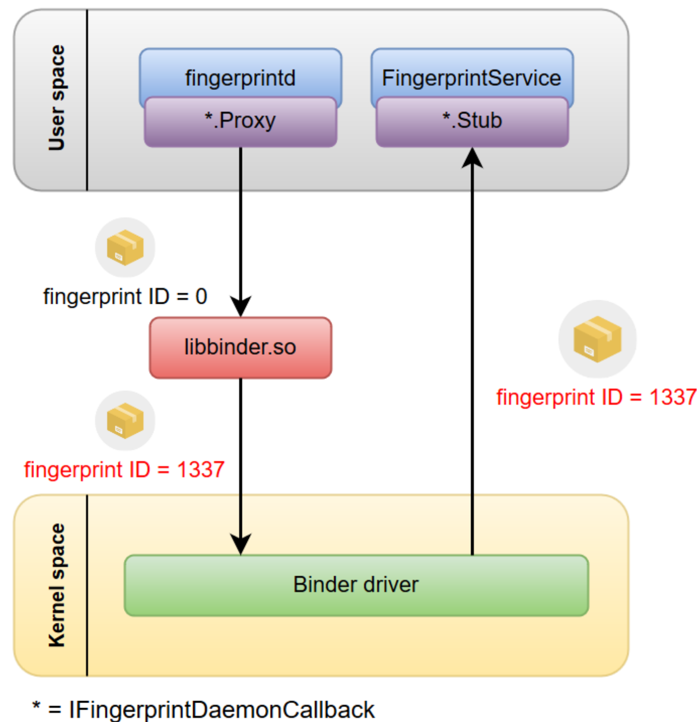

Figure 4.3: Malicious IPC data

4.2 Replaying authentication tokens

Authentication tokens (*AuthTokens*) are created in the TEE and send to the Android OS. This allows applications in the OS to use authentication-gated cryptography. This feature is used to prevent high privileged users (e.g. the *root* user) from accessing other users' private keys.

A more elaborate description of what *AuthTokens* are and how they are in authentication-gated cryptography can be found in *Appendix C: AuthTokens*. Basic knowledge of its concepts are required to understand the attacks discussed in this section.

Replaying authentication tokens (*AuthTokens*) to the *Keystore* is possible, despite the measures that should prevent this kind of attack. This behaviour can be abused perform authentication-gated cryptography using a replayed *AuthToken*. In this section, we demonstrate this attack by making changes to the *fingerprintd* binary.

4.2.1 AuthToken table

The *AuthToken* table is used by the *Keystore* to store authentication tokens and to keep track of requested authentication-gated cryptography operations (crypto operations). The amount of *AuthTokens* it may contain is bounded by the amount of crypto operations supported by the *Keystore*, this value is vendor-specific. A *Keystore* must support at least 15 crypto operations, as described in the Android implementers reference [21]. On our test device, 19 crypto operations were supported with IDs ranging from 1 to 19.

AuthTokens can be added and removed from the table. When the number of crypto operations exceeds the maximum value (e.g. 15), the least-recently used *AuthToken* is removed and replaced with the new *AuthToken*. The new *AuthToken* will be associated with the crypto operation ID of the *AuthToken* it replaced. This behaviour is shown in *Table 4.1: AuthToken challenge*.

4.2.2 Challenge-response

AuthToken may contain a challenge to prevent replay attacks. The challenge is specified as a "random" 64-bit integer, that is usually equal to the ID of a requested crypto operation [16]. If present, the AuthToken is valid only for crypto operations containing the same challenge.

Since the challenge is usually equal the ID of the requested crypto operation, there is a very small key-space for this challenge. For instance, with the specified minimum number of crypto operations (i.e. 15), only 4-bits of the entire 64-bit key-space are used.

Moreover, since crypto operation IDs are reused when an *AuthToken* is replaced in the table, the same challenge is also reused. This is caused by the fact that the challenge is equal to the crypto operation ID. This behaviour is shown in *Table 4.1: AuthToken challenge*.

The *Keymaster* expects a response to this challenge, meaning that the old *AuthToken* it replaced is equally valid. For instance, the second and last *AuthToken* from *Table 4.1: AuthToken challenge* will be both valid responses to the *Keymaster's* challenge '2', even though their timestamps differ. This behaviour be abused to perform replay attacks on the *Keymaster* in the TEE.

Crypto operation ID	AuthToken
ID = 1	AuthToken(challenge: 1, timestamp: 10000)
ID = 2	AuthToken(challenge: 2, timestamp: 11000)
ID = 3	AuthToken(challenge: 3, timestamp: 12500)
...	AuthToken(challenge: ..., timestamp: ...)
ID = 14	AuthToken(challenge: 14, timestamp: 19000)
ID = 15	AuthToken(challenge: 15, timestamp: 19500)
ID = 1	AuthToken(challenge: 1, timestamp: 24000)
ID = 2	AuthToken(challenge: 2, timestamp: 25000)

Table 4.1: AuthToken challenge

4.2.3 Retrieving the AuthToken

In order to perform a replay attack, a valid *AuthToken* must first be obtained. In our research, we obtained this token by replacing the genuine *fingerprintd* with a modified version.

As described in the appendix is more detail (*Appendix C: AuthTokens*), all *AuthTokens* pass *fingerprintd* on their way to the *Keystore*. The fingerprint *trustlet* in the TEE sends a message containing the *AuthToken* to *fingerprintd*.

We added logging statements to *fingerprintd* to print the contents of the message containing the *AuthToken*. The raw content of one of these tokens is shown in *Table 4.2: AuthToken capture*.

Field	Type	Value
AuthToken Version	1 byte	0
Challenge	64-bit unsigned integer	2
User SID	64-bit unsigned integer	6642721394326884821
Authenticator ID	64-bit unsigned integer[1]	13239196515636370186
Authenticator type	64-bit unsigned integer[1]	33554432
Timestamp	64-bit unsigned integer[1]	12838108872145108992
AuthToken HMAC	256-bit blob	243-169-20-......-57-7-76

Table 4.2: AuthToken capture

An overview of the meaning of all fields is described in on-line documentation [16]. However, for the purpose of our research we primarily focus on the challenge field. As can been seen from *Table 4.2: AuthToken capture*, the *AuthToken* contains a challenge with the value '2'.

It is important to note that used *AuthTokens* may reside in memory for some time and could consequently be obtained after they are used. In our research, we were able to retrieve used *AuthTokens* by reading from their memory addresses through modifying *fingerprintd*. The modified daemon printed the last used *AuthToken* regardless of the authentication attempt's outcome. The code snippet in *Listing 4.3: Capturing the AuthToken* shows how the *AuthToken* data is being read from the memory address.

[1] In network order (big endian)

```
if (true) { // Removed fingerprint authentication check
    const uint8_t* hat = reinterpret_cast<const
      uint8_t *>(&msg->data.authenticated.hat);
    printToLog(hat); // obtain old AuthToken
    instance->notifyKeystore(hat,
     sizeof(msg->data.authenticated.hat));
}
```

Listing 4.3: Capturing the AuthToken

Additionally, we were able to read old *AuthTokens* from memory, even if it was previously used by the genuine *fingerprintd*. After replacing the genuine binary with our modified binary, we successfully captured the last used *AuthToken*. Attackers can therefore retrieve used *AuthTokens* without having prior access to the device.

4.2.4 Replaying the AuthToken

To successfully replay a replay attack against the *Keymaster*, we must provide a valid response to its challenge. Therefore, we first need to force the *Keymaster* to provide a challenge for which we have a valid response. This can be done by requesting new crypto operations.

Every time a crypto operation is requested, a new *AuthToken* is generated, this will eventually lead to the reuse of a crypto operation ID in the *AuthToken* table and thus to the reuse of a challenge.

This behaviour is shown in *Appendix E: AuthToken challenges*, where `sid` is the challenge sent by the *Keymaster*. As can be observed from this log, the challenge '2' (of our captured *AuthToken*) is first used on 07:09:58 and later reused on 07:12:57.

To confirm the viability of replay attacks, we modified *fingerprintd* so that it would always provide the *AuthToken* from *Table 4.2: AuthToken capture* to the *Keymaster*.

In our experiments we observed that, if the *Keymaster* provided a matching challenge value '2', the *Keystore* would allow performing cryptographic operations authenticated by the replayed *AuthToken*. The modifications made to *fingerprintd* are shown in *Appendix F: Replay AuthToken code*. It was modified to always return the *AuthToken* as obtained in *Table 4.2: AuthToken capture*.

4.2.5 Exploitability

The exploitability of this replay attack depends on two factors:

1. Accessibility of the *AuthToken*
2. Validity of the *AuthToken*

In our experiments, we obtained a used AuthToken by reading the memory address where the message containing the *AuthToken* resides. However, the memory address will not hold the *AuthToken* indefinitely.

We observed a varying lifetime ranging from less than 10 minutes to over an hour. This is important as the time-frame in which a valid AuthToken can be obtained is crucial for exploiting replay attacks. Once an *AuthToken* has been captured it can be replayed for as long as it remains valid.

An *AuthToken's* validity is determined by its timestamp; a *Keymaster* property `KM_TAG_AUTH_TIMEOUT` can be set by hardware-vendors to specify when an *AuthToken* should time-out [21]. However, during our research we did not observe such a time-out. We were able to replay old *AuthTokens* as long as the test device did not reboot. Presumably this is due to the vendor setting `KM_TAG_AUTH_TIMEOUT` to the largest possible value, which translates to a validity period of approximately 136 years [22] (or until the device reboots).

4.3 Case studies

In order to observe the behaviour of modifications made to *fingerprintd* and the Binder IPC, we reviewed two applications implementing Android 6.0's fingerprint authentication. The two case studies highlight the difference in using the *Keystore* to verify authentication attempts.

4.3.1 FingerprintDialog

To familiarise application developers with implementing fingerprint authentication on Android 6.0, the OS developers provide the FingerprintDialog example application [23]. It demonstrates using fingerprint authentication to authorise a purchase as shown in *Figure 4.4: FingerprintDialog user interface [24]*.

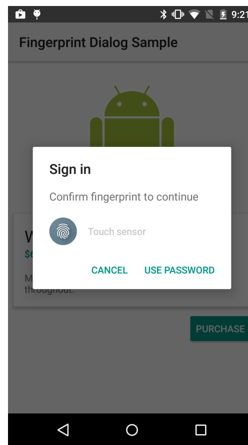

Figure 4.4: FingerprintDialog user interface [24]

On start-up the application generates a key which is stored by the *Keystore*. Access to the key is requested by the application after each authentication attempt to perform a cryptographic operation. Since cryptographic operations can only be performed by the *Keymaster* when a user successfully authenticates, the applications can verify the authenticity of this action. This implementation demonstrates the added security layer of the *Keystore* in authentication-gated cryptography, since it makes use of the TEE.

During the research project we utilised the FingerprintDialog example application to verify the behaviour caused by replacing *fingerprintd* with a "malicious" counterpart and manipulating the IPC. In both cases we were able to successfully authenticate with fingerprints that were not enrolled into the device. Moreover, by replaying AuthTokens to the Keystore, the application was able to perform cryptographic operations through *Keymaster*. This resulted in a "successful purchase" with an unauthentic fingerprint.

4.3.2 bol.com

In the final week of our research, the Dutch on-line retailer *bol.com* updated its application adding fingerprint authentication support [25].

The application performs authentication when a user is required to login, observable when editing personal details or when checking out.

By running a modified version of *fingerprintd* we were able to login successfully. This might be exploited to make purchases by unauthentic users.

The *bol.com* application does not use the *Keystore* to verify authentication attempts. Therefore, replaying AuthTokens to circumvent this authentication is not required. We were able to bypass authentication by only altering the `fingerprint ID` sent to *FingerprintService*.

CHAPTER 5

Discussion

In this chapter we will discuss our findings and elaborate on exploitability of the discovered vulnerabilities.

5.1 Vendor-dependent findings

In our research we performed all experiments on a Nexus 5X device. We found that fingerprint authentication can be circumvented and key release decisions by the *Keymaster* can be manipulated by performing a replay attack using captured *AuthTokens*.

The latter finding is dependent the hardware vendors' implementation of random challenges on *AuthTokens*. It can therefore not be concluded that every Android 6.0 device is vulnerable to this attack. However, as the official Android references state that the challenge is usually the ID of the requested crypto operation [16], it is likely that many vendors will implement this accordingly.

5.2 Feasibility of attacks

During our research distinguished two scenarios:

1. authentication alone is enough to perform the desired action;
2. cryptographic operations incorporating the *Keystore* are performed on top of authentication (authentication-gated cryptography).

Unlocking the Android *LockScreen* is a typical example of the first scenario, while as decrypting sensitive data is a typical example of the second scenario.

In both cases, modifying the *fingerprintd* binary can be used to achieve this goal. The main drawback of this approach from an attackers perspective, is that a warning message will be shown to the user upon booting the device. This might be detected by the user and is less suitable for attacks where the user retains his device (e.g. for planting backdoors).

However, to prevent warning messages from appearing, IPC traffic may be modified instead of the binaries on the `/system` partition. The disadvantage of this approach is that it may not be possible to perform replay attacks on *AuthTokens*. During our research we were not able to intercept the Binder *Parcel* that contains *AuthTokens*. This is probably because another, extended library is used for this purpose [16] which we did not investigate.

Both replacing the *fingerprintd* binary and manipulating IPC require elevated privileges (utilising the root user). This is the most important limitation to our discovered attacks.

	Replacing `fingerprintd`	Manipulating IPC
Requires root access	Yes	Yes
Shows user warning	Yes	No
Key release	Yes	Unknown

Table 5.1: Attack feasibility

It is a valid question to ask if authentication mechanism should restrain attacks against a user with those privileges. However, the TEE should not be affected when Android OS is compromised. Therefore, we argue that the *AuthToken* replay attacks should be prevented.

Table 5.1: Attack feasibility shows a comparison of the two attacks.

CHAPTER 6

Conclusion

In our research we investigated two methods to bypass Android 6.0's fingerprint authentication. In the first, we replaced the *fingerprintd* binary. In the second, we intercepted and modified IPC between the *fingerprintd* process and *FingerprintService* service.

We were able to bypass Android 6.0's fingerprint authentication and, perhaps more severely, we were able to perform replay attacks on *Keystore* components in the TEE. This enabled us to force the "release" of authentication-gated cryptographic keys without performing proper authentication.

Both replacing the *fingerprintd* binary and manipulating IPC require escalated privileges (utilising the root user). This is the most important limitation.

CHAPTER 7

Recommendations

This section will contain recommendations and mitigations towards parties involved in using or developing the fingerprint authentication system in Android 6.0.

7.1 OS developers

An Operating System should be a platform on which application developers can rely, whether is for correct memory management or adequate security measures. The android OS developers should therefore try to harden their security and improve the clarity of their implementer references.

7.1.1 dm-verity

In one of our attacks we replaced the *fingerprintd* binary. This triggered a warning message through dm-verity during booting. The warning message would disappear after five seconds and the device would boot like normal. We recommend to leverage this mechanism to require some user interaction before proceeding; for instance by pressing a button.

We also experienced that the warning message did not supply sufficient information to the user to determine what caused it, it merely showed that the system was corrupt. We recommend to make the warning more descriptive.

7.1.2 Unambiguous implementer references

Replay attacks against the *Keymaster* in the TEE are possible because a weak challenge is used. The implementer reference states the challenge is a random integer to prevent replay attacks, but at the same time states that it usually the ID of a requested crypto operation [16].

From these statements it could be deduced that the ID of a crypto operation must also be random, but as we have shown in our research, this is not how every vendor interpreted it. This is why we recommend Operating System developers to provide unambiguous implementer references and specifications.

7.1.3 Erase AuthTokens after use

During our experiments we found that the *AuthToken* sometimes remained in memory for over an hour, in other cases it was removed before 10 minutes. To mitigate the risk of used *AuthTokens* being obtained, we recommend to remove it from memory after it is used. This would mitigate the risk of replay attacks.

7.1.4 Only offer secure authentication methods

There are two fingerprint authentication methods used in Android 6.0:

1. authenticating only;
2. authenticating in conjunction with the *Keystore* (authentication-gated cryptography).

As described in *subsection 4.3.2: bol.com*, applications may use the first method where they should have ideally used the more secure second method, for instance when authorising payments. Although this decision may be considered the responsibility of application developers, we argue that an operating system should only offer the most secure authentication methods.

7.1.5 Protect Binder parcels' integrity

In our experiments we confirmed that the Binder parcels' integrity is not guaranteed. This enabled us to manipulate the IPC and thereby circumventing fingerprint authentication. We recommend that the integrity of these parcels are protected, or that they are encrypted as a whole.

There are already research studies on how this can be done [26].

7.2 App developers

Third party application developers use the fingerprint API offered by the Android OS to incorporate fingerprint authentication into their applications. They may use it to authenticate a user, or authorize sensitive actions.

Application developers should not trust rooted devices and use the *Keystore* to authorize sensitive actions.

7.2.1 Distrust rooted devices

We demonstrated methods to compromise fingerprint authentication in Android 6.0. These methods require root access to perform certain privileged actions. Applications developers should be aware that root privileges exceed normal user privileges, which may be abused for malicious intent. We recommend that application developers incorporate root detection software into their applications, and exit the application if privilege escalations is detected.

7.2.2 Use the Keystore

As discussed in *subsection 4.3.2: bol.com*, applications using fingerprint authentication to authorize sensitive actions (e.g. authorise payments) should use the *keystore*. The *keystore* makes use of the TEE which is not be affected when the Android OS is compromised. We recommend application developers to use this more secure authentication method.

7.3 End users

End users should be able to rely on the secure implementation of an OS as well as the applications they are using. However, some actions by the end users may weaken the security of the underlying OS and consequently the applications that run on it.

End users should be aware that privilege escalations (e.g. rooting) weaken the resilience of their device to certain attacks considerably. Users who want to use applications incorporating fingerprint authentication should not escalate their privileges on their device. Vice versa, users who want to use escalated privileges should not use applications incorporating fingerprint authentication.

CHAPTER 8

Future work

In this section we will propose future work that can be done build on or related to our research.

8.1 Keystore

During our research we found that the authentication-gated keys are an important security feature of fingerprint authentication in Android 6.0. An integral part of authentication-gated keys is the *Keystore*. We found that the *Keymaster* is vulnerable to replay attacks, however we did not perform an extensive research on the *Keystore* itself.

8.2 Gatekeeper

Like *fingerprint*, *gatekeeper* is a "trustlet" in the TEE that authenticates users. The *Gatekeeper* system performs authentication based on passwords an patterns. *Gatekeeper* is also used to protect authentication-gated keys, and offers similar functionality as fingerprint authentication. We assume it is likely that vulnerabilities found in our research also apply to *Gatekeeper*, however new research should provide a decisive answer.

8.3 Other fingerprint functions

In our research we focussed solely on the `authenticate()` function as offered by the *Fingerprint HAL*. However, there are several other functions available such as `enroll()`, `remove()` and `set_active_group()` that are worth investigating as well [13].

8.4 Fingerprint authentication resources

The fingerprint authentication components use resources on the Android OS. For instance, one of these resources is an encrypted database that seemingly stores some fingerprint data[1]. Furthermore, applications save their authentication-gated encryption keys on disk[2]. Investigating these resources may reveal vulnerabilities that have not yet been identified.

[1] located at texttt/data/system/users/{userID}/fpdata/user.db
[2] located at texttt/data/misc/keystore/{userID}/

8.5 Vendor specific (HAL) libraries

In our research, we focused on the vendor-independent software components used in fingerprint authentication and their IPC. We left the vendor-specific libraries almost entirely out-of-scope. However, it might be interesting to look into different implementations by distinct vendors. These libraries are close to the hardware and investigating these may reveal vulnerabilities we did not encounter during our research.

8.6 Cross-reference findings

All our tests were performed on an LG Nexus 5X device. Some findings may not apply to different vendors or even different devices by the same vendor. It might prove worthwhile to cross-reference our findings and provide a decisive answer if they apply and to what extent it affects other devices running Android 6.0.

Bibliography

[1] Static Brain Research Institute. *iPhone 5s Sales Statistics*. URL: http://www.statisticbrain.com/iphone-5s-sales-statistics/ (visited on 01/04/2016).

[2] *Fingerprint Biometric Market Intelligence*. Goode Intelligence, Dec. 2013. URL: http://www.goodeintelligence.com/media/report_link_files/1391518443goode_intelligence_fingerprint_biometric_market_intelligence_edition_one_december_2013.pdf.

[3] Research Capsule. *Fingerprint Sensors Market in Smart Mobile Devices 2012-2019*. MarketResearch.com, Mar. 1, 2015. URL: http://www.marketresearch.com/land/product.asp?productid=8918844&progid=87404.

[4] Shri Karthikeyan et al. *Smartphone Fingerprint Authentication versus PINs: A Usability Study*. Carnegie Mellon University, July 31, 2014. URL: https://www.cylab.cmu.edu/files/pdfs/tech_reports/CMUCyLab14012.pdf.

[5] Chandrasekhar Bhagavatula et al. *Biometric Authentication on iPhone and Android: Usability, Perceptions, and Influences on Adoption*. Carnegie Mellon University, Singapore Management University, Feb. 7, 2015. URL: http://www.internetsociety.org/sites/default/files/01_3_3.pdf.

[6] M. Gordon and S. Sankaranarayanan. "Biometric security mechanism in Mobile paymentts". In: *Wireless And Optical Communications Networks (WOCN), 2010 Seventh International Conference On*. Sept. 2010, pp. 1–6. DOI: 10.1109/WOCN.2010.5587318.

[7] Gadgets 360. *Google Play Gets Fingerprint Payment Authentication for Android 6.0 Marshmallow*. URL: http://gadgets.ndtv.com/apps/news/google-play-gets-fingerprint-payment-authentication-for-android-60-marshmallow-756275 (visited on 01/05/2016).

[8] Yulong Zhang et al. *Fingerprints On Mobile Devices: Abusing and Leaking*. FireEye Labs, Aug. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Fingerprints-On-Mobile-Devices-Abusing-And-Leaking-wp.pdf.

[9] Google inc. *Compatibility Definition Android 6.0*. Oct. 16, 2015. URL: http://static.googleusercontent.com/media/source.android.com/en//compatibility/android-cdd.pdf (visited on 01/05/2016).

[10] Chaos Computer Club. *Chaos Computer Club breaks Apple TouchID*. Sept. 21, 2013. URL: http://ccc.de/en/updates/2013/ccc-breaks-apple-touchid/ (visited on 01/05/2016).

[11] J. Galbally-Herrero et al. "On the Vulnerability of Fingerprint Verification Systems to Fake Fingerprints Attacks". In: *Carnahan Conferences Security Technology, Proceedings 2006 40th Annual IEEE International*. Oct. 2006, pp. 130–136. DOI: 10.1109/CCST.2006.313441.

[12] Nitay Artenstein and Idan Revivo. *Man in the Binder: He Who Controls IPC, Controls the Droid*. Malware Research Lab, Check Point Security, Sept. 2014. URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Artenstein-Man-In-The-Binder-He-Who-Controls-IPC-Controls-The-Droid-wp.pdf.

[13] Google. *Fingerprint HAL*. URL: https://source.android.com/security/authentication/fingerprint-hal.html (visited on 01/04/2016).

[14] Google. *Android 6.0 APIs*. URL: http://developer.android.com/about/versions/marshmallow/android-6.0.html#fingerprint-authentication (visited on 01/05/2016).

[15] Android Open Source Project. *Hardware-backed Keystore*. URL: https://source.android.com/security/keystore/index.html (visited on 01/29/2016).

[16] Android Open Source Project. *Authentication*. URL: https://source.android.com/security/authentication/index.html (visited on 01/28/2016).

[17] Androiding.how. *How to Root Marshmallow*. URL: http://androiding.how/root-marshmallow/ (visited on 01/26/2016).

[18] Android Open Source Project. *Downloading the Source*. URL: https://source.android.com/source/downloading.html (visited on 01/26/2016).

[19] Android Open Source Project. *Codenames, Tags and Build Numbers*. URL: https://source.android.com/source/build-numbers.html (visited on 01/26/2016).

[20] XDA Developers. *A Look at Marshmallow Root & Verity Complications*. URL: http://www.xda-developers.com/a-look-at-marshmallow-root-verity-complications/ (visited on 01/19/2016).

[21] Android Open Source Project. *Implementer's Reference*. URL: https://source.android.com/security/keystore/implementer-ref.html (visited on 01/30/2016).

[22] Android Open Source Project. *Features*. URL: https://source.android.com/security/keystore/features.html (visited on 01/30/2016).

[23] Android Open Source Project. *FingerprintDialog*. URL: http://developer.android.com/samples/FingerprintDialog/index.html (visited on 01/07/2016).

[24] Google. *Android FingerprintDialog Sample*. URL: https://github.com/googlesamples/android-FingerprintDialog (visited on 01/08/2016).

[25] bol.com. *De bol.com app voor Android ondersteunt bestellen met vingerafdruk*. URL: https://pers.bol.com/2016/01/de-bol-com-app-voor-android-ondersteunt-bestellen-met-vingerafdruk/ (visited on 01/25/2016).

[26] Yadu Kaladharan, Prabhaker Mateti, and K. P. Jevitha. "Intelligent Systems Technologies and Applications: Volume 2". In: ed. by Stefano Berretti, M. Sabu Thampi, and Soura Dasgupta. Cham: Springer International Publishing, 2016. Chap. An Encryption Technique to Thwart Android Binder Exploits, pp. 13–21. ISBN: 978-3-319-23258-4. DOI: 10.1007/978-3-319-23258-4_2. URL: http://dx.doi.org/10.1007/978-3-319-23258-4_2.

[27] Android Open Source Project. *Security Enhancements in Android 6.0.* URL: https://source.android.com/security/enhancements/enhancements60.html (visited on 01/28/2016).

[28] SysAdmin Magazine. *An Introduction to SELinux.* URL: http://www.crypt.gen.nz/papers/selinux_introduction.html (visited on 01/28/2016).

[29] Android Open Source Project. *Security-Enhanced Linux in Android.* URL: https://source.android.com/security/selinux/ (visited on 01/28/2016).

[30] Android Open Source Project. *Verifying Boot.* URL: https://source.android.com/security/verifiedboot/verified-boot.html (visited on 01/28/2016).

[31] Android Open Source Project. *Verified Boot.* URL: https://source.android.com/security/verifiedboot/ (visited on 01/28/2016).

[32] GlobalPlatform. *Trusted Execution Environment (TEE) Guide.* URL: http://www.globalplatform.org/mediaguidetee.asp (visited on 01/28/2016).

[33] Android. *Android Keystore System.* URL: https://developer.android.com/training/articles/keystore.html (visited on 01/28/2016).

[34] Thorsten Schreiber. *Android Binder. Android Interprocess Communication.* Ruhr Universität Bochum, Oct. 2011. URL: https://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf.

[35] Android. *IBinder.* URL: http://developer.android.com/reference/android/os/IBinder.html (visited on 01/29/2016).

APPENDIX A

Security enhancements

Android 6.0 introduces new security features [27]. In this appendix, we will discuss the new security features that are relevant to our research.

A.1 SELinux

SELinux is a kernel module that provides a mechanism for supporting access control security policies. Android uses SELinux to enforce Mandatory Access Control (MAC) over all processes, even processes running with root privileges. This means that if a process which is running as the root user is compromised (such as through a buffer overflow, etc.) then the damage is limited to what that process can access as defined by the policy [28].

SELinux has two operating modes:

1. enforcing - SELinux is enforcing the loaded policy, meaning that accesses which are not explicitly allowed cannot be performed.
2. permissive - SELinux is not enforcing the loaded policy, instead it logs policy violations. Processes can perform accesses that are not allowed by the policy.

Disabling SELinux is also possible, in this case no policy will be loaded.

SELinux was first introduced to android in version 4.3, where it was put into permissive mode for the entire system. With the introduction of Android 4.4, SELinux was partially put in enforcing mode, from version 5.0 Android moved to full enforcement of SELinux [29]. Android 6.0 introduces new, more strict SELinux policies.

During our research we kept SELinux in enforcing mode.

A.2 dm-verity

device-mapper-verity (dm-verity) is a verified boot mechanism which guarantees the integrity of the device software starting from a hardware root of trust up to the system partition [30]. It does this by using a cryptographic hash tree [31].

This capability is used to warn users of unexpected changes to critical software components, such as those on the */system* partition. Depending on what caused the verification to fail, users will be shown one of three warning messages. Users will always be given the option to continue using the device at their own discretion.

A.3 Trusted Execution Environment

The TEE is a secure area of the main processor in a smart phone. It ensures that sensitive data is stored, processed and protected in an isolated, trusted environment [32]. The TEE is separated from the rest of the device's hardware, meaning that Android OS cannot directly access the TEE.

The android 6.0 specifications demands that if a device implementation includes a fingerprint sensor, it must have a TEE, where it performs fingerprint matching [9]. Raw fingerprint data or derivatives (e.g. minutiae templates) must be stored in the TEE [13]. This means that rooting the device cannot compromise biometric data, since this is stored outside of the OS.

A.4 Authentication-gated keys

Android 6.0 also introduces the concept of user-authentication-gated cryptographic keys. These keys are only "released" by the keystore on a successful authentication attempt by the user. Applications can then use the keystore to perform cryptographic operations [33]. To achieve this, two key components need to work together. First is the cryptographic key storage, where secret keys are stored. Second are the user authenticators that may attest to the user's presence and successful authentication [16].

In Android 6.0 storing these cryptographic keys is facilitated by the `Keymaster`, while `Gatekeeper` (for PIN/pattern/password authentication) and `Fingerprint` (for fingerprint authentication) act as the user authenticators. These components reside in the TEE as is shown in *Figure A.1: Trusted Execution Environment*.

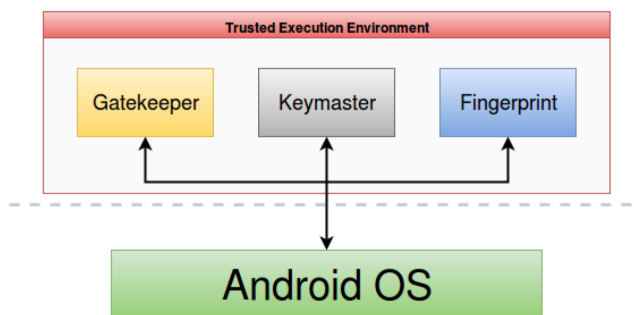

Figure A.1: Trusted Execution Environment

APPENDIX B

Binder IPC

Depending on implementation, it may be necessary for one process to access the memory of another process. On Android, an infrastructure for IPC is provided by the Binder framework.

B.1 Communication model

As an introduction to the concept of Binder IPC, we summarise its communication model.

The Binder framework implements the client-server architecture. During IPC, data flows between a *Proxy* (client) and *Stub* (server). Before sending data to a *Stub*, the *Proxy* serialises the data and stores it in a *Parcel*. Afterwards, the *Proxy* sends the *Transaction* (request) to the *Stub* which in turn processes the data and returns a *Reply* (response) [34].

B.2 Interfaces

A primary feature of the Binder framework is that it enables performing RPCs between two processes. However, both processes need to agree on a common interface first. A Binder interface not only defines object capabilities in the form of function prototypes but also enumerates these functions [12]. This allows the calling process to invoke a remote function by *Function Code*.

Using a Binder interface, developers can define a *Proxy* or *Stub* class which implements the interface. Creating an instance of such a class results in a Binder object. Usually Binder objects are published to the OS's *Service Manager* upon creation. This allows one process to query *Service Manager* for a reference to an object implementing a specific interface [34, 35].

This also applies in the case of an application wanting to use fingerprint authentication. Utilising the *Service Manager* (through *FingerprintManager*) to obtain a reference to the *FingerprintService* system service.

B.3 Serialisation and transmission

As previously mentioned in *section B.1: Communication model* IPC data is serialised before transmission in a *Parcel*. During serialisation the *Interface Descriptor* and function arguments are stored in a *Parcel*. Subsequently the *Parcel* is stored in a `binder_transaction_data` structure together with the *Function Code*. The final step is storing the transaction data in a `binder_write_read` structure which is used by the `ioctl` system call in writing the data to the target process [12].

Upon initiating the transaction, data is sent to the target process through the Binder Driver kernel module [34]. The basic data flow between user space and kernel is shown in Figure B.1.

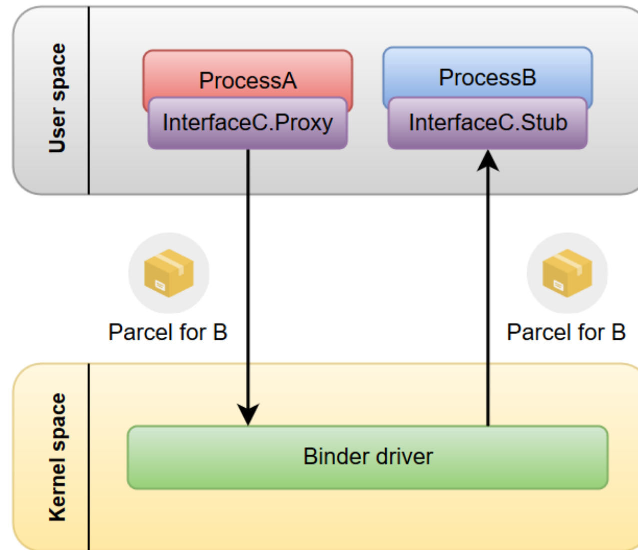

Figure B.1: Binder transaction

APPENDIX C

AuthTokens

Authentication tokens are used to grant applications usage of a certain key in the *Keystore*. *AuthTokens* are provided by the *fingerprint* or *gatekeeper trustlets* on a successful authentication attempt, these components reside in the TEE as shown in *Figure A.1: Trusted Execution Environment*.

AuthTokens are used to authenticate a user to the *Keystore* in an authentication-gated fashion. The *Keystore* can then be used to perform cryptographic operations on supplied data as described in A.4.

The integrity of *AuthTokens* is protected by a HMAC which is a keyed SHA-256 hash of all data fields in the *AuthToken* (except for the HMAC field). The key used to generate the HMAC resides in the TEE [16]. To prevent replay attacks, a challenge is included in the *AuthToken*.

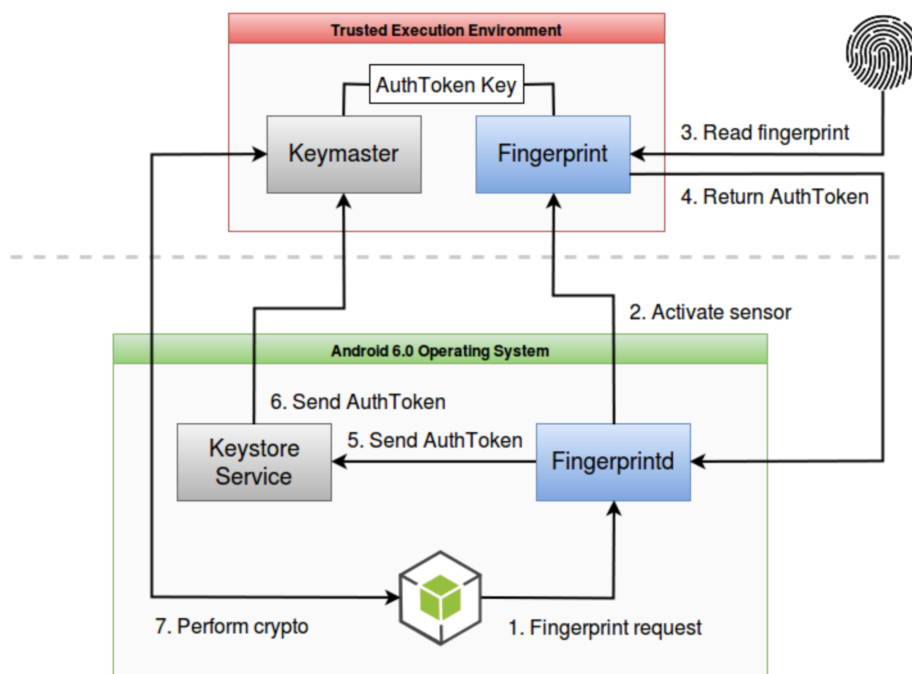

Figure C.1: Keystore AuthToken release

Figure C.1: Keystore AuthToken release illustrates how application can use AuthTokens to perform cryptographic operations using the *Keystore*:

1. An application wants to perform a cryptographic operation which requires the user to authenticate and forwards the authentication request to *fingerprintd*.
2. *fingerprintd* will inform its counterpart in the TEE about this request, which will subsequently activate the sensor and start listening for fingerprints.
3. The user places his finger on the fingerprint sensor. The fingerprint component in the TEE reads the fingerprint, and compares it with minutiae that have been enrolled in the past.
4. If there is a match, an AuthToken signed with the HMAC key is sent to *fingerprintd*.
5. *fingerprintd* cannot directly communicate with the *Keymaster*, therefore it forwards the AuthToken to *Keystore Service*.
6. *Keystore Service* subsequently forwards the *AuthToken* to *Keymaster*, which will recalculate the HMAC over all fields (except the HMAC field).
7. If the *AuthToken* is valid (i.e. the HMAC in the *AuthToken* matches the HMAC calculation of step 6) the *Keymaster* will allow the application to perform cryptographic operations.

APPENDIX D

Authentication flow

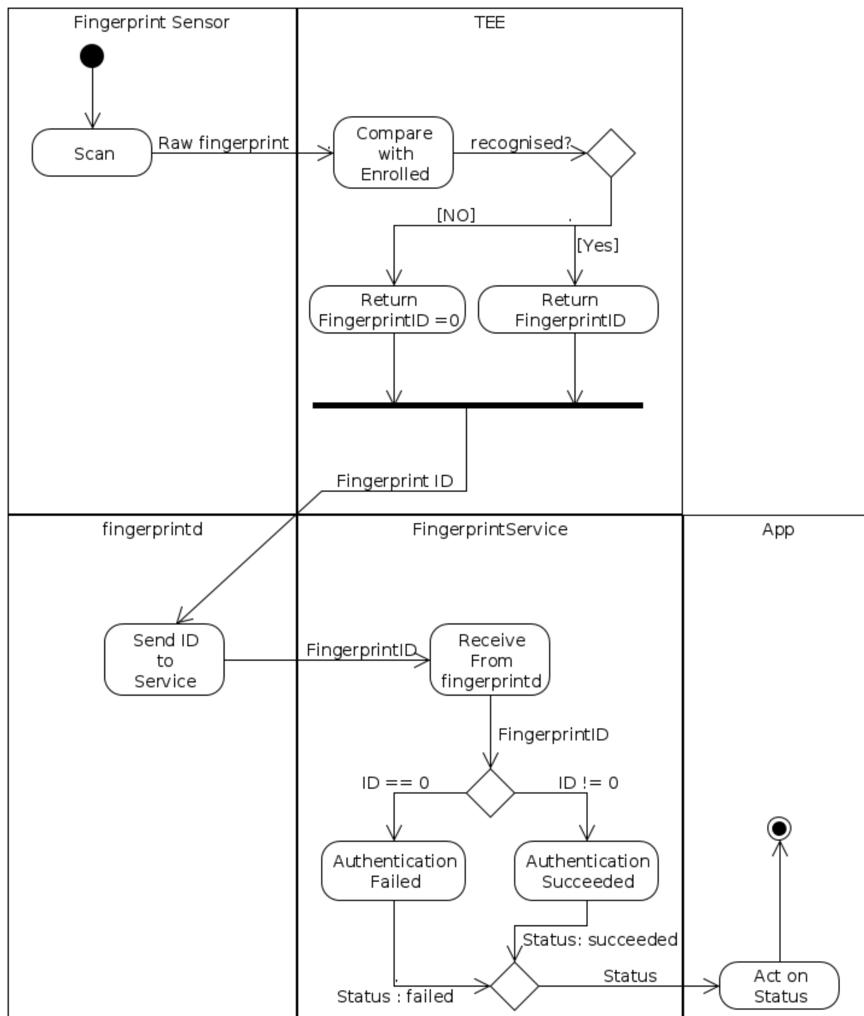

Figure D.1: Fingerprint authentication flow

APPENDIX E

AuthToken challenges

```
01-31 07:09:58.991 ... fingerprintd: authenticate(sid=2, gid=0)
01-31 07:10:01.943 ... fingerprintd: authenticate(sid=1, gid=0)
01-31 07:10:03.717 ... fingerprintd: authenticate(sid=18, gid=0)
01-31 07:10:04.733 ... fingerprintd: authenticate(sid=17, gid=0)
01-31 07:10:08.987 ... fingerprintd: authenticate(sid=16, gid=0)
01-31 07:10:41.734 ... fingerprintd: authenticate(sid=0, gid=0)
01-31 07:10:04.733 ... fingerprintd: authenticate(sid=17, gid=0)
01-31 07:10:08.987 ... fingerprintd: authenticate(sid=16, gid=0)
01-31 07:10:41.734 ... fingerprintd: authenticate(sid=0, gid=0)
01-31 07:12:05.191 ... fingerprintd: authenticate(sid=15, gid=0)
01-31 07:12:10.533 ... fingerprintd: authenticate(sid=14, gid=0)
01-31 07:12:13.274 ... fingerprintd: authenticate(sid=13, gid=0)
01-31 07:12:15.975 ... fingerprintd: authenticate(sid=12, gid=0)
01-31 07:12:18.682 ... fingerprintd: authenticate(sid=11, gid=0)
01-31 07:12:21.707 ... fingerprintd: authenticate(sid=10, gid=0)
01-31 07:12:24.744 ... fingerprintd: authenticate(sid=9, gid=0)
01-31 07:12:13.274 ... fingerprintd: authenticate(sid=13, gid=0)
01-31 07:12:15.975 ... fingerprintd: authenticate(sid=12, gid=0)
01-31 07:12:18.682 ... fingerprintd: authenticate(sid=11, gid=0)
01-31 07:12:21.707 ... fingerprintd: authenticate(sid=10, gid=0)
01-31 07:12:24.744 ... fingerprintd: authenticate(sid=9, gid=0)
01-31 07:12:37.927 ... fingerprintd: authenticate(sid=8, gid=0)
01-31 07:12:41.795 ... fingerprintd: authenticate(sid=7, gid=0)
01-31 07:12:44.471 ... fingerprintd: authenticate(sid=19, gid=0)
01-31 07:12:47.167 ... fingerprintd: authenticate(sid=6, gid=0)
01-31 07:12:49.920 ... fingerprintd: authenticate(sid=5, gid=0)
01-31 07:12:52.643 ... fingerprintd: authenticate(sid=4, gid=0)
01-31 07:12:55.317 ... fingerprintd: authenticate(sid=3, gid=0)
01-31 07:12:57.945 ... fingerprintd: authenticate(sid=2, gid=0)
```

APPENDIX F

Replay AuthToken code

```
if (true) { // Removed fingerprint ID non-zero check
    hw_auth_token_t fakehat;
    fakehat.version = (uint8_t)0;
    fakehat.challenge = (uint64_t)2;
    fakehat.user_id = (uint64_t)6642721394326884821UL;
    fakehat.authenticator_id = (uint64_t)1323919651563637018 6UL;
    fakehat.authenticator_type = (uint32_t)33554432;
    fakehat.timestamp = (uint64_t)12838108872145108992UL;
    fakehat.hmac[0] = (uint8_t)243;
    fakehat.hmac[1] = (uint8_t)169;
    fakehat.hmac[2] = (uint8_t)20;
    ...
    fakehat.hmac[29] = (uint8_t)57;
    fakehat.hmac[30] = (uint8_t)7;
    fakehat.hmac[31] = (uint8_t)76;
    const uint8_t* hat = reinterpret_cast<const
      uint8_t *>(&fakehat);
    instance->notifyKeystore(hat, sizeof(fakehat));
}
```

Listing F.1: Replaying the AuthToken