Pentest-Report OpenKeychain 08.2015

Cure53, Dr.-Ing. M. Heiderich, J. Horn, Dipl-Ing. A. Aranguren, Dr. J. Magazinius, D. Weißer

Index

- Introduction
- Scope
- Identified Vulnerabilities
  - OKC-01-001 Private Keys can be imported from Keyserver (Medium)
  - OKC-01-004 Arbitrary file write when decrypting and saving messages (High)
  - OKC-01-006 Keyserver can send arbitrary Public Keys without Verification (Low)
  - OKC-01-009 Bypassable Fingerprint-Check for Key Exchange via QR Code (High)
  - OKC-01-010 Database can be exported using Encrypt Operation (Low)
  - OKC-01-011 Unconfirmed Main Identities are shown as confirmed (Low)
  - OKC-01-012 Database Extraction possible via Version Downgrade (Medium)
  - OKC-01-013 Key Usage unchecked upon Decryption / Signature Verification (Low)
  - OKC-01-014 Multiple File overwrite Vulnerabilities via Path Traversal (High)
  - OKC-01-015 Export of PGP Information in clear-text on insecure Storage (Medium)
  - OKC-01-017 Predictable File Creation on insecure Location (Medium)
  - OKC-01-018 Key Server Verification Bypass via HTTP Redirect (Medium)
- Miscellaneous Issues
  - OKC-01-002 Malicious public Key can lead to persistent Denial of Service (Medium)
  - OKC-01-003 Malicious Key Server response can lead to Denial of Service (Low)
  - OKC-01-005 Insufficient and insecure RSA/DSA Key Sizes permitted (Medium)
  - OKC-01-007 Signing Operations with weak Key lead to Denial of Service (Info)
  - OKC-01-008 OpenKeychain accepts weak Passwords without any Warning (Info)
  - OKC-01-016 No Warnings when adding a clear-text HTTP Key Server (Low)
- Conclusion

Introduction

> *"OpenKeychain helps you communicate more privately and securely. It uses high-quality modern encryption to ensure that your messages can be read only by the people you send them to, others can send you messages that only you can read, and these messages can be digitally signed so the people getting them are sure who sent them. OpenKeychain is based on the well established OpenPGP standard making encryption compatible across your devices and operating systems."*
>
> From https://github.com/open-keychain/open-keychain

This penetration test and source code audit against the OpenKeychain mobile application took an entirety of twelve days and was performed by five testers of the Cure53 team. The test yielded an overall of eighteen issues, of which twelve were classified as vulnerabilities and six as general weaknesses.

The test was performed over a dedicated release tag created by the project maintainers in the public Github repository[1]. The Cure53 team audited the available sources and performed tests against the running application on both emulators and actual Android devices for maximum coverage. In addition to the core library, certain parts of the involved third party libraries were also audited. However, please note that only the relevant parts of libraries were analyzed. For instance, this applies to Spongy Castle[2], for which exclusively the parts that actually interact with OpenKeychain were examined.

The first and foremost conclusion of the test is that, on the one hand, none of the identified issues were classified as being of "critical" severity. On the other hand, three vulnerabilities were rated to be of a "high" severity. Quite telling is the fact, however, that these issues do not pertain to cryptographic flaws but rather to implementation-related issues. They occurred in connection to the handling of information during the process of key import and similar interaction with the app. This namely applied to the context of the external data being parsed and handled incorrectly, and thus could lead to an attack harming the device and users' security and privacy. Overall, the application presented itself as fairly robust in the core, yet requiring quite an array of smaller fixes, alterations and changes. The main focus at present should be to acquire a consistent level of security with regard to the import features, UI security and general file handling.

Scope

- **Sources made available via Github**
  - https://github.com/open-keychain/open-keychain
- **APK from F-Droid**
  - https://f-droid.org/repository/browse/?fdid=org.sufficientlysecure.keychain

[1] http://www.openkeychain.org/openkeychain-3-5/
[2] https://rtyley.github.io/spongycastle/

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *OKC-01-001*) for the purpose of facilitating any future follow-up correspondence.

OKC-01-001 Private Keys can be imported from Keyserver *(Medium)*

When a user attempts to download a public key from a keyserver, a rogue keyserver can instead send a private key inside a public key's armor. The app imports the private key without a warning and adds it to the private keyring. The method `ImportOperation.serialKeyRingImport()` invokes the method `HkpKeyserver.get()`, which then extracts the base64-encoded key from the keyserver's response.. This is done through testing for the presence of the Armor Header and Armor Tail lines and stripping them:

Affected Code:

```
Matcher matcher = PgpHelper.PGP_PUBLIC_KEY.matcher(data);
if (matcher.find()) {
    return matcher.group(1);
}
```

The base64-encoded key without Armor Header and Tail is then related to `UncachedKeyRing.decodeFromData()`, which is also used to parse private keys. This effectively causes the problem assessed and pointed out here. Consequently, it is recommended to verify that keys that are imported from a keyserver contain solely public keys.

OKC-01-004 Arbitrary file write when decrypting and saving messages *(High)*

PGP files can contain the name of the original unencrypted file. This feature is used by OpenKeychain in order to select a proper location when decrypting and finally saving a PGP file. Although value is not intended to contain path elements, OpenKeychain fails to perform any checks ensuring the prevention of possible file-overwrite-type attacks.

A malicious user - let us call him Bob - could craft an encrypted file with the *original filename* set to an arbitrary path of his choice on the file system. He could then send it to a second (regular, non-malicious) user we will call Alice. When Alice decrypts the file and clicks "OK" on the "Decrypt To File" dialog without choosing a location for storing the file, the decrypted file will be written to the PATH defined by Bob. The "Decrypt To File" dialog only shows the target file name but not the full path on Android < KitKat (namely its 4.4 version).[3]

The consequences of this issue are two-fold: a certain Denial of Service (e.g. by overwriting the key database), and a possibility for command execution. Devices running

[3] https://www.android.com/versions/kit-kat-4-4/

Android versions equals or newer KitKat are seemingly not affected due to the fact that they employ a different file manager.

Example Data:

- Password used here: a
- Original filename: `////////////////////////////////../../../../../../data/data/org.sufficientlysecure.keychain/databases/badfile`

Encrypted File:

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v2

jA0EBwMCtdLrYNpnR9rZ0nMBAPoOTWO+Q0RaYo22ZttlVdDL6gbu1xEVYoYOtsRj
lVWNuwc2FxSfHUFsY26DRx0Ho8JlONAyj6LxWbJ3XWn/7hR17t47d5AvqbgWNHbL
Aoq+OlPdsTFy8qeMi19HtyBVol5kGbPMBUnoS7Cv7FQyYlG+
=qA7c
-----END PGP MESSAGE-----
```

This occurrence can be explained with several following conditions. Firstly, when a user decrypts the file shown above, only a sequence of “/” is displayed as the filename. Secondly, when a user running Android versions older than KitKat selects “Save file”, he only sees “badfile” in the dialog. In case the user does not select a new download folder and simply presses “OK”, then the decrypted file contents will be written to the location chosen by the attacker. The existing files will be overwritten as a result.

Affected file: */keychain/ui/DecryptListFragment.java*

Code:

```
private void askForOutputFilename(Uri inputUri, String originalFilename, String
mimeType) {
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.KITKAT) {
            File file = new File(inputUri.getPath());
            File parentDir = file.exists() ? file.getParentFile() :
              Constants.Path.APP_DIR;
            File targetFile = new File(parentDir, originalFilename);
            FileHelper.saveFile(this, getString(R.string.title_decrypt_to_file),
                    getString(R.string.specify_file_to_decrypt_to), targetFile,
                     REQUEST_CODE_OUTPUT);
        } else {
             FileHelper.saveDocument(this, mimeType, originalFilename,
               REQUEST_CODE_OUTPUT);
        }
    }
```

The code above shows exactly the point where the issue produces the described vulnerability. Alongside, the following code demonstrates where the actual root of the problem is located:

Affected file: */keychain/pgp/PgpDecryptVerify.java*
Code:
```
String originalFilename = literalData.getFileName();
```

It is recommended to either extract the basename from the *original name,* or, alternatively to entirely reject encrypted files containing path elements.

OKC-01-006 Keyserver can send arbitrary Public Keys without Verification *(Low)*

Another one of the identified minor problems has to do with the process of importing keys from the keyserver. More specifically, in the first step, the keyserver is queried for a list of keys with the use of a search string. The server returns a list that describes the matching keys and those are then presented to the user. Once the user has made a selection of the keys he or she wants to import, these particular keys are promptly added to their OpenKeychain.

However, it is not validated that the data shown to the user about the key is consistent with the key that is actually imported. If the keyserver sent a fingerprint in its first response, the imported key is checked against that fingerprint, but that does not happen if the keyserver only sends a 64-bit Key-ID instead of a fingerprint.

It is recommended to check that the Key-ID and the identities of the imported keys are consistent with the result of the query.

OKC-01-009 Bypassable Fingerprint-Check for Key Exchange via QR Code *(High)*

When the keys are exchanged via QR code,[4] the expected fingerprint is contained in the QR code. The receiving user's device downloads the full key from the keyserver, performs a fingerprint comparison against the expected fingerprint, and, finally, asks the user to sign the identities associated with the key.

The fingerprint comparison is performed in `ImportOperation.serialKeyRingImport()` via a following check:

Affected Code:
```
// If we have an expected fingerprint, make sure it matches
if (entry.mExpectedFingerprint != null) {
    if (!key.containsSubkey(entry.mExpectedFingerprint)) {
        log.add(LogType.MSG_IMPORT_FINGERPRINT_ERROR, 2);
        badKeys += 1;
        continue;
    } else {
        log.add(LogType.MSG_IMPORT_FINGERPRINT_OK, 2);
    }
}
```

[4] https://en.wikipedia.org/wiki/QR_code

This is bypassable for two reasons outlined below:

1. This check is performed prior to the key being canonicalized by `mProviderHelper.savePublicKeyRing()`. Therefore, it is possible to bypass the check by adding the primary key of the expected key to the malicious key as a subkey. This can be done by extracting the first packet from the first key in a binary form and then changing the first byte (type field) from *0x99* to *0xB9.* Next, it needs to be turned it into a subkey, appending the result to the malicious key in a binary form and adding ASCII armor. OpenKeychain will accept the key because of the fingerprint match and will only be able to remove the fake subkey during the canonicalization process.
2. The method `UncachedKeyRing.containsSubkey()` matches the supplied fingerprint against all subkeys of the key. However, only when signing subkeys one requires a primary key binding certificate to confirm that the subkey was in fact created by the primary key's owner. Consequently, even if the key was canonicalized before the fingerprint check, the attacker could still bypass the fingerprint check by adding the expected primary key as a fake encrypt-only subkey, as long as it is signed with the subkey's binding signature.

For the mitigation purposes, it is recommended to require that the primary key matches the expected fingerprint. If that is not possible to implement for some reason, fingerprint matching should be restricted to subkeys with a valid primary key binding certificate. As a precaution, it should also be considered to initiate the canonicalization of the received key prior to performing the fingerprint comparison.

OKC-01-010 Database can be exported using Encrypt Operation *(Low)*

The threat scenario assumed for this issue is that an attacker gains access to the victim's unlocked android device and wants to efficiently bruteforce the password to their private key. Per the FAQ entry presented below, OpenKeychain attempts to block the extraction of the private key using software attacks in this scenario:

> *"Why is my password requested when I backup my keys?*
> *It is not required cryptographically, but prevents simple stealing of your keys."*

An attacker with ADB access or an ability to install the application on the device can open the `EncryptFilesActivity` ("Encrypt with OpenKeychain") with an arbitrary file:// URI. By starting `EncryptFilesActivity` with the URI `file:///data/data/org.sufficientlysecure.keychain/databases/openkeychain.db`, it is possible to encrypt OpenKeychain's internal database (including the encrypted keys stored within it). This can be done with the use of an attacker-chosen symmetric passphrase. Later, the result can be, for example, stored on external storage or exfiltrated with Android Beam.

To reproduce the issue, run the following command line on the device via *adb*.[5]

```
am start -a android.intent.action.SEND -t text/plain -n
"org.sufficientlysecure.keychain/.ui.EncryptFilesActivity" --eu
android.intent.extra.STREAM
file:///data/data/org.sufficientlysecure.keychain/databases/openkeychain.db
```

This should be followed by encrypting the file symmetrically and saving the result. The full attack could likely be carried out relatively fast. Firstly, open the settings, “security”, and tick “unknown sources”. Secondly, open Chrome, use the incognito mode, navigate to the attack helper APK, open the downloaded file, confirm permissions, and, last but not leas, press “open”. In the third step the attack helper APK opens and the dialog can be encrypted. Next set of actions requires the ticking of “encrypt with password”, typing a single character into both boxes, pressing the “share” button, and selecting the attack helper app. From then on the attack helper can successfully upload the database to some server, delete its own APK from the file system and request its own deinstallation afterwards. At this point one needs to confirm deinstallation. It is vital to note that the entire attack can be carried out in about one minute. This time already includes “cleanup” (closing Chrome’s incognito mode, removing the APK from “Downloads”, possibly some other actions), which would mean that the attack remains hidden from a cursory inspection.

Fixing this issue is somewhat difficult because the symlinks are allowed on the */data* file system. This means that an app cannot on the one hand follow symlinks created through *adb* - SELinux blocks that -, but, on the other hand, following symlinks that were created by other apps works.

For file:// URIs, if the ability to specify paths outside external storage is required, it is recommended to open the file at the given path and obtain a file descriptor to it (e.g. using the method `android.system.Os.open()`). One can then verify that the file referenced by the open `FileDescriptor` is world-readable. The method `android.system.Os.fstat()` can be used for that purpose, and then the file contents can be read through that file descriptor. As long as no secret world-readable files exist inside a non-world-executable folders owned by OpenKeychain, the attack will be prevented.

OKC-01-011 Unconfirmed Main Identities are shown as confirmed *(Low)*

Before moving on, please note that the term “main identity” used in the description of this issue description refers to the identity that is shown in the application’s main menu. Although this will usually signify the identity marked as primary, be aware that a key can equally have zero or multiple identities that are marked as primary. In consequence, this setting can cause a situation in which a non-primary identity is the main identity or one where a primary identity is not the main identity.

[5] http://developer.android.com/tools/help/adb.html

When a non-main identity in a key has been confirmed by the user, the key is shown as confirmed in the main menu under the main identity. This occurs even if the main identity has not been confirmed by the user. This can easily become confusing, especially in light of the fact that after the user has signed at least one identity of a key, a key sync adds a new identity as a main identity (e.g. by adding a primary identity to a key which does not have a primary identity or by marking the old primary identity as revoked). The user would see a new identity on the list of known keys. Interestingly that key would be marked as verified in spite of the fact that the user has never actually verified it.

It is recommended to either only mark the keys as verified if the main identity has been confirmed or, alternatively, to show the first verified identity (even if it is non-primary) for keys marked as verified.

OKC-01-012 Database Extraction possible via Version Downgrade *(Medium)*

Because of the issue described in OKC-01-011, signed release APKs of OpenKeychain exist. They essentially allowed the primary device user to dump the database without a password. What is important to report is that the fixing process of the above issue was not successful and persists in the newest version. In sum, when one was once given access to the device as the main user, it will remain possible to downgrade OpenKeychain to an older version via the command `adb install -r -d <apk>`. This will work as long as the old version was signed with the same key.

To guard against such attacks, it is recommended to add a version marker to the database (against downgrades between future versions) and modify the database layout. Another option would be to modify the file system structure so that it reliably crashes existing versions, for example by renaming the fields inside the `keyrings_secret` table.

OKC-01-013 Key Usage unchecked upon Decryption / Signature Verification *(Low)*

When a key is used for cryptographic operations, the declared key usage in the key flags of the used key is not checked. This allows an attacker, who has control over a decryption subkey, to create fake signatures. It further allows adding a foreign signing key to the attacker's own key as an encryption-only key, meaning that this key will be used for signature verification, thus bypassing the Primary Key Binding Signature requirement.

It is unclear whether this is a security issue in Spongy Castle or whether, perhaps, Spongy Castle intends for the caller to be responsible for checking the key flags, instead of doing it on its own.

OKC-01-014 Multiple File overwrite Vulnerabilities via Path Traversal *(High)*

The OpenKeychain app fails to sanitize the *export.log* filename when exporting a PGP import log. This takes place after a search for PGP keys on the cloud has been conducted. This could be abused by an attacker to fool users into pasting a text that overwrites the OpenKeychain database with the export log, effectively crashing the app and making the user lose all PGP information. As a result, all files and messages would be made undecryptable for the user. It is important to note that, as illustrated later in this ticket, the attack can also be triggered with the use of other field names, for example the backup field names.

This attack scenario seems plausible given that HTML web pages makes it possible for the users to see a given text while they are actually copying something else. For example, the following demonstration page superposes a given text which is not selectable, while the attack text is selectable and invisible. The attack will work even if the user selects and copies the whole page, simply using Cascading Stylesheets (CSS):

```
<html><body>
<div style="-webkit-user-select: none;"><h1>Just select aaaall this text and
copy paste it!</h1></div>
<div style="-webkit-user-select: text; z-index:10; opacity: 0; position:
absolute; top: 20px;"><font
size=4>../../../data/data/org.sufficientlysecure.keychain/databases/openkeychain
.db</font></div></body>
</html>
```

When the users visit the prepared page, they only see seemingly harmless text. When it is pasted, however, it actually pastes the attack, which is further obfuscated through the user-friendly text truncation on the mobile app:

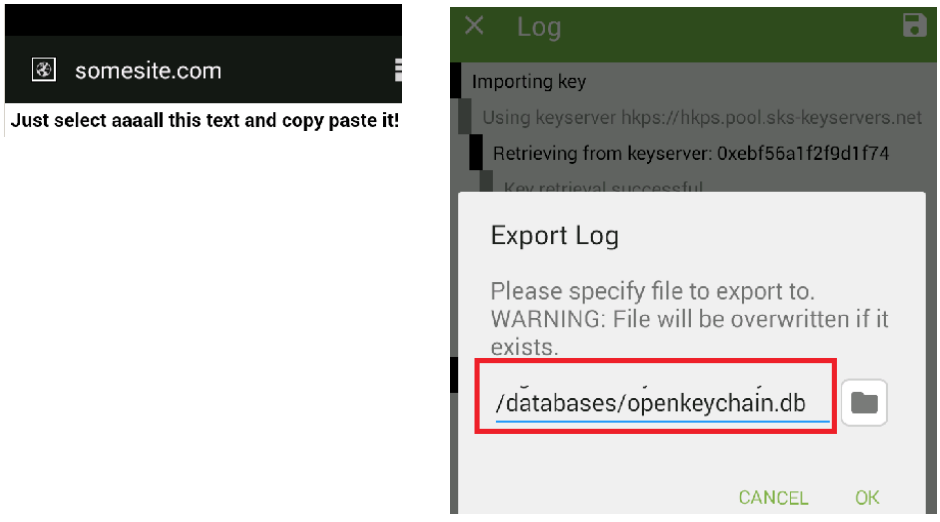

Fig.: The page displayed the user vs. the pasted text (truncated)

When the user taps on the OK button, the OpenKeychain app goes ahead and overwrites its entire database, eventually showing a message announcing success. As soon as the user taps away from that page, the OpenKeychain app crashes and, when opened again, it reports to have no keys. The three messages are demonstrated below:

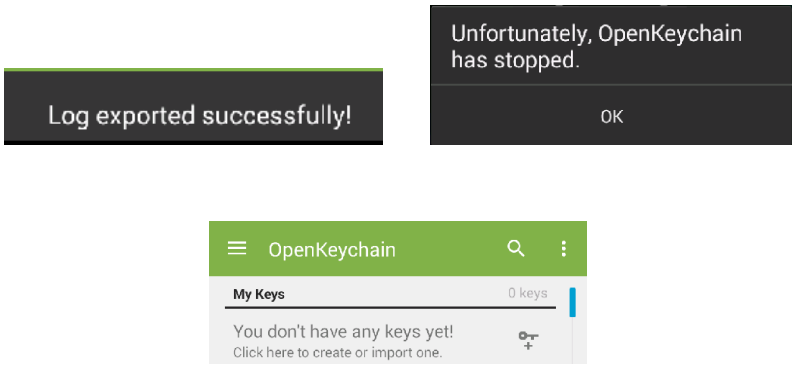

Fig.: OpenKeychain crash sequence after the user taps on the OK button

The same attack vector also works on the backup screen, but, in this case, it is not even necessary to tap away since the crash is instantaneous:

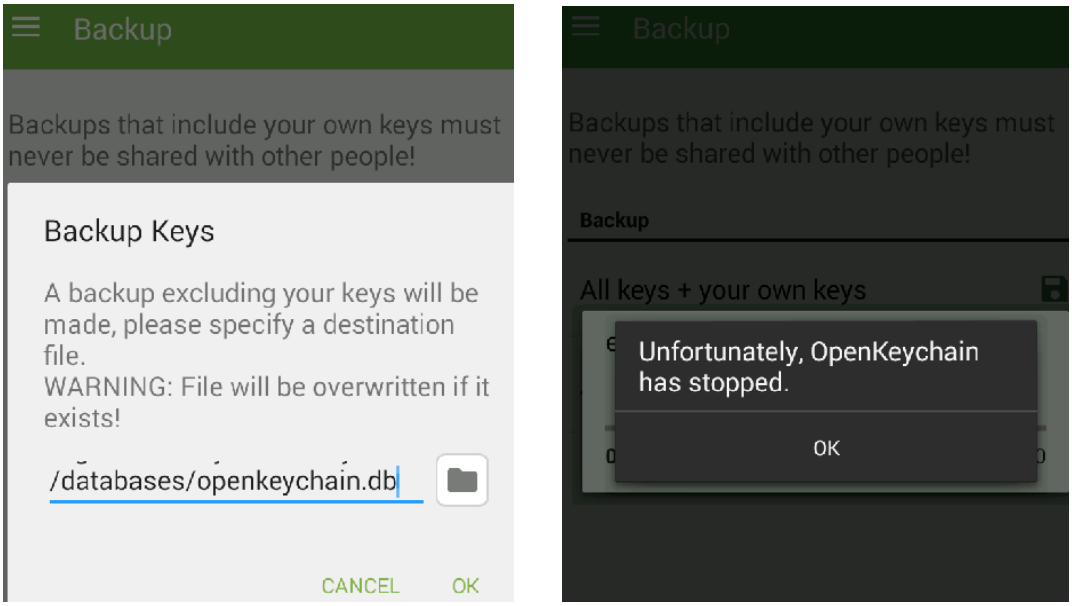

Fig.: Path traversal attack via backup functionality filename

Using the *export.log* functionality to overwrite the database results in an *openkeychain.db* file which has a file size identical to the intended *export.log* file:

```
$ ls -l
-rw------- u0_a54   u0_a54       2315 2015-08-28 16:24 export.log
-rw-rw---- u0_a54   u0_a54       2315 2015-08-28 17:35 openkeychain.db
```

Due to the above, it would seem that overwriting the database using a backup would, at the very least, make the information recoverable. However, this is not the case because the backup is generated on the fly as the app reads from the database. Hence, crashing at the start occurs and results in a zero-size file *openkeychain.db* database. From now on all information becomes irrecoverable:

```
$ ls -l
-rw-rw---- u0_a54   u0_a54        0 2015-08-28 19:38 openkeychain.db
```

In order to avoid this problem, it is recommended to validate all user-supplied filenames by implementing as many of the following countermeasures as possible:

- Only the most restrictive list of characters should be allowed, for example: accept `|a-zA-Z\.|` and reject everything else.
- ../ sequences and, in particular, '/' characters should not be allowed.
- The filename length should be restricted. Consider rejecting file names longer than 30-40 characters, as this would prevent writing into the internal app storage area (i.e. this attack required 76 characters).
- Concatenate the file extension to the filename provided by the user. For example, append ".log" to the value entered by the user (i.e. this attack required a ".db" extension).

OKC-01-015 Export of PGP Information in clear-text on insecure Storage *(Medium)*

When PGP keys are imported, the OpenKeychain app offers the possibility to save the import log. Provided that the user selects this option, the file will be saved in clear-text on the SD Card. The PGP fingerprints in the log expose the identities of the PGP contacts. Further, the SD Card can be extracted and read from a mobile phone without even needing the phone to be unlocked. The severity of this problem becomes much greater if the user takes advantages of the backup option, especially its particular variant which exports the private keys. In this latter scenario, all keys are saved to the SD Card in clear-text, without any encryption whatsoever.

This problem can clearly be a cause for concern across multiple scenarios. One example pertains to situations when a journalist or an activist has a device confiscated by the police in an oppressive regime. Here it is very probable that the exposed proof of a relationship between the device owner and certain individuals might result in criminal charges or even bodily harm in some edge cases. Moreover, a malicious app or corrupt police force members could fabricate false evidence (i.e. fingerprints are public) to get somebody in trouble, simply writing the right data onto the SD Card. Finally, a malicious application with permissions to navigate the SD card could leverage this weakness to determine who the PGP contacts of the device's owner are. Accordingly, this would grant a capacity of relaying this crucial information to a third party.

In order to solve this problem it is recommended to display a clear warning to OpenKeychain users at the time when they select the option to save the PGP import log. Presently, the user interface fails to clearly inform the user about the fact that the import log is saved to the SD card. Similarly, no mention can be found of the fact that this action is highly insecure, leaking PGP contacts and possibly having even further privacy problems that need to be considered. Quite evidently, if the mechanism is to be kept, then the data that must be exported to the SD Card should at least be encrypted with a strong password and not passed over in clear-text, especially when it might contain private keys.

OKC-01-017 Predictable File Creation on insecure Location *(Medium)*

The OpenKeychain mobile app allows users to export a number of files to the SD Card. On all screen dialogs these filenames are defaulted to well-known values, such as *export.asc*, *import.log* or *priv_export.asc*.

Alongside other problems described in this report, formatting the SD Card with the use of an ext2-4 file system[6] lets malicious apps create symbolic links pointing to the OpenKeychain internal storage. While most SD Cards are formatted with a FAT file system[7] in use, there is evidence of some users actually utilizing ext file system for the formatting task of either the full SD Card or just its partitions.[8] [9] .

When the SD Card is formatted using an ext file system, a malicious app could, for example, create the symlinks presented below. In the next step, it would need to wait for the OpenKeychain's user to do an export, private-, or public-key backup. As long as one of these occurs, the OpenKeychain database will be overwritten and all information that it contains will be lost:

```
cd /mnt/sdcard/OpenKeychain
ln -s /data/data/org.sufficientlysecure.keychain/databases/openkeychain.db
export.asc
ln -s /data/data/org.sufficientlysecure.keychain/databases/openkeychain.db
import.log
ln -s /data/data/org.sufficientlysecure.keychain/databases/openkeychain.db
priv_export.asc
```

In order to solve this problem it is recommended to avoid writing to a file when the file exists.

[6] https://en.wikipedia.org/wiki/Extended_file_system
[7] https://en.wikipedia.org/wiki/File_Allocation_Table
[8] http://forum.xda-developers.com/showthread.php?t=2123862
[9] http://forum.xda-developers.com/showthread.php?t=1442729

OKC-01-018 Key Server Verification Bypass via HTTP Redirect *(Medium)*

The OpenKeychain app implements a poor validation mechanism when a new keyserver is added to the database. As it stands it simply checks if the connection could be created and attempts to pin the certificate if the URL starts with “https”. For example, the app comes pre-packaged to use *hkps://pgp.mit.edu* with a pinned certificate.

However, it is possible to add a new key server using clear-text *http://* (or *https* with a valid certificate) that redirects to *hkps://pgp.mit.edu*. The mobile app can therefore be fooled to connect to any server, believing that the key server is the very first URL visited. A malicious attacker can leverage this and trick the users into adding a fake server to the OpenKeychain database. He or she can then monitor all PGP searches - a task that will not pose any problems. Due to the way that the cloud searches are invoked, the only necessary precondition for having a working attack is to have a URL ending in “?”. This facilitates a trivial fake cloud server to appear later:

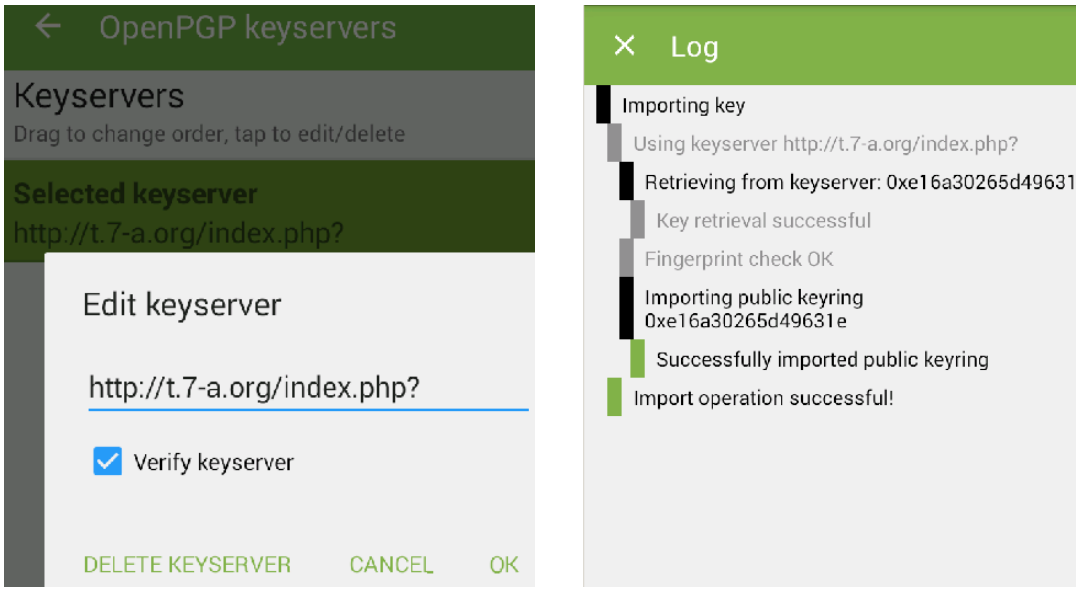

Fig.: Adding and importing keys with a fake key server

The fake server only needs to send a single HTTP header to the app. From then on the app will retrieve the information from there without a problem:

Example File: *index.php*
Code:

```
<?php
if (substr($_GET['search'], 0, 2) == '0x') {//Download public key
       header("Location: http://pgp.mit.edu/pks/lookup?
op=get&options=mr&search=" . urlencode((string) @$_GET['search']));
}
else {//Search
```

```
      header("Location: http://pgp.mit.edu/pks/lookup?
op=index&options=mr&search=" . urlencode((string) @$_GET['search']));
}
```

What is happening here pertains to the fact that the OpenKeychain mobile app is transparently following HTTP redirects for at least all of the following operations:

1. Key Server verification;
2. Key Server Public Key Search;
3. Key Server Public Key Download.

By default, the curl command-line utility has a set limit which permits following of up to 3 redirects. In comparison, the OpenKeychain mobile app will transparently follow up to 20 redirects without even issuing a warning. A consistency verification was successfully conducted with the use of the following fake server code for testing purposes:

Example File: *index.php*
Code:

```
<?php
$num = rand(1, 99999999);
header("Location: index.php?$num");
```

When searching for a PGP user, the access log trace from the OpenKeychain mobile app when visiting the above code looks as follows (i.e. 20 redirects followed):

```
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:27 -0400] "GET
/index.php?:11371/pks/lookup?op=index&options=mr&search=aranguren HTTP/1.1" 302
273 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:27 -0400] "GET /index.php?30914370
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:27 -0400] "GET /index.php?47259339
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:27 -0400] "GET /index.php?23638123
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:27 -0400] "GET /index.php?90428291
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:27 -0400] "GET /index.php?35458561
...
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:28 -0400] "GET /index.php?24057047
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:28 -0400] "GET /index.php?74711169
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:28 -0400] "GET /index.php?48491502
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
xxx.xxx.xxx.xxx - - [28/Aug/2015:22:58:28 -0400] "GET /index.php?46621610
HTTP/1.1" 302 272 "-" "okhttp/2.4.0"
```

In order to solve this problem, it is recommended to implement and provide additional verification checks. Some effort could be also invested in hopes of coming up with greater trust verification mechanisms. For example, the OpenKeychain mobile app could

download a list of trusted PGP servers over TLS and use Pinning from a trusted PGP server repository. After that, if the user attempts to add a key server that is not on the list, the user should be warned about the problem. This approach somewhat resembles the approach that is used by Psiphon.[10] In addition, if it is deemed feasible, it is recommended to at least disable the following of HTTP redirects which occur transparently. The recommendation applies to server verification, PGP search and PGP key download purposes.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

OKC-01-002 Malicious pubkey can lead to persistent Denial of Service *(Medium)*

OpenKeychain does not limit the length and amount of user-IDs in the keys. This allows attackers to create absurdly large public keys. If a large key is imported by the unaware user, the app crashes (OOM[11]) when the main screen (list of all keys) is displayed. Essentially, the victim-user ends up with a persistent Denial of Service. Attempting recovery without losing data can prove very difficult for inexperienced users, especially since resetting of the database will be required.

It is recommended to reject keys that exceed a realistic length or amount of user ids.

OKC-01-003 Malicious Key Server response can lead to Denial of Service *(Low)*

A malicious response from a key server can cause the application to crash. The result of a search contains various integers defining algorithm, keysize and timestamps. If the provided values are too large, then the app crashes due to missing exception handlers upon a conversion of the numbers to Integer/Long. A search response ships the user id of a key. A malformed UTF-8 string or faulty URL-encoding of this string also leads to a crash.

Example search result (convert-to-long crash):

```
pub:AAAA175EB0A516AC84DBBBBBB:117:2048:265985242345345345345535::
uid:asd <asd@asd.asd>:1265985242::
```

Example search result (bad url encoding):

```
pub:AAAA175EB0A516AC84DBBBBBB:117:2048:1265985242::
uid:bad%user <asd@asd.asd>:1265985242::
```

Affected file: */keychain/keyimport/HkpKeyserver.java*
Code:

[10] https://psiphon.ca/
[11] https://en.wikipedia.org/wiki/Out_of_memory

```
while (matcher.find()) {
	...
	int bitSize = Integer.parseInt(matcher.group(3));
	int algorithmId = Integer.decode(matcher.group(2));
	...
	final long creationDate = Long.parseLong(matcher.group(4));
	...
	String tmp = uidMatcher.group(1).trim();
	...
		tmp = URLDecoder.decode(tmp, "UTF8");
}
```

If the key server delivers random data instead of a key, OpenKeychain cannot parse the result and returns null instead of a String at *HkpKeyserver::get*. This leads to a crash caused by a null pointer exception. It is recommended to check the value for null prior to calling any methods.

OKC-01-005 Insufficient and insecure RSA/DSA Key Sizes permitted *(Medium)*

OpenKeychain requires new RSA keys to be at least 1032 bits in size. The keys smaller than 2048 bits are considered insecure and should no longer be created. Despite a default value of 4096 it is nevertheless recommended to increase the minimum size to 2048. OpenKeychain also imports weak 512-bit keys without issuing a warning.

It is recommended to deny all operations involving RSA keys smaller than 1024 bits. If very insecure keys need to be supported for some reason, a confirmation dialog should be displayed. This warning mechanism should comprehensively inform the user about the problem. Furthermore it was discovered that OpenKeychain allows generation of DSA keys with less than 1024 bits. Newly generated DSA keys should be between 1024 and 3072 bits in size. It is recommended to evaluate whether compatibility with DSA keys smaller than 1024 bit is required. If it is deemed not to be the case, importing and using such keys should be denied, or, once again, a warning should be displayed as a bare minimum:

> *"We estimate that even in the 1024-bit case, the computations are plausible given nation-state resource"*[12]

OKC-01-007 Signing Operations with weak Key lead to Denial of Service *(Info)*

The OpenKeychain application crashes when a signing operation with a weak RSA key (<1024 bit) is performed. This is due to an uncaught exception which cannot process the key properly. This causes a crash and the OpenKeychain app is terminated.

[12] https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf

Crash Log:

```
Process: org.sufficientlysecure.keychain.debug, PID: 30884
    java.lang.IllegalStateException: unable to create signature
            at
org.spongycastle.openpgp.operator.jcajce.JcaPGPContentSignerBuilder$1.
getSignature(JcaPGPContentSignerBuilder.java:146)
            at org.spongycastle.openpgp.PGPSignatureGenerator.
generate(PGPSignatureGenerator.java:263)
            at org.sufficientlysecure.keychain.pgp.PgpSignEncryptOperation.
execute(PgpSignEncryptOperation.java:505)
```

The exception is thrown by Spongy Castle in the file `JcaPGPContentSignerBuilder.java:146`. The explanation for this is that the library rejects signing with weak keys. The calling code is located in the file `/keychain/pgp/PgpSignEncrypt-Operation.java:505`

Calling File: */keychain/*pgp/PgpSignEncryptOperation.java
Code:

```
if (detachedBcpgOut != null) {
   signatureGenerator.generate().encode(detachedBcpgOut);
} else {
   signatureGenerator.generate().encode(pOut);
}
```

It is recommended to not only implement an exception handler but also to display a proper error message instead of having the application run into a crash. Since keys that small are not very common, please keep in mind that operations on them should be generally avoided (as mentioned in OKC-01-005). Therefore, this issue is only presented with an “Info” tag - something to keep in mind.

OKC-01-008 OpenKeychain accepts weak Passwords without any Warning *(Info)*

The security of the private keys depends just as much on the physical security of the device as it does rely on the passwords chosen to protect the keys. In its current state, the OpenKeychain application accepts very weak passwords to be selected by its users.

A weak password for private keys does not really provide any reasonable security and is especially problematic on mobile devices with considerably weaker physical security. It is recommended to show a confirmation dialog in case the user is about to set a weak password. A user needs to be informed and educated about the risks connected with this action instead of being met with a salient approval of weak passwords.

OKC-01-016 No Warnings when adding a clear-text HTTP Key Server *(Low)*

The OpenKeychain app offers the possibility to add new key servers on the key server management screen. If the key server has a self-signed certificate, or a MiTM attempts to intercept communications with a fake certificate, the OpenKeychain app correctly rejects the connection and the Key server cannot be added to the database. However, when the key server has a clear-text HTTP URL, then no warnings are found to be displayed.

This is worrisome because using plain-text HTTP key servers would leak information about the PGP searches that the user makes over the Internet and public Wi-Fi. In order to mitigate the possible consequences stemming from this issue it is recommended to discourage users from adding plain-text HTTP key servers. Additionally, a warning should be included on the mobile app. Finally, a simple rejection of http URLs in new mobile app versions should be taken into consideration in the future.

Conclusion

A considerable amount of time was dedicated to this thorough and far-reaching test against the OpenKeychain app and some parts of the instrumented libraries. The findings can be briefly summed up to an overall eighteen vulnerabilities and weaknesses that have been uncovered. This rather high number is not caused by several cryptographic flaws but rather several smaller issues that can be abused by an attacker to harm users' security and privacy. Note that the attacker model assumed for this test also included a rogue keyserver and an attacker who has limited control over the network traffic that the victim would be sending and receiving.

While some of the issues are clearly actionable and need prompt response, other might be negotiated with the maintainer. Especially in the second section of the report one finds some problems that can be treated as lower priority, mainly because a successful working and realistic attack for these scenarios would have to involve moderate social engineering or preying on overly trustful users.

Still, the Cure53 team believes that the OpenKeychain should strive towards providing not only safe cryptographic implementation, but also a working, well-tested and secure-by-default user interface. This would surely benefit the less technology-savvy users, who seek to have safe experience that supports the detection of erroneous information and attacks involving social engineering. OpenKeychain has the potential to be a generally usable and user-friendly one-stop-shop for cryptographic communication purposes on Android devices. In order to do so, however, it needs to tackle the requirements for acting responsibly and leaving no gaps in the user-security in case some parts of the cryptographic process have been tampered with.

Aside from the issues mentioned in this report, the software makes a robust impression. More importantly, it appears to be well-maintained, which is definitely something that the OpenKeychain team should be praised for. As it has already been underlined, it needs to be reiterated that none of the spotted issues were considered to be of a critical severity in terms of security implications. The latter is a significant and impressive result for an app of this complexity and relevance.

Cure53 would like to thank Dominik Schürmann and the entire OpenKeychain Team for their excellent project coordination, as well as support and assistance, which greatly benefited our work before and during this assignment.