Exploring vulnerabilities in Android 6.0 fingerprint authentication

Thom Does & Mike Maarse

KPMG

02-02-2016
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

![LG Nexus 5X](image)

**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

```java
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

![Diagram showing result propagation]

**Figure 5:** Result propagation
Fake fingerprint ID

Figure 6: False positive
Figure 7: dm-verity warning
False positive recognition

Method II - Manipulating IPC traffic
Figure 8: Binder transaction flow
Manipulating IPC traffic

User space:
- fingerprintd
- *.Proxy
- FingerprintService
- *.Stub

fingerprint ID = 0

libbinder.so

fingerprint ID = 1337

Kernel space:
- Binder driver

fingerprint ID = 1337

* = IFingerprintDaemonCallback
Comparing attack methods

<table>
<thead>
<tr>
<th>Requires root access</th>
<th>Replacing fingerprintd</th>
<th>Yes</th>
<th>Yes</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shows user warning</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Key release</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Manipulating IPC</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Method comparison

\(^1\)Future work...
Forced release of authentication-gated keys
Figure 9: Keystore interaction
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?
```
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;
```
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
## HAT Data Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuthToken Version</td>
<td>1 byte</td>
<td>0</td>
</tr>
<tr>
<td>Challenge</td>
<td>64-bit unsigned integer</td>
<td>2</td>
</tr>
<tr>
<td>User SID</td>
<td>64-bit unsigned integer</td>
<td>6642721394326884821</td>
</tr>
<tr>
<td>Authenticator ID</td>
<td>64-bit unsigned integer³</td>
<td>13239196515636370186</td>
</tr>
<tr>
<td>Authenticator type</td>
<td>64-bit unsigned integer¹</td>
<td>33554432</td>
</tr>
<tr>
<td>Timestamp</td>
<td>64-bit unsigned integer¹</td>
<td>12838108872145108992</td>
</tr>
<tr>
<td>AuthToken HMAC</td>
<td>256-bit blob</td>
<td>243-169-20-223-...</td>
</tr>
</tbody>
</table>

Table 2: AuthToken capture

³In network order (big endian)
### Logcat output

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:12:05.191</td>
<td>fingerprintd: authenticate(sid=15, gid=0)</td>
</tr>
<tr>
<td>07:12:10.533</td>
<td>fingerprintd: authenticate(sid=14, gid=0)</td>
</tr>
<tr>
<td>07:12:13.274</td>
<td>fingerprintd: authenticate(sid=13, gid=0)</td>
</tr>
<tr>
<td>07:12:15.975</td>
<td>fingerprintd: authenticate(sid=12, gid=0)</td>
</tr>
<tr>
<td>07:12:18.682</td>
<td>fingerprintd: authenticate(sid=11, gid=0)</td>
</tr>
<tr>
<td>07:12:21.707</td>
<td>fingerprintd: authenticate(sid=10, gid=0)</td>
</tr>
<tr>
<td>07:12:24.744</td>
<td>fingerprintd: authenticate(sid=9, gid=0)</td>
</tr>
</tbody>
</table>