Android 7 File Based Encryption and the Attacks Against It

Ronan Loftus
ronan.loftus@os3.nl

Marwin Baumann
marwin.baumann@os3.nl

Research Project 1

University of Amsterdam

Supervised by:
Rick Van Galen
Ruben De Vries

January 2017

Abstract

Android users have been provided with some level of disk encryption since Android 3.0 “Honeycomb”. This is marketed as ‘Full Disk’ encryption (FDE). FDE allows users to encrypt their /data partition. The major problem with FDE is that after rebooting, multiple critical functions of the device are unusable without user interaction. File Based encryption (FBE) was introduced to overcome this issue as part of the release of Android 7.0 “Nougat” in August 2016. FBE allows different files to be encrypted with different keys that can be unlocked independently. This fixes the shortcoming of FDE and also allows for more fine grained control of what’s encrypted.

The security of FDE has been researched quite extensively. Due to its recent release FBE has not been studied. In this paper we elucidate the workings of FBE. We then catalogue some of the known attacks against Android FDE. For each attack we introduce how they function along with the Android specific mechanisms they use. Then we either reason about or practically apply these attacks to the most recent Android version. Over half of the attacks we test are still applicable for Android 7. Finally we provide some recommendations for how these attacks can be rendered obsolete.

Contents

1 Introduction

Since the release of Android 3.0 "Honeycomb" Google has provided what they market as 'Full Disk' Encryption (FDE) to encrypt the /data (also called "userdata") partition of an Android device. This functionality provides users with the means of encrypting their personal data. After powering on the device all data on the /data partition is inaccessible until the user has provided authentication credentials. The security of FDE has been intensively researched in multiple papers. Different attack scenarios like the Evil Maid Attack and the Cold Boot Attack have been shown to work against FDE [1].

The major problem with FDE is that after rebooting, multiple critical functions of the device are unusable without user interaction. This problem becomes evident as the device is unable to receive calls after an unexpected reboot, e.g. initiated by a software update [2]. File Based Encryption (FBE) was introduced to overcome this issue as part of the release of Android 7.0 "Nougat" in August 2016 [3]. FBE allows different files to be encrypted with different keys that can be unlocked independently. This fixes the mentioned shortcoming of FDE and also allows for more fine grained control of what's encrypted.

Knowledge about the effects of potential vulnerabilities in FBE are important for maintaining the confidentiality of user data while at rest. The perceived insecurity of Android in terms of data confidentiality has led to low adoption rates in business. This is one of the reasons that it is important for businesses with sensitive data on company phones to know about possible vulnerabilities in Android FBE.

It is unclear if FBE is vulnerable to the same attacks as FDE as it has not been extensively researched. However, most of the known attacks against FDE are side channel attacks. These attacks are mostly based on capturing user credentials and subverting core Android components. Because these attacks do not directly attack the cryptography, we hypothesise that some of the attacks feasible against FDE are also possible against FBE. In section 2 we outline some of the attacks that have been discovered against FDE in previous Android versions. We then give a brief outline of the attacks and why they may or may not be feasible. The aim of this study is to research the potential vulnerability of Android 7 FBE against currently known attack scenarios.

1.1 Research question

To test our hypothesis we answer the following research question.

- Is Android 7 FBE vulnerable to the same attacks as FDE in previous Android versions?

To answer our main research question we first answer the following questions.

- What are the currently known attacks against (Android) FDE?
- How do these attacks work against FDE?
- Are the attacks on FDE still applicable to FBE?

2 Related Work

Oliver Kunz [4] describes two possible types of brute force attack on FDE in Android 5.0: 'Online' and 'Semi-Offline'. These are variants on the usual exhaustive search. We assume this attack is still feasible against FBE in Android 7 because enumerating all combinations is always possible in theory.

The "Cold Boot Attack" was shown to be viable against encrypted Android smartphones by Müller et. al in 2012 [5]. This is attack is no longer possible against devices containing a Trusted Execution Environment (TEE) as the cryptographic keys are no longer stored in RAM. We therefore reason that it is still practical against devices without a TEE running Android 7.

In June 2016 Gal Beniamini showed that the Android keystore is not actually bound to the underlying hardware for devices using Qualcomm chips [6]. This makes it possible to extract the encrypted Disk Encryption Key and perform an offline brute force attack on the users authentication method. If FBE uses a similar "crypto footer" then this may be a viable attack.

The so called "Evil Maid Attack" has been known since at least early 2009 [7]. In relation to Android based devices it was first mentioned by Defreez in 2012 [8]. It was shown to be practical by Götzfried and Müller in 2014 [1]. We reason this is still a viable attack on FBE as it is based on capturing user input.

In 2014 it was shown by Artenstein et. al that it is possible to intercept "Binder" communications [9]. Binder is the method by which Inter Process Communication (IPC) occurs on an Android device. If the cryptographic keys are ever communicated between processes then this remains a viable attack. This attack does not work on a device with a TEE.

In 2016 Does and Maarse showed that there are a number of practical attacks that can be used to subvert fingerprint authentication [10]. If the user chooses a fingerprint as their method of authenticating to the device then these attacks may still be viable.

2.1 Contribution

In this paper we introduce an overview of the components involved with Android FBE. No academic research has been done into FBE before. Our research is a first step in providing more insight into this topic. Next, we show which attacks possible for FDE, are still applicable for FDE.

3 Methodology

This research was conducted using an LG Nexus 5X (codename "bullhead") device. The Nexus line of devices are considered the most close to pure Android. They run stock Android, so if there are issues with this device they are likely to propagate to devices from other vendors. We updated the device to Android 7.1.1. This device was running the 3.10.73 Linux kernel. The device had an unlocked bootloader. We also rooted the device and enabled Android Debug Bridge (ADB) for ease of testing.

We used the 7.1.1 revision 11 (build N4F26J) branch of Android on the device. When we began our research (early January 2017), this was the most recent build supported for the nexus 5X [11]. We also obtained the source tree from the official repository [12]. We used this as the most authoritative source of documentation. The source code was used to deduce the internal workings of FBE. We set up our build environment as described in the upstream documentation [13]. We then used the source to compile modified versions of various Android components. These modified binaries were then pushed to the phone using ADB.

4 Android Disk Encryption

In this section we give an introduction to the operation of both FDE and FBE. As FDE has been studied in depth already we do not give it a thorough treatment. We give FBE a more comprehensive treatment as it has been less studied.

For both encryption schemes we introduce what is encrypted on the device. We make note of the keys that are used in each encryption scheme. Furthermore, we elucidate the way in which the keys are used and how they are created. We also give an explanation of how the keys are stored on the device.

4.1 'Full Disk'

As previously mentioned, only the /data partition of a device is encrypted. This encryption scheme is based on the "dm-crypt" Linux kernel module. dm-crypt provides transparent encryption of a block device. On the fly, all data is encrypted before being written to disk and decrypted after being read from it.

Keys

The /data partition is encrypted as a single volume. Therefore, only one key is required to encrypt or decrypt it. This key is called the Disk Encryption Key (DEK). The DEK is a 16 byte (128 bit) sequence read from */dev/urandom* when the /data partition is first encrypted [14, line 1500]. The DEK remains constant until the /data partition is wiped. If the encryption scheme were implemented in this manner it would not be possible to change the decryption key without re-encrypting the whole partition. Therefore, this scheme is not used and a second key is employed called the Key Encryption Key (KEK).

This key is derived from the users authentication credentials. The KEK is then used to encrypt the DEK. The user can now change their authentication credentials at will without needing to re-encrypt the whole /data partition. Instead this will trigger a re-encryption of the DEK. The users credentials can be a PIN code or a password, since Android 5.0 "Lollipop" a pattern lock can also be used.

Modes of Operation

There are two modes of operation applied using Advanced Encryption Standard (AES) for FDE. The /data partition is encrypted with the DEK using AES-128 in Cipher Block Chaining (CBC) mode. The DEK is encrypted with the KEK using AES-128 in CBC mode. The /data partition is not encrypted as one single unit. If this was the case then any time a file is changed, the entire disk up until that point would need to be re-encrypted. To overcome this issue, the 128 bit DEK is used to encrypt/decrypt each sector separately.

CBC mode requires an Initialisation Vector (IV) for each sector. Reusing an IV with a given key can undermine the security of the cipher. There needs to be a deterministic mechanism for deriving a unique IV for each sector to encrypt the disk in this manner. This is achieved

using Encrypted Salt-Sector Initialisation Vector (ESSIV). The equation for computing the IV of a sector N is given by Equation 1.

$$IV(S_N) = E_{h(k)}(S_N) \tag{1}$$

Where S_N is the sector number, E is a block cipher, h is a hash function and, k is the DEK. The IV of a sector is its sector number encrypted using the hash of the master key as the encryption key.

Key Storage

The encrypted DEK is stored in a non-encrypted area of the device called the "crypto footer" (/metadata partition). The crypto footer stores the encrypted DEK, a 128 bit randomly-chosen salt, and some other parameters that determine how the /data partition was encrypted. As shown in Figure 1 the KEK is derived using a Key Derivation Function (KDF), which takes the user's unlock credentials (PIN/Password/Pattern) and the salt. The salt used to prevent pre-computation attacks as it diversifies user keys. The Android KDF uses the "scrypt" function as one of its core components. For an overview of scrypt and the KDF used in Android FDE see appendix A.

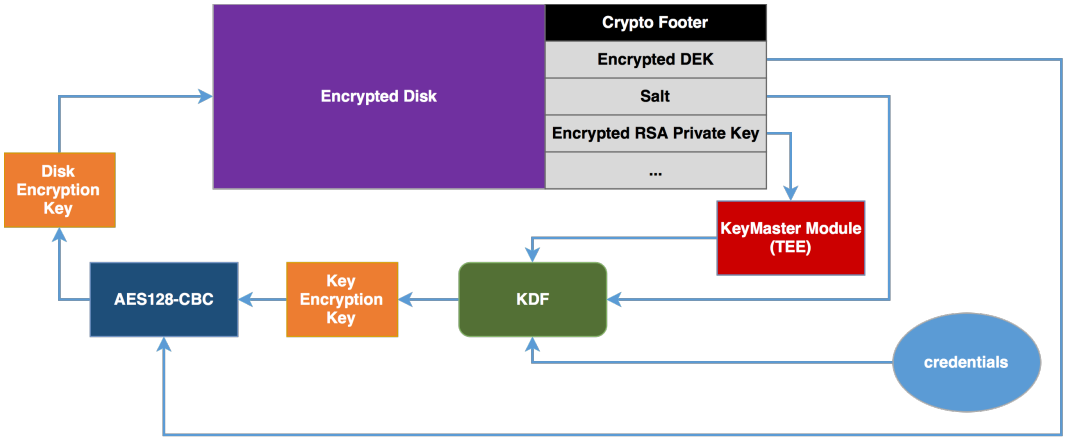

Figure 1: FDE Decryption Process[6]

Trusted Execution Environment

Since Android 5.0 there has been support for a hardware-backed key storage facility called KeyMaster. This is implemented using a TEE. The TEE is a secure area of the main processor, and is completely separate from the Android Operating System (OS). The KeyMaster module is intended to assure the protection of cryptographic keys generated by applications. The KeyMaster module can generate keys, and perform cryptographic operations on them. Once the keys are generated in the KeyMaster module, they are encrypted and returned to Android. If Android wants to perform an operation (e.g. sign data) using the keys, it supplies

the encrypted key to KeyMaster. KeyMaster decrypts the key, signs the data, and returns the result. In order to make the KEK resilient against offline-attacks, the KDF is extended by signing the key with a key stored in KeyMaster. This key is the RSA-2048 private key, which is stored encrypted in the crypto footer.

GateKeeper

Since Android 6 [15], the module responsible for device pattern/password/PIN authentication is called "Gatekeeper" [16]. Gatekeeper also handles rate limiting for incorrect authentication attempts. Gatekeeper consists of 3 main components: "gatekeeperd", "gatekeeper (Hardware Abstraction Layer (HAL))" and, "gatekeeper (TEE)". Gatekeeperd is the component that performs the platform independent logic. Gatekeeper (HAL) interfaces gatekeeperd with the device hardware. Gatekeeper (TEE) performs the authentication of the device PIN, pattern and, password inside the TEE.

4.2 File Based

As with FDE, devices employing FBE only have their /data partition encrypted. The encryption is now done at a filesystem level rather than at a block level. This is achieved using native ext4 filesystem encryption [1]. This encryption method is implemented by Google. The code to do this was pushed to the mainline linux kernel in version 4.4 [2]. This functionality was also backported to version 3.10. There is also some question in the community as to whether this code is stable enough for inclusion in the kernel. During our review of the source tree we have noticed 14 "TODOs" and 4 "FIXMEs" [14] [17]. This further indicates that there may be some potential implementation issues.

Multiple Encryption Areas

FBE allows more fine-grained control of what is encrypted when compared to FDE. This enables the device to have two areas of storage that are encrypted with separate encryption policies. There is the Device Encrypted (DE) area which is accessible immediately after the device has powered on. The DE storage area is available before user authentication. There is also a Credential Encrypted (CE) area that is only accessible after the user has input their authentication credentials. The CE area is the default storage location. Having these two areas decrypted according to these different policies solves the problem with FDE that was mentioned in section 1.

When the device only has access to the DE storage area it is in what's called "direct boot" mode. Direct boot mode enables the device to receive phone calls and honour user set alarms even after unexpected reboots. This separation also allows multiple profiles to be encrypted with different keys on the same filesystem. This gives more security for work profiles as the data of the personal and work profiles can be better partitioned.

[1] `https://www.phoronix.com/scan.php?page=news_item&px=EXT4-Changes-Linux-4.1`
[2] `https://www.phoronix.com/scan.php?page=news_item&px=EXT4-Encryption-Support`

Keys

When using FBE there is a DEK that is read from */dev/urandom* [18, line 354]. This is a 64 byte (512 bit) key. The whole DEK is used to encrypt the file contents. 256 bits of the DEK are used for file name encryption. As is the same with FDE, a second key is employed called the KEK. This key is derived from user input and encrypts the DEK.

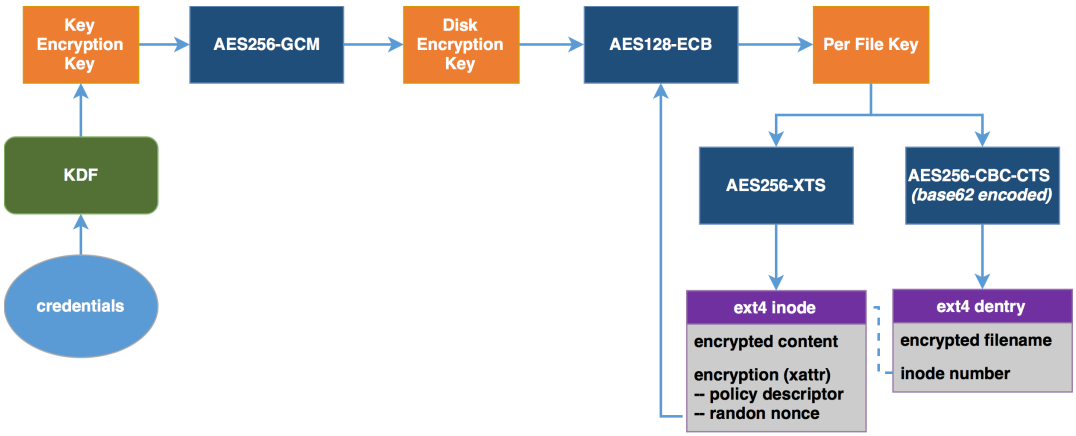

Figure 2: File Based Encryption Decryption Process

Modes of Operation

As shown in Figure 2 there are 3 main modes of operation applied when using FBE. The DEK is encrypted with the KEK using AES-256 in Galois Counter Mode (GCM). This KEK is held securely in the TEE. Three things are required to access the KEK held in the TEE [2].

- Stretched Credential: The users' authentication credentials are salted and hashed, to be used as input for the scrypt algorithm. The output is the 'stretched credential'.
- Auth Token: A cryptographically authenticated token that is generated by gatekeeper when the user logs in.
- "secdiscardable hash": A 512 bit hash of a random 16KB file that is stored for each user. This file is stored alongside some other information that is used to derive the user keys.

Using ext4 filesystem level encryption each file is encrypted using a separate key. The keys used are all derived from the DEK. Each file has a nonce associated with it. The nonce for each file is a 16 byte sequence [19]. This nonce is stored in the "dentry" associated with each file. A dentry is a data structure that links a file to its inode entry in the directory structure

of an ext4 filesystem [20]. The Per File Keys are derived from the DEK and nonce combined using AES in Electronic Code Book (ECB) mode [19].

The names of files are encrypted using AES-256 in Cipher Block Chaining with Ciphertext Stealing (CBC with CTS) mode. An IV of 0 is used for this operation [21, line 116]. The resulting ciphertext is then encoded using a base62 encoding. The ciphertext bits are encoded to the set [a-zA-Z0-9_+] [21, line 200]. This is to ensure that the encrypted filenames are still legal for an ext4 filesystem. The content of the files are encrypted using AES-256 in XTS mode. As XTS is being used 512 bits of key material are needed. For an overview of the modes of operation employed see appendix B.

Key Storage

The closest thing to a direct analogy of the crypto footer on an Android 7 device is the directory */data/misc/vold/user_keys* [17, line 67]. This directory contains two sub-directories: 'de' and 'ce'. These directories both contain one sub-directory for each user on the device. The user directories are called '0' for the primary user, '10' for the secondary user and, increasing sequentially for further users. User keys for both encryption areas are stored within this directory tree This directory structure can be seen in Listing 1.

Listing 1: FBE Key Storage Directory Structure

```
user_keys
|-- ce
|   |-- 0
|       |-- current
|           |-- encrypted_key
|           |-- keymaster_key_blob
|           |-- salt
|           |-- secdiscardable
|           |-- stretching
|           --- version
--- de
    |-- 0
        |-- encrypted_key
        |-- keymaster_key_blob
        |-- secdiscardable
        |-- stretching
        --- version
```

The directory 'ce' for each user contains the required data for deriving the Per User Key for unlocking the credential encrypted area. This includes the salt that will be combined with the users credentials when unlocking the DEK. The file 'stretching' contains the tweak parameters that will be used in scrypt when stretching the users credentials [22]. The contents of this file have the form "scrypt X:Y:Z" e.g. "scrypt 15:3:1". This file is parsed by "system/vold/ScryptParameters.cpp"[22]. Each of these values equates to one of the tweak parameters to be used in scrypt. In the given example the values $N = 2^{15}$, $r = 2^3$ and, $p = 2^1$ [23, line 225]. The stretched value of the user credentials is one of the tokens needed for accessing the DEK. The file 'secdiscardable' is hashed to create a 512 bit digest. This is

the second token needed to unlock the DEK. The final authentication token is generated by gatekeeper on successful login. This final token is not stored on disk.

In [23, line 204] we see the function "stretchSecret". It seems to apply the scrypt function only once to the user credentials. The file 'keymaster_key_blob' contains, we assume, an encrypted RSA key that is passed to the Keymaster for use as part of the key derivation process. During the FDE key derivation process a RSA key is needed to sign intermediate values. FBE follows a pattern that is similar to FDE. Therefore, we assume the file 'encrypted_key' contains the encrypted DEK.

The de folder contains similar files to those in the ce folder. We believe they are used in a manner that is almost identical to the way the files in the ce folder are used. There is obviously no salt value needed for the ce area as there is no user input required.

5 Existing Attack Scenarios Against Android Full Disk Encryption

In this section we will introduce some of the attacks that have been shown to be effective against FDE schemes. More specifically, we focus on these attacks as applied to Android FDE. We give an explanation of the workings of the aforementioned attacks and how they have been applied to Android.

5.1 Brute Force Attacks

A brute force attack (exhaustive search) consists of systematically trying all possible credentials until the correct one is found. With the online variant the password is entered directly on to the device. With the offline variant the brute forcing is carried out on a different host than the device.

5.1.1 Online

In general, with an online search the attacker would have to manually enter one authentication attempt after another. The countermeasure against an online brute force attack for Android 4.4 and 5 is a 30 second timeout after every 5th incorrect authentication attempt [4]. In Android 6 the rate limiting in place is a 30 second timeout after the 5th incorrect attempt. Furthermore, there is a 30 second timeout after every incorrect attempt from the 10th on [24, line 245]. The rate limiting scheme in Android 7 is described in section 6.1.1.

Oliver Kunz [4] describes an attack where the passwords are entered using the ADB. The Android command line program "input" can be used to simulate user interaction with the device. Using the option "text" for filling in the pincode, he managed to have an average PIN input duration of 1.8 seconds per attempt. Searching linearly through the keyspace of a 4 digit PIN (10000 keys) took 22 hours and 40 minutes to complete. The proposed attack requires ADB to be active on the device. The host computer must also be authenticated by the device in order to succeed. To overcome these issues two solutions exists.

The first solution uses the USB On-The-Go (OTG) specification. This allows USB devices to be attached to a smartphone. OTG is active even on a locked screen. Therefore, a self-programmed device that acts as a keyboard or mouse can enter the password automatically using this interface. A computer vision system can be used to automatically recognise an unlocked screen. A second approach was proposed by Engler and Vines [25]. They demonstrated a method which does not need any device connected externally. They developed a robot that presses buttons and automatically recognises a successful attempt. With this method 10,000 attempts could be made in 19 hours and 40 minutes. These attacks may not be possible depending on whether custom rate limits are enforced by the OEM. Furthermore, on initial boot the device may automatically wipe itself after a certain number of incorrect attempts.

5.1.2 Offline

As part of an offline search the attacker will perform password guessing on a different host without any input limitations. This is possible if the encrypted key can be extracted from the device storage along with an encrypted piece of known plaintext. On the external device, many trial keys can be used to decrypt the ciphertext. Once the decrypted text matches the known plaintext, the correct key has been found. In practice, specific bytes of an encrypted filesystem are used e.g. a magic number at a fixed offset. These bytes often have a known constant value so can be used as known plaintext.

Imaging a partition is possible via ADB, a custom recovery (if the bootloader is unlocked), or via the JTAG interface to the flash-memory. The advantage of imaging via JTAG is that it bypasses the OS. However, this method requires electronics and soldering skill. After extraction the attack is carried out on a host with more computational power and/or storage.

Tom Cannon [26] showed a proof-of-concept for Android 4.0 devices, by extracting the /metadata partition and the /data partition of the device. Oliver Kunz [4] confirmed that this attack is still feasible for Android 4.4 devices. A Python script was developed by Cannon to automate this attack[27]. The script parses the crypto footer from the /metadata partition, and uses "scrypt" with the parameters found in this footer. With this attack 10000 attempts could be made in under 1 hour.

In Android 5.0, Google improved the KEK generation with an optional hardware-backing to prevent offline attacks. In order to bind the KDF to the hardware, an intermediate step in the KDF is signed with a 'KeyMaster-encrypted RSA-2048 private key'. If the hardware backing is correctly implemented the offline attack is not possible anymore. This is because the hardware-backed KeyMaster module is required to produce this intermediate result.

Gal Beniamini [6] showed that the Android keystore (which is part of the TEE) is not actually bound to the underlying hardware for devices using Qualcomm chips. Due to this vulnerability, offline attacks as mentioned earlier are feasible for devices with a Qualcomm chip. Qualcomm makes chips for the majority of the world's smartphones. They have a market share of 65 percent[28]. Vulnerability CVE-2015-6639 and CVE-2016-2431 are used in order to extract the Qualcomm KeyMaster keys.

Qualcomm's implementation of a TEE is called Qualcomm Secure Execution Environment (QSEE). The QSEE environment allows "trustlets" (small applications) to execute on a secured area of the main processor. Trustlets provide a secure service to the insecure ("Normal World") Android OS. One of these trustlets is the KeyMaster application, which implements the key management API provided by the Android "keystore" daemon. As stated earlier, all generated keypairs are encrypted using a hardware-backed encryption key and returned to Android. The structure of the encrypted keypair is defined in the header files supplied by Qualcomm [29]. The header shows us that the encrypted keypair consists inter alia, of the modulus, public exponent and encrypted private exponent of the generated RSA key, and a HMAC key to verify the authenticity of the key [29, line 54].

The "sign_data" command was reverse engineered by Gal Beniamini. He showed that the HMAC key and the encryption key are both generated using a KDF which is not directly bound to the hardware. The KDF uses a pair of hard-coded strings and a hardware key (SHK) as input. The resulting key is stored in the KeyMaster's global buffer, and the pointer to the key is returned to the caller. The SHK cannot be extracted by software, because it is fused into the hardware. But the KeyMaster uses a key derived from the SHK and is therefore directly available to TrustZone. Besides that, the keys are constant because they are directly derived from the SHK and the two hardcoded strings. Since the key is available to TrustZone it is possible to extract the keys directly from TrustZone. This is done using vulnerability CVE-2016-2431, which made it possible to execute code in the TrustZone kernel. A shellcode stub was written, to be executed in the TrustZone kernel. This code reads the keys from the KeyMaster Application. Because the key can be extracted, it is possible to perform an offline brute force attack again on the users authentication method.

Vulnerability CVE-2015-6639 is fixed in the Security Bulletin of January 2016. Builds LMY49F or later and Android 6.0 with Security Patch Level of January 1 2016 do not have this issue. Vulnerability is CVE-2016-2431 fixed in may 2016. But even on patched devices, an attacker can downgrade the device to a vulnerable version and extract the key. A countermeasure to this attack is choosing a longer and more complex password. Using a 4 digit PIN only allows a maximum of 10,000 possible combinations. If we use a password of 4 characters, from upper and lower case characters along with numbers, we already have 36^4 possible combinations. This is already an increase of ≈ 168 times.

5.1.3 Semi-Offline

To overcome the hardware-backed keystore, Oliver Kunz [4] developed the semi-offline password search approach. As shown in appendix A it is only during the signing operation the Keymaster is needed. Therefore, it is possible to run the two scrypt computations on an external more powerful host. To execute this attack a proof of concept client-server application was developed for Android 5 and 6 [30]. The server software is based on cryptfs.c, the client software is based on the offline attack script mentioned in section 5.1.2.

First the /metadata and /data partitions are extracted from the device, using one of the methods stated in section 5.1.2. After extraction the bootloader is unlocked and the device automatically wiped. This is necessary in order to load the server software on the device. If the device already has an unlocked bootloader, this step can be skipped. The client software is run on the external host. Next, the KeyMaster-generated key blob is extracted from the crypto footer and send to the device. This is necessary because if a phone is wiped, this results in a new master key. Now the software on the device is initialised. Finally the following iterative process is run, until the external host finds the correct password:

1. External host runs the first scrypt function using a password and the salt from the crypto footer
2. This intermediate value is returned to the device
3. Device signs this value with the Keymaster

4. The signed value is returned to the external host

5. External host decrypts the master key and tests it for correctness

After the correct key is found, the /data partition can be decrypted on the external host. With this method 10,000 attempts could be made in 2 hours and 8 minutes by Oliver Kunz.

5.2 Cold Boot

With a cold boot attack an attacker with physical access to the device is able to retrieve data stored in RAM from a running system after restart from a cold boot. The attack carried out by powering the device on and off without letting the OS shut down in an orderly manner. The attack relies on the "remanence effect" of RAM [31]. Without power RAM contents fade away gradually over time, not instantly as is commonly thought. The colder the RAM is the slower content fades away.

Due to this effect, keys can be restored from the RAM through rebooting a PC with malicious USB driver, or replugging the RAM into another PC. The encryption key used by FDE, needs to be present in the systems main memory (RAM) in order to provide encryption. Cold boot attacks are known since at least 2008, when Halderman et. al [32] showed a proof of concept. This made it possible to extract sensitive information, such as cryptographic keys, from memory used in laptops and desktop computers.

Müller et. al [5] showed that the same attack was feasible against Android devices. They developed a proof of concept tool FROST (Forensic Recovery of Scrambled Telephones) for Galaxy Nexus devices with Android 4.0 installed. FROST is a recovery image installed after having physical access to the device. First the phone is cooled down to between 5°Cand 10°C, by putting it in a freezer. Next the phone is restarted (removing and putting back the battery) and the recovery image (FROST) is installed. On devices with an unlocked bootloader encryption keys can be recovered from RAM. The device must have an unlocked bootloader for FROST to be installed. Unlocking the bootloader triggers a wipe of the /data partition. After the wipe FROST can still be installed, but the contents on the user partition are gone. To overcome this issue the attacker could first image the partitions, before executing the attack.

The countermeasure for this attack is keeping the keys outside of RAM. Multiple solutions have been proposed to achieve this. These solutions focus on keeping the key material only in the CPU and not writing it to memory. Götzfried and Müller [1] proposed to hold the key in CPU caches. Müller and Dewald et. al proposed to hold the key in SSE registers [33]. Müller and Freiling et. al [34] proposed to keep the key in debug registers. This problem is also solved by using a TEE.

5.3 Evil Maid

Implementations of FDE are often vulnerable to a class of boot-time attacks generally referred to as an Evil Maid attack [8]. The typical scenario used for an evil maid attack involves a person travelling with an encrypted device (computer/smartphone/tablet). When the traveller leaves the room without the device, an Evil Maid comes in to clean. The data stored on the device is encrypted, so this cannot be read. To overcome this, the Evil Maid installs a keylogger on the device. The keylogger can be hardware or software, but is often a small piece of software installed to the unencrypted boot partition of a hard disk. When the traveller returns and uses the device, the keylogger reads the FDE password and stores it on the disk or sends it out over the network. Evil Maid attacks are possible because there is always a part of the disk left unencrypted. This is the same with Android FDE, which only encrypts the /data partition. Encrypting everything and even sign or encrypt the boot-loader does not prevent this attack, because the device could be physically modified [35].

Götzfried and Müller [1] were the first to provide a proof of concept (EvilDroid) for a Galaxy Nexus device running Android 4.0. They showed that with physical access to an encrypted smartphone, the Android system partition can be subverted with keylogging. The attack requires a phone with an unlocked bootloader. EvilDroid is installed on the system partition. EvilDroid patches the file */system/vold/cryptfs.c*. At the time they applied the attack this file included code for displaying PIN prompts. The PIN's are stored in the unencrypted cache partition of the device. A second approach is mentioned for devices with a locked bootloader. The target phone is exchanged for an identical model. Then if the PIN is entered into the replaced phone, the PIN is send to the attacker via SMS or internet. The victim will notice that something is wrong after entering the PIN, but by then the Evil Maid already has the phone and the PIN.

Another approach to execute an Evil Maid attack on Android is proposed by Artenstein et. al [9]. They showed it is possible to intercept "Binder" communications, and use Binder as a Keylogger. Their approach is called "Man in the Binder attack". Binder is the method by which IPC occurs on an Android device. It is even used for communication between activities in a single application. To receive keyboard data, an application has to register with an Input Method Editor (IME) server. The IME is the keyboard implementation used in Android. In most Android images the default IME is *com.android.inputmethod.latin*. By intercepting the Binder communication sent by the IME, passwords can be recovered. The attack proposed requires a device running with root permissions, or with an unlocked bootloader.

5.4 Fingerprint Authentication

The total number of devices incorporating a fingerprint scanner is expected to reach 990 million in 2017 [36]. Since version 6.0, Android has standardised support for fingerprint authentication [37]. Fingerprint authentication attacks can be hardware based (e.g. faking fingerprints) or software-based. Thom Does et. al [10] were the first to subvert the fingerprint authentication mechanism for Android 6.0.

When a fingerprint is enrolled, the minutiae template of this fingerprint is stored in the TEE. The TEE assigns a random ID to to the minutiae template which can be any number except ‘0’. The ID ‘0’ is reserved for indicating a non recognised fingerprint.

When an authentication attempt takes place using a fingerprint, the raw fingerprint data is send to the TEE. The TEE compares the raw fingerprint data with the enrolled minutiae templates. The TEE sends back a ‘0’ to Android if the fingerprint is not enrolled, and the corresponding fingerprint ID if the fingerprint is enrolled. To verify that the fingerprint is enrolled into the TEE, Android checks if the returned fingerprint ID from the TEE is not ‘0’. Android has no knowledge of which fingerprints are enrolled into the TEE. Therefore it is possible to let Android think the fingerprint is valid by changing the ID in the response to anything non-zero. In order to execute this attack, root access to the phone is required.

6 Results

In this section we address how the previously mentioned attacks apply to the current Android version. We explain what we have done to apply these attacks to our testing device. Furthermore, we demonstrate the results we have obtained and what has been learnt during this process.

6.1 Brute Force Attacks

6.1.1 Online

As stated in section 4.1 the module responsible for handling rate limiting for incorrect authentication attempts is called "Gatekeeper". The rate limits in place for Android 7 have been changed since previous versions [38, line 259]. Listing 2 shows the current rate limiting behaviour. Listing 2 is a modified comment that was taken from the file "gatekeeper.cpp" [38, line 247]. It was supposed to document the behaviour of the code. It is worth noting that this comment was however inaccurate [3].

Listing 2: Android 7 Rate Limits

```
1  /*
2   * Calculates the timeout in milliseconds as a function of the failure
3   * counter 'x' as follows:
4   *
5   * [0. 5) -> 0
6   * 5 -> 30
7   * [6, 10) -> 0
8   * [10, 30) -> 30
9   * [30, 140) -> 30 * (2^((x - 30)/10))
10  * [140, inf) -> 1 day
11  *
12  */
```

After the 5th and 10th incorrect authentication attempt there is a timeout of 30 seconds. Every successive attempt up to the 30th gets the same timeout. Between 30 and 140 attempts the timeout grows in an exponential manner from 32 seconds to 61440 seconds (17 hours 4 minutes). After 140 attempts the timeout for each incorrect attempt is 1 day. Due to the thresholds used, an online brute force attack against a 4 digit PIN would take around 27 years to complete.

If the device has an unlocked bootloader, it is possible to re-flash the system partition of the device after a given number of attempts. This will reset the rate limiting counter. After reflashing, the device starts up in Direct Boot mode. ADB is not available in this mode, therefore only manual input is possible. But this method will still bring down the total time to brute force the device.

[3] Line 8 incorrectly read "[11,30) ->30"

As previously mentioned, the file "gatekeeper.cpp"[38] contains the platform independent gatekeeper code. There are two methods in this file that seem to be of most interest in terms of stopping rate limiting. These are "ComputeRetryTimeout"[38, line 259] and "IncrementFailureRecord"[38, line 306]. The ComputeRetryTimeout method returns the delay between authentication attempts depending on how many incorrect attempts have already been made. IncrementFailureRecord keeps track of the number of incorrect authentication attempts that have been made. After making changes to the necessary file the gatekeeper module can be rebuilt using *mmma system/gatekeeper* from the root of the source tree. This generates, among other files, two libraries in the output folder: *system/lib{,64}/libgatekeeper.so*. These are then be pushed to the same location on the device.

Two of the things we tried were making ComputeRetryTimeout always return 0 and removing the increment of the failure counter in IncrementFailureRecord. The freshly built libraries were pushed to the device. They had no effect on the authentication behaviour of the device. We noticed that there were also two similar libraries located on the device: */vendor/hw/lib{,64}/libgatekeeper.msm*8992.*so* We don't know the exact function of these libraries. The "msm8992" in their names refers to the model of Qualcomm Snapdragon 808 SoC in the device. Therefore, we assume they implement functions that are used by gatekeeper (HAL).

The modifications mentioned previously did not work by themselves. When we removed the similar libraries from /vendor/hw/libs{,64} we had different results. We got the device into a state where unlimited authentication attempts were allowed. However, we could not authenticate even with correct credentials. We assume that the vendor libraries contain code implementing the previously mentioned methods that takes precedence. If this assumption is correct it would be possible to modify the binary version of the vendor supplied libraries and implement the same changes there. We theorise that this would then have the desired effect of removing all authentication rate limits. We were unable to implement this due to time constraints.

6.1.2 Offline

Since Android 5.0, the KDF to generate the KEK is bound to hardware. This is still the case for Android 7.0, rendering the offline attack unfeasible. But, the offline attack was made feasible again due to a vulnerability in Qualcomm chips. This is patched in software, but not in hardware making a lot of devices vulnerable again. These vulnerabilities are fixed since the first release of Android 7.0. A downgrade attack is not possible with FBE enabled, because the attacker would need to downgrade to Android 6 or lower which does not support FBE.

6.1.3 Semi-Offline

The Semi-offline attack on FDE was possible because the internal Keymaster key used to encrypt the private RSA exponent is static [4]. If the internal Keymaster key is still static in Android 7, then this attack could be applied to Android 7 as well. A proof-of-concept of this attack could work as follows:

First the /data partition is extracted from the device, using one of the methods stated in section 5.1.2. The server software implements the scrypt function, used by the KDF to derive the KEK. The client software should be altered to incorporate the Per File Keys. Next, the "keymaster_key_blob" is extracted from the */data/misc/vold/user_keys* and send to the device. Finally the following iterative process is run, until the external host finds the correct password:

1. External host runs the first scrypt function using a password and the salt from */data/misc/vold/user_keys*
2. This intermediate value is returned to the device
3. Device signs this value with the Keymaster
4. The signed value is returned to the external host
5. External host decrypts the master key and tests it for correctness

After the correct key is found, the /data partition can be decrypted on the external host.

6.2 Cold Boot

Since the release of Android 7.0, Google has mandated the following for all new devices shipping with this version. "The keys protecting CE and DE storage areas MUST be cryptographically bound to a hardware-backed Keystore"[39]. This effectively means that all devices which support FBE are required to implement some form of TEE.

The cryptographic keys protecting the device encryption are now stored only in the TEE. Because the keys are held only in the TEE, they are never written to RAM. Data remanace attacks applied to cryptographic keys are rendered obsolete because of this fact.

6.3 Evil Maid

As previously mentioned, a Man in the Binder attack is possible against Android. To implement this we modify the file "IPCThreadState.cpp" [40]. In this file there is an ioctl system call. This is the mechanism by which all Binder parcels are sent. Therefore, all data must flow through this single point. This is the point where we intercept the passing parcels. This ioctl call is situated in the "talkWithDriver" method [40, line 877].

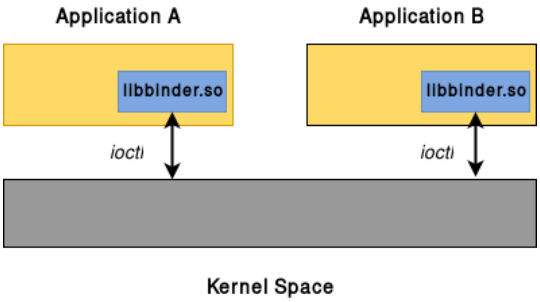

Figure 3: Ioctl Bottleneck

We wrote a hooked version of this function that will allow the ioctl function to be called normally. When the response has been made however, some of the data is intercepted. This hooked function can be seen in Listing 3.

Listing 3: IPCThreadState.cpp

```
int evil_ioctl(int fd, int op_type, binder_write_read* bwr)
{
    int res = ioctl(fd, op_type, bwr);
    evil_reply_manipulation(bwr);
    return res;
}
```

Binder calls usually follow a request response pattern. As shown Listing 3 the "bwr" data structure contains two buffers. One which points to the request ("command" in Binder terminology). Another which points to the response ("reply" in Binder terminology). These buffers point to two things sequentially: a transaction tag, and a "binder_transaction_data" structure. The transaction tag is used to indicate the type of communication that is being done. It makes it possible to limit the number of communications we need to take action on. The binder_transaction_data contains a number of fields. For our purposes the most important fields are the "code" and "data.ptr.buffer" fields. Code denotes the function that will be called by the receiver of the Binder transaction. data.ptr.buffer is in fact a Binder parcel object. It contains the data that will be needed for the transaction. The definitions of the mentioned structures can be found in [41] [4].

We are applying this attack to implement a keylogger. Therefore, we are only interested in transaction tags that have the value "BR_TRANSACTION". This is the value that is used when an application is receiving data from the IME application [9]. We then know if there's a binder_transaction_data structure attached that is of interest to us. Now we have a relevant binder_transaction_data we can further narrow our search by looking at its code member. Whenever a key is pressed on the keyboard application a callback to the interface "IInputContext" is triggered.[9]. This interface sends data up to the "InputContext" class,

[4]The structure labelled "interface descriptor" in the figure is only one portion of the parcel.

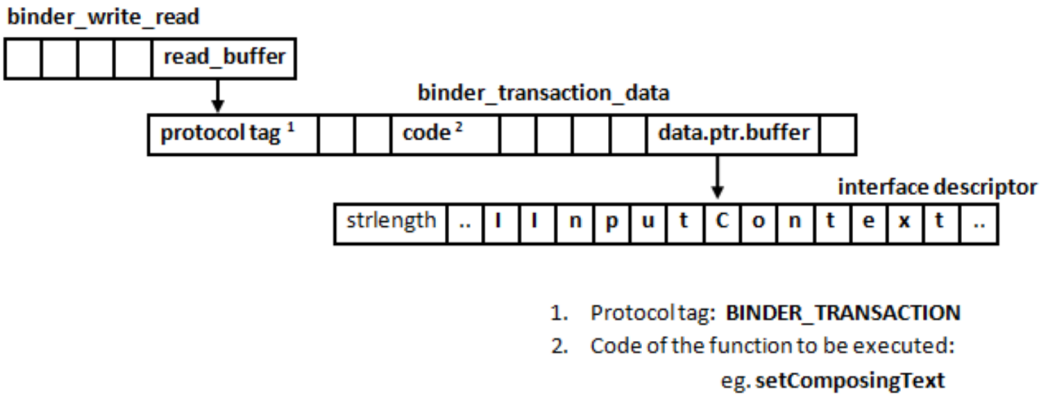

Figure 4: Overview of Binder Data Structures [42]

which is responsible for handling received keyboard input. Through searching the source tree we find that the only file containing a definition of IInputContext is an Android Interface Definition Language (AIDL) file [43]. AIDL is an Android specific language that defines a standard interface that processes use for IPC. If we look in the necessary AIDL file [44] we can see that the 7th function defined is "setComposingText". When an AIDL file is used for compilation, each function is generated in the order that it appears in this file. The function numbering begins at one. The code field in a binder_transaction_data structure refers to this.

Now we know the transaction tag we are interested in, we can only take action on these transactions. Within a binder_transaction_data structure we know the function code that is associated with a function that is used to send text via Binder. The data.ptr.buffer member within the same transaction data is actually a pointer to a Binder parcel. This parcel contains the name of the interface that it is being sent to along with the actual text data being sent. It is clear how Binder can be used as a keylogger from this description. After the text data has been intercepted it can be written to an unencrypted area of the device. This process can also begin the creation of a new background thread. This background thread could send the textual data to a remote network location.

Due to time constraints we were unable to implement the last part portion of the attack where the internals of the parcel are parsed. We were able to get to the point where we were reliably intercepting the transaction data. We created logging statements that were printing information about the transaction to ADB logcat. This has been shown to be a successful attack before. Furthermore we could not find any evidence that Binder has had any major overhauls in recent Android releases. So, we are reasonably confident that it still works.

6.4 Fingerprint bypass

As shown in Figure 5, the fingerprint authentication mechanism consists of multiple components. The TEE is used for the key management and stores fingerprint data. On top of the TEE is the Fingerprint HAL. This is a vendor specific interface to the hardware. HAL is communicating with the "fingerprintd" component in the Android OS. Fingerprintd makes the calls for fingerprint enrolment, removal, and authentication. Fingerprintd communicates via Binder with "FingerprintService". This component ensures that third-party applications are not able to distinguish individual fingerprints. FingerprintService communicates via Binder with "FingerprintManager API". This is the component used by application developers to use fingerprint authentication.

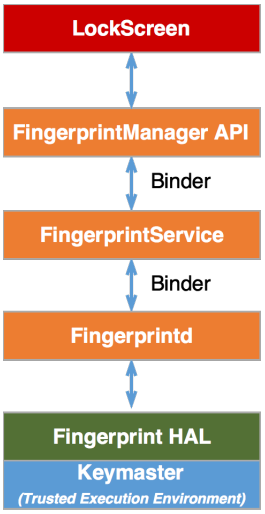

Figure 5: Fingerprint Authentication Method [45]

FingerprintService performs a check on the fingerprint ID received from fingerprintd. If the received ID is not '0', the FingerprintService sends a message to the FingerprintManager instance indicating that the authentication succeeded. The application implementing FingerprintManager performs an action based on this message. E.g. the Android LockScreen will unlock.

We modified the "hal_notify_callback" method in *FingerprintDaemonProxy.cpp* [46, line 70]. As shown in Listing 4, the fingerprint ID returned is modified to return an arbitrary non-zero value. We build the modified version and replace the legitimate binary with the modified one. The fingerprintd binary on the phone is located at *system/bin/fingerprintd*.

Listing 4: FingerprintDaemonProxy.cpp

```
1    callback->onAuthenticated(device,
2 //            msg->data.authenticated.finger.fid,
3            0x1a4, // non-zero id
4            msg->data.authenticated.finger.gid);
```

Due to this change, the device will accept any fingerprint. In order to carry out this attack, root access is required. Without root access it is not possible to change files at the system partition.

7 Discussion

Three out of six attacks appear feasible for Android 7; Semi-Offline Brute Force Attack, Evil Maid Attack, and Fingerprint Authentication Attack. The Online Brute Force Attack may be possible if gatekeeper can be subverted. Important to state is that these attacks will not work for all devices, pre-conditions need to be met in order for the attack to work. The pre-conditions needed for the Semi-Offline attack is physical access to the device for as long as the attack takes, and a means to image a partition. For the Evil Maid Attack and the Fingerprint Authentication Attack the pre-conditions are a rooted device and physical access for as long as it takes to replace the binary.

The Evil Maid Attack and Fingerprint Authentication Attack require the subversion of the libbinder.so library and the fingerprintd binary respectively. Replacing a system binary on an Android device has a few consequences. Android 4.4 and later supports verified boot through the device-mapper-verity (dm-verity) kernel feature [47]. dm-verity provides transparent integrity checking of block devices. For every block, Android stores a SHA256 hash. dm-verity helps Android users be sure when booting a device it is in the same state as when it was last used. When a binary is changed or replaced by another in the /system partition, dm-verity will display a warning to the user during booting. This warning message will disappear after five seconds and after this message the device boots like normal. If the user has a unlocked bootloader, dm-verity displays a different warning. Instead a warning will display during booting that the bootloader is unlocked. This makes a changed binary harder to notice for the user.

In this paper we focused exclusively on stock Android. This means that any of the previous results will hold for devices running stock Android. OEMs often apply customisations to the Android software that is shipped with their devices. These customisations are usually cosmetic and do not effect the core Android components. If OEMs do make modifications to the underlying Android components then these attacks may be rendered obsolete.

Finally, we only described a subset of the attacks that have been shown to work against Android FDE. The list of attacks we chose is by no means exhaustive. There are also other attacks that have been shown to be effective against Android authentication. Included in these attacks are the "smudge attack" from 2010 [48]. Video-based side-channel attacks. The more recent video-based side-channel attack demonstrated that it is possible to reliably infer the pattern used on a lockscreen [49]. This attack only requires a mobile phone camera, filming the target at 2 meter distance. The video did not need to capture any content displayed on the screen. The pattern is deduced only from the movement of the fingertip.

7.1 AOSP recommendations

In the Evil Maid Attack we replaced the fingerprintd binary which is on the system partition. On a device with a locked bootloader, this triggers a warning message from dm-verity. The warning is that the device integrity cannot be guaranteed. This occurs because the system partition has been modified. The warning message disappears after five seconds without requiring user interaction. We recommend to extend dm-verity with the requirement of user input before proceeding. This removes the possibility that a user can accidentally miss the warning.

Binder parcels integrity is not guaranteed. The data transferred between an application and the Binder driver is in plain text This enables us to manipulate the Binder communication, and subvert the fingerprint authentication. We recommend that binder parcels are protected. Yadu Kaladharan et. al propose a method for doing this using encryption [42]. This method introduces some overhead as the sending and receiving processes must encrypt/decrypt parcels on the fly. Due to the power of modern smartphone hardware this overhead is not prohibitive. We believe that this performance impact is worth it for the security gain.

This Evil Maid attack is possible because only the /data partition of the device is encrypted. We recommend to encrypt more of the device, For example the /system partition. This will reduce the unencrypted attack surface. Encrypting a much larger quantity of the device is standard practice for PC FDE. If the system partition is encrypted in a manner similar to the DE area of the data partition the critical system components of the phone will bee protected at rest. Although encrypting more of the device, will not entirely prevent this attack it will be harder to execute [35].

7.2 End-user recommendations

By default, the system partition of an Android device is mounted as read-only. Root privileges are required to remount it as read-write. This is one way of exposing the device to the possibility of having malicious binaries pushed to it. To prevent modification of the system partition while the device is powered on the user can make sure that root access is disabled. This also prevents the user from removing any of the preloaded system applications which they may consider junk. There is a trade-off to be considered here. From a purely security perspective root access should be disabled.

Users of Android devices are able to flash custom recovery menus. This enables them to update their device software manually instead of waiting for the manufacturer. They are also able to replace the shipped Android version with a different one of their preference. Custom recovery menus allow updates to be flashed that are signed by the publicly known Android debug key. Android devices generally come with a quite restrictive recovery menu. The default recovery menus will usually only update the phone using a package that has been signed by the manufacturer. These recovery menus also do not usually allow ADB access. From a security standpoint having a restrictive recovery menu is a good thing. This means we cannot have shell access to a phone in recovery mode, making it much harder to read or write files on the device. It is also not possible to flash malicious updates.

Having an unlocked bootloader is another mechanism by which a user can modify the software on their phone. Users often do this to install a custom recovery menu. The initial unlocking of the bootloader will trigger the whole device to be wiped for security reasons. Usually the bootloader can be relocked without any loss of data. It is our recommendation that if a user has a phone configuration they are happy with, the device bootloader should be relocked.

8 Conclusion

In this paper we described the current attack scenario's for Android FDE. Brute Force attacks, Cold Boot attacks, Evil Maid attacks, and Fingerprint Authentication attacks were researched. We have shown that FBE is susceptible to many of the same attacks that FDE was. Three out of six attacks appear feasible for Android 7; Semi-Offline Brute Force Attack, Evil Maid Attack, and Fingerprint Authentication Attack. The Online Brute Force Attack is possible if the gatekeeper module can be subverted.

9 Future Work

In this paper we provided a first step into explaining the inner workings of Android FBE. However, some parts of FBE remain unclear. First, we provided an overview of the key storage in FBE. But is is still unclear if the "keymaster_key_blob" is used the same way as with FDE. We assume that this is the encrypted RSA key that is passed to the Keymaster, during the KDF. Further research should provide more clarity about the exact working of the key storage method.

Next, we provided an theoretical attack that can be used to implement the semi-offline attack against FBE. This attack relies on the private RSA exponent to be static. In Qualcomm chips this is the case, but it is unclear if other vendors have the same issue. We reason that if the private RSA exponent in the TEE is static then this attack is also feasible against FBE. A proof-of-concept should be made in order to test our theoretical attack.

Finally, we made a proof-of-concept for the Evil Maid attack. We were able to monitor and store Binder communication. But we were not able to extract the Parcels sent via Binder. Further research should provide a way to extract the Parcels in order to effectively use this attack in practice.

We have noted in section 4.2 that there is an encrypted version of the master key stored on the filesystem. It remains unclear to us how precisely this key is used. 512 bits of key material are needed for XEX-based tweaked-codebook mode with CTS (XTS) mode encryption, only 256 bits are needed for CBC with CTS. Is this file used for both keys or, in part, for both? With an AEAD encryption mode we would expect to see some inflation of ciphertext. This file is larger than the 512 bits needed for GCM. Further research should clarify this point.

References

[1] Johannes Götzfried and Tilo Müller. Analysing android's full disk encryption feature. *JoWUA*, 5(1):84–100, 2014. http://isyou.info/jowua/papers/jowua-v5n1-4.pdf.

[2] Google Inc. File Based Encryption — Android Open Source Project. https://source.android.com/security/encryption/file-based.html, November 2016. retrieved on: 09-01-2017.

[3] Google Inc. Android 7.0 Nougat. https://blog.google/products/android/android-70-nougat-more-powerful-os-made/, August 2016. retrieved on: 10-01-2017.

[4] Oliver Kunz. Android full-disk encryption: a security assessment. https://www.royalholloway.ac.uk/isg/documents/pdf/technicalreports/2016/rhul-isg-2016-8-oliver-kunz.pdf, April 2016. retrieved on: 10-01-2017.

[5] Tilo Müller and Michael Spreitzenbarth. Frost. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.466.5756&rep=rep1&type=pdf, 2013. retrieved on: 13-01-2017.

[6] Gal Beniamini. Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption. https://bits-please.blogspot.nl/2016/06/extracting-qualcomms-keymaster-keys.html, June 2016. retrieved on: 12-01-2017.

[7] Joanna Rutkowska. The Invisible Things Lab's blog: Why do I miss Microsoft BitLocker? https://theinvisiblethings.blogspot.nl/2009/01/why-do-i-miss-microsoft-biotlocker.html, January 2009. retrieved on: 13-01-2017.

[8] Daniel DeFreez. *Android privacy through encryption.* PhD thesis, Master's thesis, Southern Oregon University, 2012. http://defreez.com/articles/thesis.pdf.

[9] Nitay Artenstein and Idan Revivo. Man in the binder: He who controls ipc, controls the droid. http://docs.huihoo.com/blackhat/europe-2014/eu-14-Artenstein-Man-In-The-Binder-He-Who-Controls-IPC-Controls-The-Droid-wp.pdf, 2014. retrieved on: 13-01-2017.

[10] Thom Does and Mike Maarse. Subverting Android 6.0 fingerprint authentication. https://homepages.staff.os3.nl/~delaat/rp/2015-2016/p30/report.pdf, February 2016. retrieved on: 12-01-2017.

[11] Google Inc. Codenames, Tags, and Build Numbers AOSP. https://source.android.com/source/build-numbers.html, January 2017. retrieved on: 31-01-2017.

[12] The Android Open Source Project. android-7.1.1_r11 github repository. https://android.googlesource.com/platform/manifest/+/refs/heads/android-7.1.1_r11, January 2017. retrieved on: 31-01-2017.

[13] Google Inc. Building Android Open Source Project. https://source.android.com/source/building.html, August 2016. retrieved on: 31-01-2017.

[14] The Android Open Source Project. cryptfs.c. https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/cryptfs.c, December 2016. retrieved on: 13-01-2017.

[15] Nikolay Elenkov. Android explorations: Keystore redesign in android m. https://nelenkov.blogspot.nl/2015/06/keystore-redesign-in-android-m.html?m=1, June 2016. retrieved on: 10-02-2017.

[16] Google Inc. Gatekeeper android open source project. https://source.android.com/security/authentication/gatekeeper.html, August 2016. retrieved on: 10-02-2017.

[17] The Android Open Source Project. Ext4Crypt.cpp. https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/Ext4Crypt.cpp, December 2016. retrieved on: 01-02-2017.

[18] The Android Open Source Project. Utils.cpp. https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/Utils.cpp, December 2016. retrieved on: 01-02-2017.

[19] Guillaume Delugré. A glimpse of ext4 filesystem-level encryption. http://blog.quarkslab.com/a-glimpse-of-ext4-filesystem-level-encryption.html, August 2015. retrieved on: 05-02-2017.

[20] M. Tim Jones. Anatomy of the linux file system. https://web.archive.org/web/20150505112327/http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/, October 2007. retrieved on: 10-02-2017.

[21] The Android Open Source Project. crypto_fname.c. https://android.googlesource.com/kernel/msm/+/android-7.1.1_r0.20/fs/ext4/crypto_fname.c, November 2016. retrieved on: 01-02-2017.

[22] The Android Open Source Project. ScryptParameters.cpp. https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/ScryptParameters.cpp, December 2016. retrieved on: 01-02-2017.

[23] The Android Open Source Project. Utils.cpp. https://android.googlesource.com/platform/system/vold/+/android-7.1.1_r11/KeyStorage.cpp, December 2016. retrieved on: 01-02-2017.

[24] The Android Open Source Project. gatekeeper.cpp. https://android.googlesource.com/platform/system/gatekeeper/+/marshmallow-release/gatekeeper.cpp, June 2015. retrieved on: 08-02-2017.

[25] J. Engler and P. Vines. Electromechanical PIN Cracking Implementation and Practicality. https://www.defcon.org/images/defcon-21/dc-21-presentations/Engler-Vines/DEFCON-21-Engler-Vines-Electromechanical-PIN-Cracking-WP.pdf, July 2013. retrieved on: 27-01-2017.

[26] Thomas Cannon. Into the droid - gaining access to android user data. https://www.defcon.org/images/defcon-20/dc-20-presentations/Cannon/DEFCON-20-Cannon-Into-The-Droid.pdf, July 2012. retrieved on: 08-02-2017.

[27] N. Elenkov T. Cannon, S. Bradford and O. Kunz. bruteforce-stdcrypto.py. Python Script. https://github.com/mlet/Santoku-Linux, June 2015. retrieved on: 27-01-2017.

[28] Catalin Cimpanu. QuadRooter Android Security Bugs Affect over 900 Million Devices. http://news.softpedia.com/news/quadrooter-android-security-bugs-affect-over-900-million-devices-507052.shtml, August 2016. retrieved on: 31-01-2017.

[29] The Android Open Source Project. keymaster_qcom.h. https://android.googlesource.com/platform/hardware/qcom/keymaster/+/android-7.1.1_r11/keymaster_qcom.h, December 2016. retrieved on: 31-01-2017.

[30] O. Kunz. BSides Lisbon 2016 - Semi-Offline Attack on the Android Full-Disk Encryption. url:https://www.youtube.com/watch?v=QpCWS5dM7eY, November 2016. retrieved on: 27-01-2017.

[31] Peter Gutmann. Data remanence in semiconductor devices. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, page 4. USENIX Association, 2001.

[32] JA Haldermany, SD Schoenz, N Heningery, W Clarksony, W Paulx, JA Calandrinoy, AJ Feldmany, J Appelbaum, and EW Felten. Lest we remember: Cold boot attacks on encryption keys, 2008. http://citpsite.s3-website-us-east-1.amazonaws.com/oldsite-htdocs/pub/coldboot.pdf, February 2008. retrieved on: 27-01-2017.

[33] Tilo Müller, Andreas Dewald, and Felix C Freiling. Aesse: a cold-boot resistant implementation of aes. In *Proceedings of the Third European Workshop on System Security*, pages 42–47. ACM, 2010.

[34] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, volume 17, 2011.

[35] Sven Türpe, Andreas Poller, Jan Steffan, Jan-Peter Stotz, and Jan Trukenmüller. Attacking the bitlocker boot process. In *International Conference on Trusted Computing*, pages 183–196. Springer, 2009.

[36] Goode Intelligence. Fingerprint Biometric Market Intelligence. http://www.goodeintelligence.com/media/report_link_files/1391518443goode_intelligence_fingerprint_biometric_market_intelligence_edition_one_december_2013.pdf, December 2013. retrieved on: 31-01-2017.

[37] Google Inc. Android 6.0 Compatibility Definition. https://static.googleusercontent.com/media/source.android.com/en//compatibility/6.0/android-6.0-cdd.pdf, April 2016. retrieved on: 31-01-2017.

[38] The Android Open Source Project. gatekeeper.cpp. https://android.googlesource.com/platform/system/gatekeeper/+/android-7.1.1_r11/gatekeeper.cpp, December 2016. retrieved on: 31-01-2017.

[39] Google Inc. Android 7.0 (N) Compatibility Definition. https://static.googleusercontent.com/media/source.android.com/en//compatibility/7.0/android-7.0-cdd.pdf, October 2016. retrieved on: 31-01-2017.

[40] The Android Open Source Project. IPCThreadState.cpp. https://android.googlesource.com/platform/frameworks/native/+/android-7.1.1_r11/libs/binder/IPCThreadState.cpp, December 2016. retrieved on: 01-02-2017.

[41] The Android Open Source Project. binder.h. https://android.googlesource.com/platform/bionic/+/android-7.1.1_r11/libc/kernel/uapi/linux/android/binder.h, December 2016. retrieved on: 01-02-2017. See also: https://android.googlesource.com/platform/external/kernel-headers/+/android-7.1.1_r11/original/uapi/linux/android/binder.h.

[42] Yadu Kaladharan, Prabhaker Mateti, and KP Jevitha. An encryption technique to thwart android binder exploits. In *Intelligent Systems Technologies and Applications*, pages 13–21. Springer, 2016.

[43] The Android Open Source Project. Android Interface Definition Language (AIDL) — Android Developers. https://developer.android.com/guide/components/aidl.html, December 2016. retrieved on: 01-02-2017.

[44] The Android Open Source Project. IInputContext.aidl. https://android.googlesource.com/platform/frameworks/base/+/android-7.1.1_r11/core/java/com/android/internal/view/IInputContext.aidl, December 2016. retrieved on: 01-02-2017.

[45] Google Inc. Fingerprint hal android open source project. https://source.android.com/security/authentication/fingerprint-hal.html, August 2016. retrieved on: 07-02-2017.

[46] The Android Open Source Project. FingerprintDaemonProxy. https://android.googlesource.com/platform/system/core/+/android-7.1.1_r11/fingerprintd/FingerprintDaemonProxy.cpp, December 2016. retrieved on: 01-02-2017.

[47] Google Inc. Verified boot android open source project. https://source.android.com/security/verifiedboot/, August 2016. retrieved on: 07-02-2017.

[48] Adam J Aviv, Katherine L Gibson, Evan Mossop, Matt Blaze, and Jonathan M Smith. Smudge attacks on smartphone touch screens. https://www.usenix.org/legacy/event/woot10/tech/full_papers/Aviv.pdf, August 2010. retrieved on: 07-02-2017.

[49] Guixin Ye, Zhanyong Tang, Dingyi Fang, Xiaojiang Chen, Kwang In Kim, Ben Taylor, and Zheng Wang. Cracking android pattern lock in five attempts. http://www.lancaster.ac.uk/staff/wangz3/publications/ndss_17.pdf, January 2017. retrieved on: 07-02-2017.

[50] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, pages 1–16, 2009. retrieved on: 05-02-2017.

[51] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. `http://web.cs.ucdavis.edu/~rogaway/papers/offsets.pdf`, 2004. retrieved on: 05-02-2017.

[52] Wikipedia. Disk encryption theory. `https://en.wikipedia.org/wiki/Disk_encryption_theory`, December 2016. retrieved on: 10-02-2017.

Appendices

A Scrypt and Android's KDF

The scrypt function was created by Colin Percival in 2009 [50]. It was created to be a KDF that's both CPU and memory hard. The intention is to make it hard to do quick computation even in the face of specialised hardware.

The scrypt function has general form.

$$x = scrypt(secret, salt, N, p, lenDK) \tag{2}$$

Where x is the stretched key that has been derived, *secret* is the credential to be stretched, *salt* is the value the credential should be salted with, N is a CPU/memory cost parameter, p is a parallelisation parameter and, $lenDK$ is the desired length of the derived key. Android's implementation also adds a parameter r. This signifies the block size that is used internally. N, r and, p are the tweak parameters. They tweak the functioning of scrypt to best suit the hardware and time requirements.

Scrypt is the core component if the KDF used for Android FDE. In Figure 6 we can see a graphical overview of how the scrypt function is used. The "scrypted IK" value is derived from intermediate key 3. It is used to verify that the user input key is correct if there is data corruption on disk.

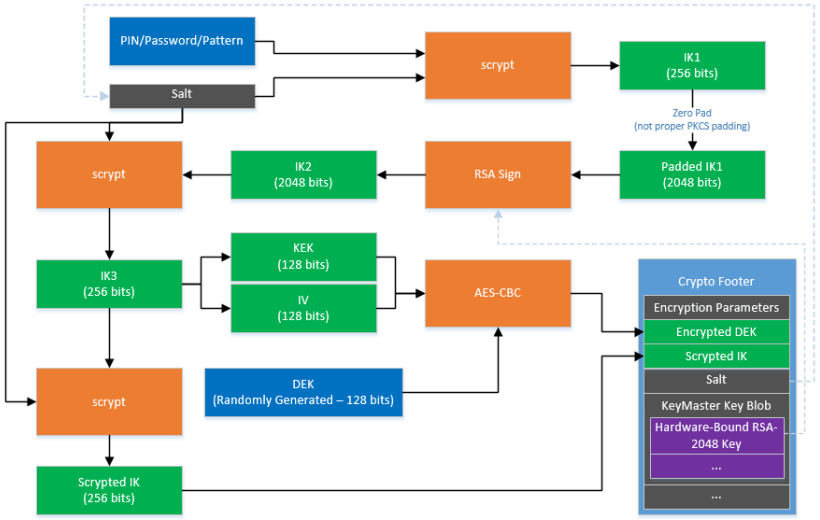

Figure 6: Overview of the Key Derivation Function of Full Disk Encryption[6]

B Modes of Operation

Ciphertext Stealing (CTS) is a technique used when encrypting with block ciphers. It solves the problem of needing to add padding to the plaintext if it's not a multiple of the cipher's block size. When the penultimate block is encrypted some of its ciphertext is removed and appended to the plaintext of the final piece of plaintext. Enough of the ciphertext is 'stolen' to pad the final chunk of plaintext up to the block size. The final block is then encrypted. Finally, the position of the final two blocks are swapped. We are left with ciphertext that's the same length as the original plaintext This enables a message that is not a multiple of the block size to be encrypted without padding.

Xor-Encrypt-Xor (XEX) is a mode of operation used for block ciphers. It is a tweakable mode that was created by Phillip Rogaway in 2004 [51]. XEX is different from other modes in that it needs a key double the size of the effective encryption key. The second half of the key (the lower half) is first used to encrypt an initialisation vector (often sector number). This encrypted key half is then XORed with the plaintext block. Next, this intermediate value is encrypted using the upper half of the original key. This encrypted value is then XORed again with the initial encrypted key half. The result is the ciphertext for the first block, For all further blocks the initial encrypted key half is XORed with some tweak value. This process is repeated for all further blocks. A graphical representation of XEX can be seen in the first two blocks of Figure 7.

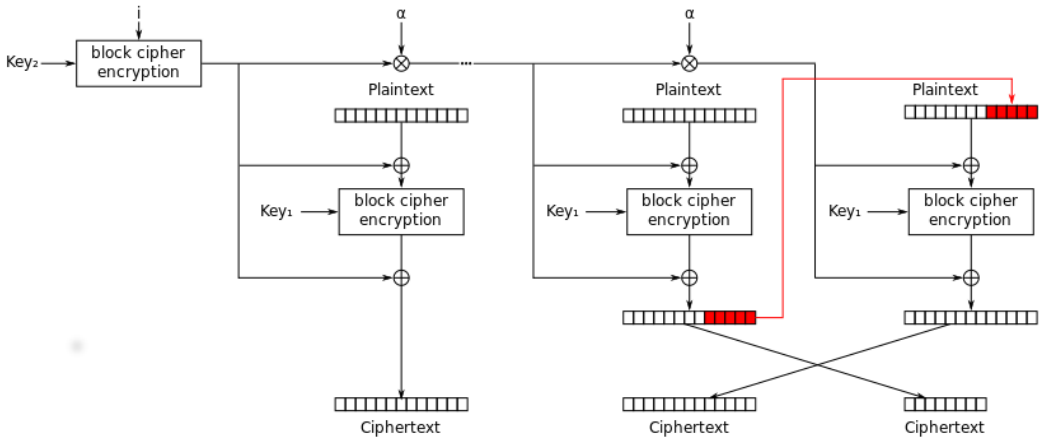

Figure 7: Overview of XTS Mode [52]

XTS is one of the more commonly used modes of operation for whole disk encryption. It is simply XEX mode but CTS is employed for the final two blocks. A graphical overview of XTS mode can be seen in Figure 7.