Exploring vulnerabilities in Android 6.0 fingerprint authentication

Thom Does & Mike Maarse

KPMG

02-02-2016

Introduction

Motivation/relevance

- Preferred authentication method by users
- Growing number of mobile devices with fingerprint hardware
  - 990 million in 2017 (Goode Intelligence)
  - Over 50% of all smartphones by 2019 (MarketResearch.com)
- Used to protect sensitive data/transactions
- Android 6.0 provides "native" support through API

Introduction

Motivation/relevance

- Preferred authentication method by users
- Growing number of mobile devices with fingerprint hardware
  - 990 million in 2017 (Goode Intelligence)
  - Over 50% of all smartphones by 2019 (MarketResearch.com)
- Used to protect sensitive data/transactions
- Android 6.0 provides "native" support through API

Research question

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?

Introduction

Motivation/relevance

- Preferred authentication method by users
- Growing number of mobile devices with fingerprint hardware
  - 990 million in 2017 (Goode Intelligence)
  - Over 50% of all smartphones by 2019 (MarketResearch.com)
- Used to protect sensitive data/transactions
- Android 6.0 provides "native" support through API

Research question

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?

The short answer...

Yes, in both cases!

Results

1. **False positive recognition**
   - Fingerprints not enrolled can perform authentication
   - ... or any capacitative body part (live demo)

2. **Forced release of authentication protected keys**
   - Allows attackers to perform cryptographic operations
     - Decrypt sensitive data
   - Attacks possible within vendor specific time-frame

Impact

- Determined by number of API implementations
- Compromises apps handling sensitive data
  - Financial transactions
  - Personal data

Case study bol.com

First large Dutch web shop to use fingerprint authentication

Observations

1. Triggers authentication on:
   - checkout
   - editing user profile
2. Trusts rooted device
3. Does not use the keystore

Figure 1: bol.com app dialog

Methodology - Equipment

Hardware

Figure 2: LG Nexus 5X

Software

- Android 6.0 "bullhead" (MDA89E)
- Android SDK platform tools

Methodology - Approach

- Explore the authentication system
- Analyse source code
- Replace software components
- Intercept and manipulate IPC

Goal

Forcing a successful authentication by returning a positive result code.

Software components

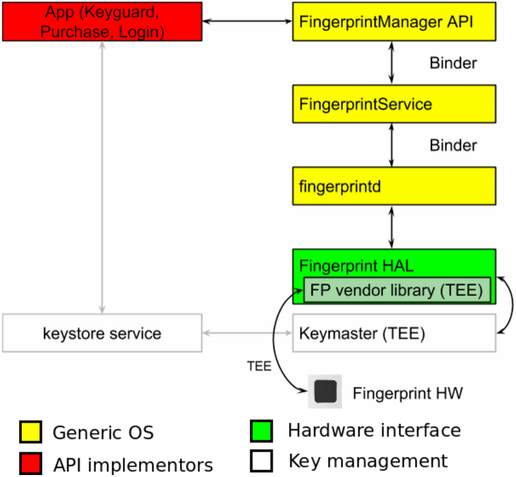


Figure 3: Fingerprint authentication software

Software components

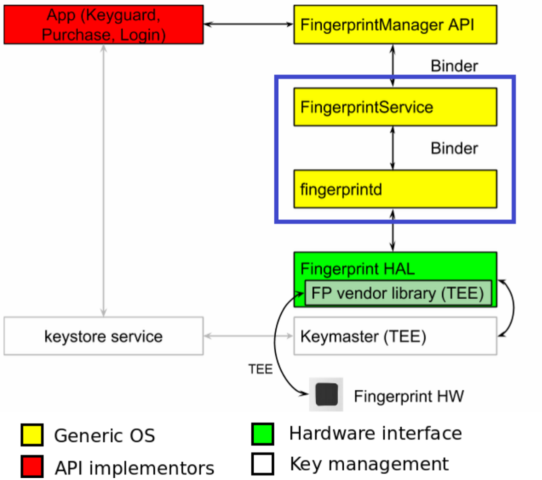


Figure 4: Communication components

Source code analysis

Finding the entry point...

- FingerprintService
  - Managed (Java) code
  - System service
  - Compiled as *.class
- `fingerprintd`
  - Native (C/C++) code
  - Separate process
  - Compiled as single executable

Source code analysis

FingerprintService checks return values

```
if fingerprint_id == 0
  return false
else
  return true
```

Problem?

No verification the fingerprint ID actually exists.

False positive recognition

Method I - Replacing fingerprintd

Fake fingerprint ID

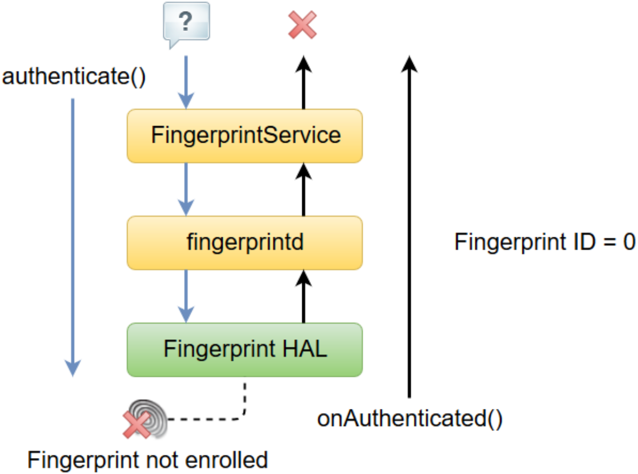


Figure 5: Result propagation

Fake fingerprint ID

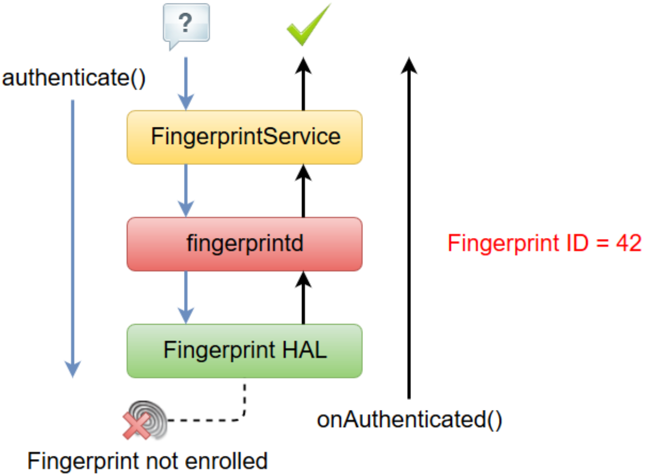


Figure 6: False positive

User warning

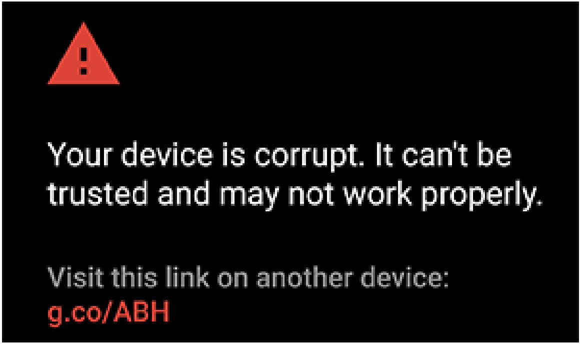


Figure 7: dm-verity warning

False positive recognition

Method II - Manipulating IPC traffic

Binder IPC

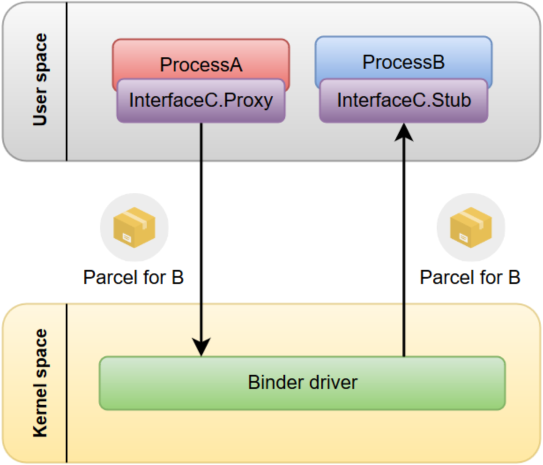
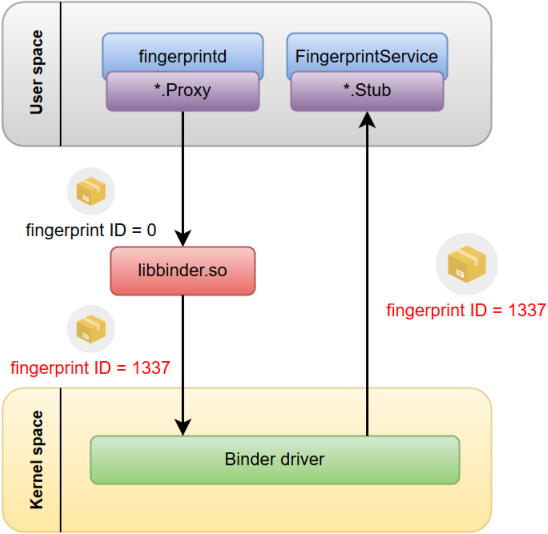


Figure 8: Binder transaction flow

Manipulating IPC traffic



* = IFingerprintDaemonCallback

Comparing attack methods

	Replacing `fingerprintd`	Manipulating IPC
Requires root access	Yes	Yes
Shows user warning	Yes	No
Key release	Yes	No[1]

Table 1: Method comparison

[1] Future work...

Forced release of authentication-gated keys

Key release

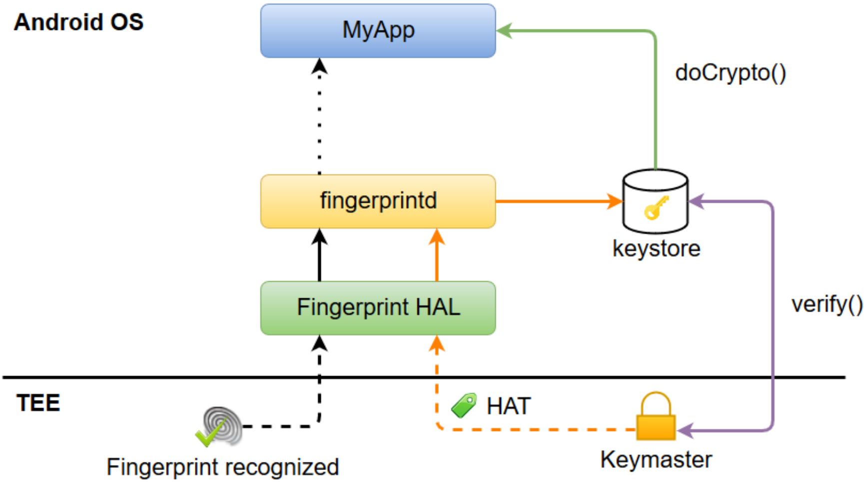


Figure 9: Keystore interaction

HAT replay

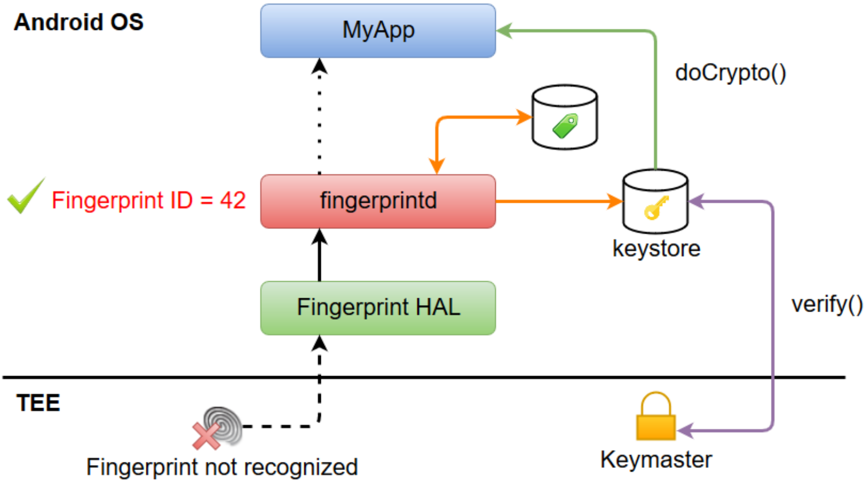


Figure 10: Replay attack

Challenge implementation

Hardware Authentication Token[2] security

- 64-bit "random" challenge...
- ...prevents replay attacks?

Problem?

Value of challenge equal to crypto operation ID [1..19].

[2] Also referred to as "AuthToken"

Attack feasibility

- Attacks only possible with root
- Can only be practically be exploited with physical access
- Might trigger warnings on start-up
  - But this can be circumvented using Binder

Mitigation

Application developers

- Use keystore
- Do not trust rooted devices

OS developers

- Randomise HAT challenge values (vendor's responsibility?)
- Erase HAT from memory after use
- Why offer less secure method?
- Protect Binder message integrity

End-users

- Do not use fingerprint authentication on rooted device

Questions?

FingerprintService.java

/frameworks/base/services/core...
/java/com/android/server/fingerprint/FingerprintService.java

```
if (fpId == 0) {
    if (receiver != null) {
        FingerprintUtils.vibrateFingerprintError(getContext());
    }
    result |= handleFailedAttempt(this);
} else {
    if (receiver != null) {
        FingerprintUtils.vibrateFingerprintSuccess(getContext());
    }
    result |= true; // we have a valid fingerprint
    resetFailedAttempts();
}
return result;
```

FingerprintDaemonProxy.cpp

/system/core/fingerprintd/FingerprintDaemonProxy.cpp

```
if (msg->data.authenticated.finger.fid != 0) {
    const uint8_t* hat = reinterpret_cast<const uint8_t *>(&msg->data.authenticated.hat);
    instance->notifyKeystore(hat, sizeof(msg->data.authenticated.hat));
}
callback->onAuthenticated(device,
        msg->data.authenticated.finger.fid,
        msg->data.authenticated.finger.gid);
break;
```

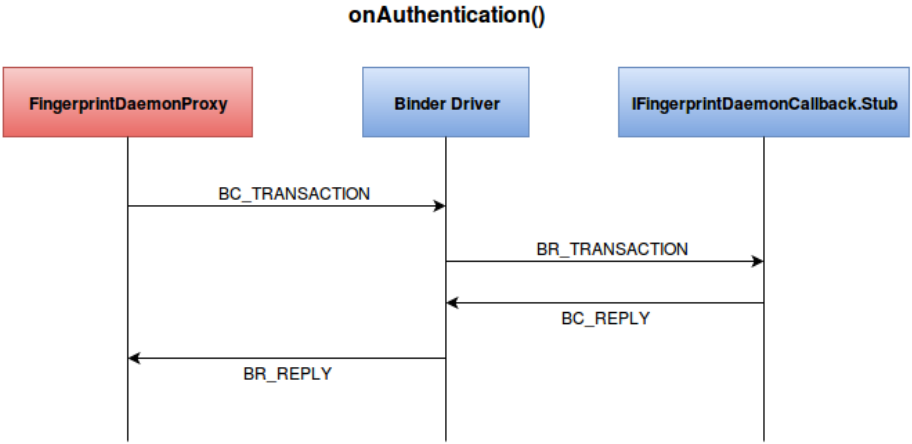
```
if (true) {
    const uint8_t* hat = reinterpret_cast<const uint8_t *>(&msg->data.authenticated.hat);
    instance->notifyKeystore(hat, sizeof(msg->data.authenticated.hat));
}
callback->onAuthenticated(device,
        1337,
        msg->data.authenticated.finger.gid);
```

Manipulating IPC traffic

Subverting the Binder

- Capturing IPC traffic
  - Library injection
  - Hooking IOCTL system calls
  - Dumping raw parcel data
- Manipulating parcel content
  - Select parcel by *Interface Descriptor* and *Function Code*
  - Retrieve memory address of IPC data from parcel
- Proves to be less detectable for end-users
  - No warning is triggered on start-up

IPC Traffic


onAuthentication()
FingerprintDaemonProxy
Binder Driver
IFingerprintDaemonCallback.Stub
BC_TRANSACTION
BR_TRANSACTION
BC_REPLY
BR_REPLY

HAT Data Structure

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[3]	13239196515636370186
Authenticator type	64-bit unsigned integer[1]	33554432
Timestamp	64-bit unsigned integer[1]	12838108872145108992
AuthToken HMAC	256-bit blob	243-169-20-223-...

Table 2: AuthToken capture

[3] In network order (big endian)

Challenge ID

Logcat output

```
07:12:05.191 ... fingerprintd: authenticate(sid=15, gid=0)
07:12:10.533 ... fingerprintd: authenticate(sid=14, gid=0)
07:12:13.274 ... fingerprintd: authenticate(sid=13, gid=0)
07:12:15.975 ... fingerprintd: authenticate(sid=12, gid=0)
07:12:18.682 ... fingerprintd: authenticate(sid=11, gid=0)
07:12:21.707 ... fingerprintd: authenticate(sid=10, gid=0)
07:12:24.744 ... fingerprintd: authenticate(sid=9, gid=0)
```