A Security Assessment of Android Full-disk Encryption[1]

Authors:
Oliver Kunz, MSc (Royal Holloway, 2015)
Keith Martin, Information Security Group, Royal Holloway, University of London

Abstract

Lost or stolen smartphones need protection. Full-disk encryption has become a popular method of protecting data stored on a smartphone in such events. In this article we introduce two attacks on full-disk encryption for Android. The first targets older pre-Android 5.0 versions. This attack is well known and exploitation tools are publicly available. Android 5.0 claims to have resolved the previous attack. We describe a new attack on Android 5.0 which does not take significantly longer to conduct than the pre-Android 5.0 exploit. We discuss potential countermeasures, observing that simple solutions are hard to find.

Introduction

Full-disk encryption of mobile devices, such as iPhone and Android smartphones, has become a popular method of protecting data in the event that the device is lost or stolen. Access to the unencrypted data is only given to the user presenting the correct decryption key. However, anyone in possession of the device is able to access the data in encrypted form and launch attacks. In this article we will describe two such attacks.

Google announced in autumn 2014 that Android 5.0 would be shipped with a full-disk encryption feature enabled by default, forcing the device to encrypt the storage on first boot. This announcement led to widespread criticism from law enforcement agencies. We thus chose Android full-disk encryption as the subject of our study and performed a holistic analysis of its security.

We will first give a short overview on the cryptographic background of full-disk encryption and then present the attacks that were found during the analysis process.

Encryption Techniques

Full-disk encryption is the process of scrambling all data stored on a disk and protecting it using encryption. In Android, the widely-deployed standardised block cipher the Advanced Encryption Standard (AES) is used.

AES can be implemented using different deployment techniques known as modes of operation. Android uses the classical Cipher Block Chaining (CBC) mode by default. It additionally supports a newer mode of operation known as XOR-Encrypt-XOR with Tweak and Ciphertext Stealing (XTS), which was specifically created for the purpose of encrypting storage media. However the user cannot select the mode of operation or AES key length (Android uses 128-bit keys) since these are defined in the source code and any change requires recompilation of the Android OS.

---

[1] This article is to be published online by Computer Weekly as part of the 2016 Royal Holloway information security thesis series. It is based on an MSc dissertation written as part of the MSc in Information Security at the ISG, Royal Holloway, University of London. The full MSc thesis is published on the ISG's website.

CBC Encryption Procedure

It is worth clarifying several features of the CBC mode of encryption for AES, since this is the default Android technique. The plaintext (disk data) is first chopped into a series of 128-bit data blocks. The CBC mode then chains the ciphertext blocks together by reusing the previous ciphertext block as part of the encryption process of the current plaintext block with the current plaintext (see Figure 1). More precisely, the previous ciphertext block is combined with the current plaintext block using an XOR operation, after which the result is input, together with the encryption key, into the AES encryption algorithm. Since there is no previous ciphertext block for the first plaintext block, an Initialisation Vector (IV) is needed to bootstrap the encryption process.

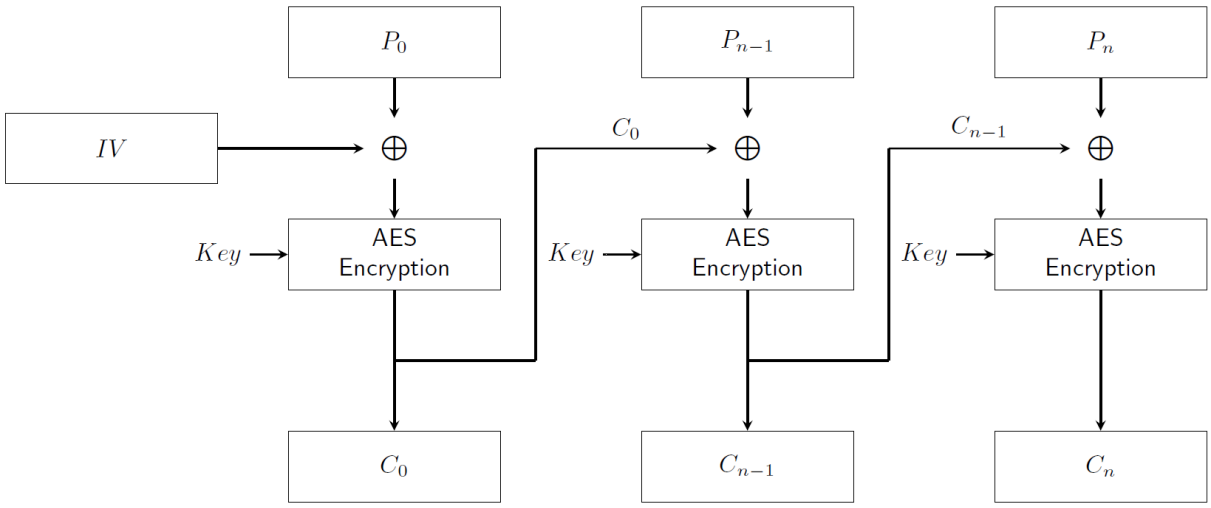

Figure 1: AES-CBC Encryption Process

Issues with full-disk encryption

In this type of application the plaintext data (the disk data) is likely to change over time. In CBC mode the encryption of a given plaintext block has an influence on the encryption of all subsequent plaintext blocks. This has the consequence that any change to a plaintext block will require not only re-encryption of that block but also re-encryption of all subsequent blocks. This is undesirable and hence a disk is typically not encrypted as one single data object, but rather each disk sector is processed individually.

Another issue with CBC mode is that it is undesirable for the IVs to be used more than once. In order to prevent IV reuse, Android uses a technique called Encrypted Salt-Sector IV (ESSIV) to create a fresh IV per sector.

Android Master Key

When we say that Android encrypts the full disk, this is not entirely true. Only the *userdata* partition containing the user and installed application data is encrypted. Other partitions, such as *system* or *metadata* (contains parameters for the full-disk encryption), are not encrypted.

The actual encryption process uses a 128-bit key for disk encryption known as the master key. Users never see the master key and do not need to remember it. Android employs a simple key hierarchy to protect the master key, which means that another key is used to encrypt the master key. This encrypted master key is then stored in the unencrypted *metadata* partition.

Key Derivation Functions

A key derivation function generates a cryptographic key based on a user-defined password. In Android, the password is either a screen PIN or password and the resulting key is used to protect the master key as described above. Therefore, this is also called a key encryption key (KEK).

Android implements two key derivation functions. The well-known Password-Based Key Derivation Function 2 (PBKDF2) and scrypt. Key derivation functions are engineered to be sufficiently slow to run that brute force attacks to find the password (and hence key encryption key) are made inefficient. In contrast, single use of a key derivation function is barely noticeable by users.

Pre-Android 5.0

We first looked at Android 4.4.4 to set a benchmark. In this pre-Android 5.0 version, users had to enable full-disk encryption manually and were asked to set a screen PIN or password.

Android 4.4.4 Offline Attack

The KEK generation function used to derive the KEK is scrypt. The encrypted master key is then stored in the crypto header located in the *metadata* partition. However, there is no means to prevent the scrypt computation being done offline on a much more powerful device than the smartphone. The attacker can therefore launch an attack on a specifically designed device to brute force every possible password combination.

Imaging the Partitions

Brute forcing the KEK is not enough since an attacker needs to know the encrypted master key and have access to the encrypted *userdata* partition. Thus, before launching the brute force attack, a form of accessing the two relevant partitions (*metadata* and *userdata*) is needed.

Over time Google has improved their security mechanisms, making it harder for an unauthorized person to copy a disk image off the device. However, an attacker is always left with the option of unsoldering and reading the chip directly. A less damaging approach is the access of device memory using the popular developer interface JTAG.

Brute Forcing the KEK

Once the *metadata* partition has been imaged the attacker reads the encrypted master key from the crypto header. His goal is it to find the correct password that generates the correct KEK to decrypt the master key, so that he can decrypt the *userdata* partition.

In our research, we have chosen the most basic brute force strategy to demonstrate how the attacks work and to compare the attack durations. The strategy chosen was a simple linear brute force of the 10 000 options for a four digit PIN. The attack script starts at PIN 0000 and run scrypt on it. It decrypts the master key and uses this candidate master key to decrypt *userdata* (see below). If the correct master key is recovered, the script ends. Otherwise, the next candidate PIN is processed.

Decrypt Userdata

A crucial step in the attack is the decryption of the *userdata.* The attacker must somehow be able to tell whether the correct master key has been found or not. To do this, he must have a little bit of inside knowledge concerning the data that is stored.

We used and enhanced a script shipped with Santoku Linux. Initially, this script only decrypted the first 32 bytes and then checked whether the bytes at offset 16 to 32 are zero. We investigated why this pattern was chosen to decide if the decryption was successful or not. The reason seems to be the filesystem ext4 and its disk management. Usually, the first kilobyte of an ext4 partition is set to zero. The next block following the ext4 superblock contains detailed management information and also a

distinctive signature, the “magic number”. This is a hexadecimal value 0xEF53 and is always stored at offset byte 1080. By additionally looking for the magic number, we can exclude the rare case that the first kilobyte is not zeroed out.

An attacker is thus able to have inside knowledge of the plaintext data without knowing any specific user data. Particularly valuably, this inside information is stored at a fixed location. This significantly reduces the amount of data necessary to decrypt in order to check for success.

Attack Time

Our Android 4.4.4 test device was a Nexus 4. We set the screen PIN to 9999 and copied the two partitions. We then ran the enhanced attack script on our external attack device. The simple linear brute force strategy processed 10 000 PINs in 51 minutes.

Android 5.0 Improvements

The new Android 5.0 has a number of improvements of relevance to full-disk encryption.

New Passwords

With Android 5.0 the screen pattern method can also be used as a password and a default password was introduced. The default password is mainly used during the first encryption and for as long as the user does not set a screen password.

Hardware Binding

Another security improvement that affects full-disk encryption is the KEK hardware-binding. When an Android device has a Trusted Execution Environment (TEE), an isolated execution environment that only runs pre-approved applications, the KEK generation is bound to the hardware. The pre-approved application for this purpose is the keymaster. We do not have much inside knowledge of the keymaster application. However, we know that there is an internal static key which is used to encrypt cryptographic material that is stored in the crypto header.

With Hardware binding and scrypt came a new KEK generation process (see Figure 2). Scrypt is applied on the user’s password, resulting in an intermediate key (I_KEY). The keymaster is then called to sign I_KEY and scrypt is applied on the signature, resulting in the KEK.

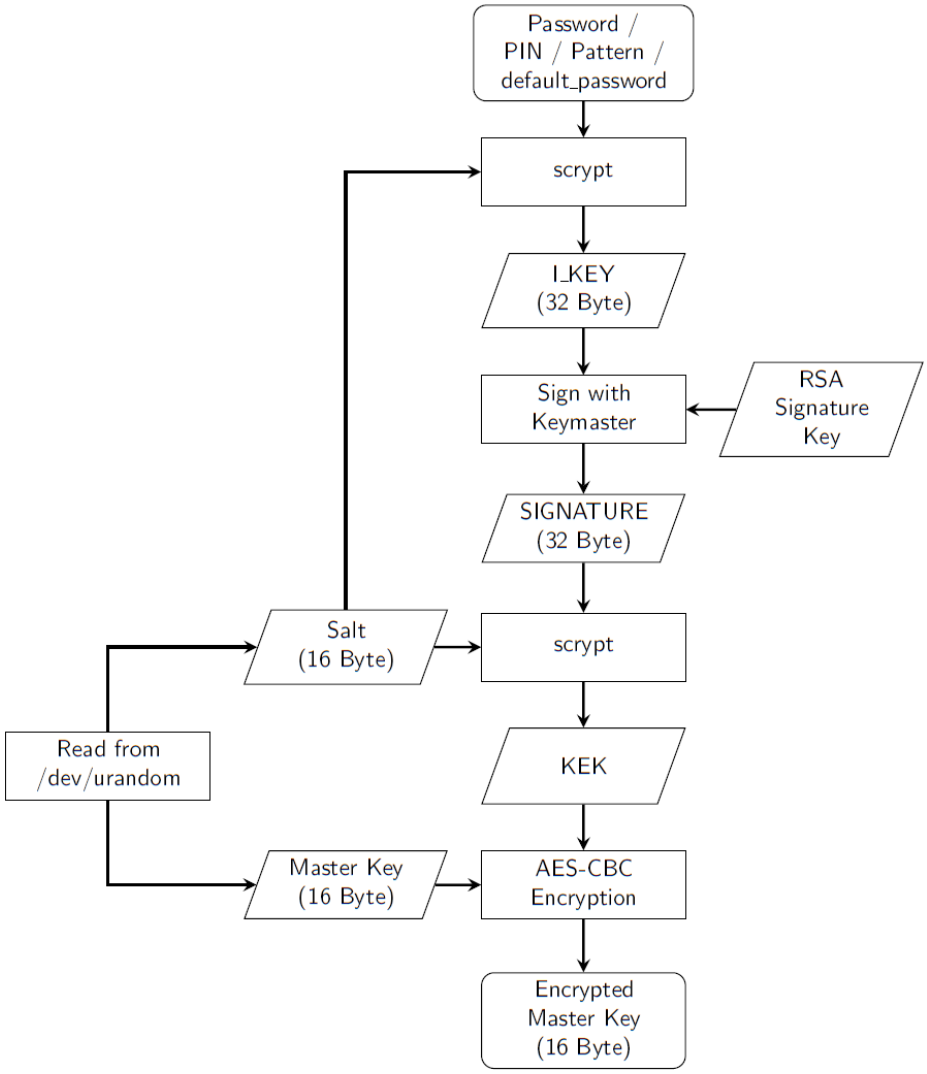

Figure 2: Key Encryption Key and Master Key Generation

With this procedure, only the device that has the correct internal keymaster key can generate the KEK which protects the master key.

Implications of Android 5.0 Improvements

It is our belief that these improvements to Android 5.0 are sufficient to defeat the offline attack that we previously described for Android 4.4.4. Indeed we confirmed that the previously implemented attack is no longer effective. An attacker appears to be reduced to brute forcing the 128-bit AES master key, or finding another attack vector.

New Attack on Android 5.0

During our investigation of the full-disk encryption feature in Android 5.0, we found an additional attack which successfully recovers the password and decrypts the master key. We named this attack the Semi-Offline Exhaustive Password Search Attack (hereafter, semi-offline attack).

The increased challenge for an attacker of Android 5.0 is that the KEK generation now requires access to the smartphone itself, which rules out a fully offline attack. However, if a smartphone is lost or stolen then an attacker certainly has that access.

The New Attack Procedure

The start of this new attack is exactly the same as it is during the offline attack on Android 4.4.4. First, the two partitions need to be copied off the device. There are no additional difficulties that prevent the technique previously described from being used to do this.

The new challenge is how to compute the expensive tasks on an external device while still including the smartphone for the hardware binding. We thus need an element on the smartphone and we need a communication channel between the external device and the smartphone. Once this has been achieved, the brute force strategy remains the same.

The first step in the new procedure is to generate a PIN and apply scrypt. The derived key now needs signing from the keymaster on the smartphone. We thus send the value to the device and request a signature. That signature is returned to the external device and again scrypt is applied. The derived key, if the correct PIN was used, should decrypt the master key. The final step of checking the correctness of the master key is similar to the offline attack.

However, it is clear that we not only need an attack script on the external attack device, but also an application that runs on the smartphone. This application must manage the signing process and communication with the keymaster inside the TEE, as well as communication with the external attack device.

The Attack Application

Our solution to this was a client-server application that covers the new attack procedure. The server application, written in C, runs on the smartphone. The client part, written in Python and based on the previously used extended Santoku Linux script, runs on the external attack system.

Besides dealing with the brute force strategy and computing scrypt, the client has another major task to do. It needs to parse the crypto header from the *metadata* partition and send the encrypted signature key material to the server so that the correct signing key can be initialized for the keymaster.

The server initializes the keymaster with the received key material. Since the internal keymaster key is static and not even destroyed after a device wipe, any previous key material can be used to initialize the keymaster. Hence, any *userdata* and *metadata* partition previously imaged from a device can be targeted.

Analysis of the Attack

In the new semi-offline attack, the amount of cryptographic computation tripled compared to the previous offline attack. Not only are two key derivation function calls needed (compared to one scrypt computation for the offline attack) but there is also a signature computation. However, even with the additional communication overhead between the attack system and the smartphone, the implemented semi-offline attack took only 2 hours 8 minutes to process 10 000 PINs.

The semi-offline attack is successful because an attacker can image the two partitions from a lost or stolen device. After that there is no need to keep the smartphone in its current state and it can be rooted, or a rooted OS loaded. This will probably trigger a userdata wipe operation by Android. However, that wipe does not affect the internal keymaster key which is used to decrypt the signature key. Since this key is never destroyed or changed, any old crypto header of a device can be used to initialize the keymaster and launch the attack.

Potential Countermeasures

The obvious solution to the semi-offline attack is to prevent imaging of any smartphone partition. However, as previously observed, if there is no JTAG interface then the final option for an attacker is always the unsoldering and direct accessing of the chip.

Another possible solution would be destruction of the internal keymaster key. This raises a very difficult question: when is it best to destroy the internal keymaster key? Probably the most convenient time is when the userdata partition gets wiped. This is most likely to be the case before a custom OS can be loaded. However, there might be other exploits an attacker can use to gain root privileges. Additionally, there might be another way of loading a custom OS which does not require triggering the userdata wipe.

Closing Remarks

Full-disk encryption is most often deployed on mobile computing systems. However, there are distinctive weaknesses in the current implementations. Our research has demonstrated that recent improvements by Google to counter offline attacks have not been completely successful. We have discussed our research with Google and received feedback that they were aware of the theoretical possibilities of such an attack and have therefore already started to address this issue prior to our research.

Note that we do not want to impart a false message to users of smartphones. Despite the weaknesses of full-disk encryption on smartphones we can only recommend enabling it. Any attack requires time to be successfully deployed. This time is valuable if the smartphone is enabled for remote wipe on the owner's command. Additionally, not every attacker is sufficiently sophisticated to launch a full scale brute force attack. In contrast, accessing unencrypted data is easy.

That said, it is important to know the limits of a security feature. Our security assessment of Android 5.0's full-disk encryption has been conducted to provide security practitioners with a document that supports their risk management decisions.

Biographies

Oliver Kunz

Oliver Kunz graduated in 2015 from Royal Holloway University of London with an MSc in Information Security and received the prize for the most outstanding MSc project for 2015 from the Information Security Group (ISG). He has worked in Information Security since 2010 and has helped his clients to resolve incidents, perform risk assessments, and analyse the security of applications and devices.

Keith Martin

Keith Martin is a professor in, and former Director of, the Information Security Group at Royal Holloway, University of London. His current research interests include key management, cryptographic applications and securing lightweight networks. He is the author of the recently published "Everyday Cryptography" by Oxford University Press. As well as conventional teaching, Keith is a designer and module leader on Royal Holloway's distance learning MSc Information Security programme, and regularly presents to industrial audiences and schools.