



University of Piraeus
Department of Digital Systems

Postgraduate Programme
“Techno-economic Management & Security of Digital Systems”

Master's Thesis

**SECURITY EVALUATION OF ANDROID
KEYSTORE**

Georgios Kasagiannis
MTE/1515, kasagiannis@ssl-unipi.gr

Under the supervision of:
Dr Christoforos Dadoyan, dadoyan@unipi.gr

Piraeus 26/2/2018

I would like to thank the Professors of this MSc because I learned a great deal about all forms of security and how wide an attack-surface can be, but also how human factor and human nature can be of use or can be exploited. Maybe the biggest lesson I've learned is that any system that is based on humans can be exploited even with not much effort if you really want it. I would like to thank Vicky & Stathis for all the kinds of help they gave me for this thesis, Geralt, and also my academic friends that helped me through the semesters.

Last but foremost, I would like to thank the fellow students who not only did not help my team through the projects, but they "took care only of them"... Maybe they taught and trained me the most!

The more I learn, the less I know...

Table of Contents

Table of Contents	iii
Table of Figures	iv
Table of Tables	v
Abstract	vi
1. Introduction	1
1.1 Foreword	1
1.1.1 <i>What is Android (1)?</i>	1
1.1.2 <i>What is computer security and how important is it in the 21th century?</i>	1
1.1.3 <i>Mobile Platform Security</i>	2
1.2 Subject of Thesis	3
1.3 Chapter Reference.....	5
2. Background	6
2.1 Cryptography	6
2.1.1 <i>Symmetric Cryptography</i>	7
2.1.2 <i>Asymmetric Cryptography</i>	7
2.2 Trusted Execution Environment	9
2.2.1 <i>ARM TrustZone®</i>	11
2.3 Android	18
3. Android Keystore	21
3.1 APIs for Key storage.....	21
3.2 Methodology	23
3.2.1 <i>Criteria</i>	23
3.2.2 <i>Evaluation</i>	25
4. Experiment	29
4.1 Android KeyStore test using TEE on Qualcomm devices	30
4.1.1 <i>Evaluation in Android 5.0</i>	32
4.1.2 <i>Evaluation in Android 6.0.1</i>	34
4.1.3 <i>Evaluation in Android 7.1.2</i>	36
4.2 Android KeyStore using software-based Keymaster	38
4.2.1 <i>Evaluation in Android 5.0</i>	40
4.2.2 <i>Evaluation in Android 6.0</i>	42
4.2.3 <i>Evaluation in Android 7.0</i>	44
5. Future Work & Conclusions	46
References	47
Appendix A - Acronyms	49
Appendix B – Source Code	50
Appendix C - Thesis presentation	Error! Bookmark not defined.

Table of Figures

Figure 1: AXI-bus diagram	13
Figure 2: NS Bit functionality normal world	14
Figure 3: NS Bit functionality secure world	14
Figure 4: Arm Trusted Firmware	15
Figure 5: ARM Trusted Firmware Architecture	16
Figure 6: TrustZone boot procedure	17
Figure 7: The separation of the hardware by TrustZone in two worlds	17
Figure 8: Schematic overview of the attacker models	25
Figure 9: Per use key authentication	28
Figure 10: Legit app and Rogue app installed	29
Figure 11: Forcibly inserting bouncycastle library to keystore-decryptor	30
Figure 12: Copying key files with other UID	31
Figure 13: 5HW key generation and validation of signature	32
Figure 14: 5HW Finding UID of rogue and legit apps	33
Figure 15: 5HW Copy key files and change ownership	33
Figure 16: 5HW rogue app parsing key of legit app	34
Figure 17: 5HW using keystore-decryptor	34
Figure 18: 6HW key generation and validation of signature	35
Figure 19: 6HW Copy key files and change ownership	35
Figure 20: 6HW rogue app parsing key of legit app	36
Figure 21: 7HW key generation and validation of signature	36
Figure 22: 7HW Copy key files and change ownership	37
Figure 23: 7HW rogue app parsing key of legit app	37
Figure 24: Android Studio downloading SDKs	39
Figure 25: Creating the AVDs	40
Figure 26: 5SW Copy key files and change ownership	41
Figure 27: 5SW rogue app parsing key of legit app	41
Figure 28: 5SW using keystore-decryptor	42
Figure 29: 6SW key generation and validation of signature	42
Figure 30: 6SW Copy key files and change ownership	43
Figure 31: 6SW rogue app parsing key of legit app	43
Figure 32: 7SW key generation and validation of signature	44
Figure 33: 7SW Copy key files and change ownership	44
Figure 34: 6SW rogue app parsing key of legit app	45

Table of Tables

Table 1: Chapter reference	5
Table 2: Generating an RSA key pair	26
Table 3: Symmetric Key generation	27

Abstract

This thesis was prepared in such a way that anyone – with basic Android and Security knowledge – can understand the problems around the key storage module of Android OS called Android Keystore. Keystore is the secure way of Android for storing the sensitive data of Applications.

Most of the use cases are examined – regarding the application of Android Keystore – on AVDs (android Virtual Devices), but a physical machine (Nexus 5) is included as well, with or without TEE (Trusted Execution Environment), for Android versions 5, 6 and 7. The contents and areas included in this thesis are as follows:

Chapter one is the introduction chapter. The reader gets familiar with the subjects that this thesis explores, mainly the security in Android Apps.

In the second chapter, Security background is analyzed for the reader to understand this thesis.

The third chapter is focused on the theoretical approach of Android Keystore vulnerability.

Chapter four presents the proof of concept for the Android Keystore vulnerability.

Chapter five focus on future work that will be done concerning this vulnerability and conclusions.

The whole project was executed on 3 AVDs and 1 physical machine with the following Android versions:

- Android 5 Lollipop
- Android 6 Marshmallow
- Android 7 Nougat

1. Introduction

1.1 Foreword

This is an introduction in definitions, facts and paradigms.

1.1.1 What is Android (1)?

Android is a mobile operating system developed by Google, based on a modified version of the Linux (2) kernel and other open source software and designed primarily for touchscreen mobile devices such as smartphones and tablets. In addition, Google has further developed Android TV for televisions, Android Auto for cars, and Android Wear for wrist watches, each with a specialized user interface. Variants of Android are also used on game consoles, digital cameras, PCs and other electronics.

Android's default user interface is mainly based on direct manipulation, using touch inputs that loosely correspond to real-world actions, like swiping, tapping, pinching, and reverse pinching to manipulate on-screen objects, along with a virtual keyboard.

Applications ("apps"), which extend the functionality of devices, are written using the Android software development kit (SDK) and, often, the Java programming language. Java may be combined with C/C++, together with a choice of non-default runtimes that allow better C++ support

1.1.2 What is computer security and how important is it in the 21th century?

Information security (3) means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, perusal, inspection, recording or destruction.

Governments, military, corporations, financial institutions, hospitals, and private businesses amass a great deal of confidential information about their employees, customers, products, research, and financial status. Most of this information is now collected, processed and stored on electronic computers and transmitted across networks to other computers. Should confidential information about a business' customers or finances or new product line fall into the hands of a competitor, such a breach of security could lead to lost business, law suits or even bankruptcy of the business. Protecting confidential information is a business requirement, and in many cases also an ethical and legal requirement.

For the individual, information security has a significant effect on privacy, which is viewed very differently in different cultures. Information security is the ongoing process of exercising due care and due diligence to protect information, and information systems, from unauthorized access, use, disclosure, destruction, modification, or disruption or distribution. The never ending process of information security involves ongoing training, assessment, protection, monitoring & detection, incident response & repair, documentation, and review. This makes information security an indispensable part of all the business operations across different domains.

1.1.3 Mobile Platform Security

The usage of smartphones has grown beyond imagination, in U.S.A. the percentage of smartphone adult users was 35% and nowadays has become 77%¹. Also according to Newzoo's Global Mobile Market Report, globally the smartphone penetration has reached 50% for 2017² and the number of smartphone users for 2018 is 2,5 billion people. Looking at the numbers we realize how important Security can be for smartphones and the owners of the devices.

Most companies sell or give as freeware Android Applications either as a product that earn profit (paid Applications), or as a tool to be used for their business model (mostly free applications, web-banking etc). Also, most applications have sensitive data stored inside the device that hosts them, data that users create or data that are needed for their functionalities. This fact has caught the attention of malicious individuals or even criminals who started creating malware for the mobile platforms. There are several attacks that have happened through the short life of smartphones, most have to do with malware and software, but there are also hardware, communication, network attacks and even high tech physics has play some role into forming attack models.

There has been a lot of research about rogue Apps inside Google Play Store, Google has blocked over 700.000 rogue apps in 2017³ and many "legitimate" apps found to be malware from researchers at the same year⁴. Another one called Juice Jacking is a physical or hardware vulnerability specific to mobile platforms. Mobile ransomware is a type of malware that locks users out of their mobile devices in a pay-to-unlock-your-device ploy, it has grown by leaps and bounds as a threat category since 2014 (4). Utilizing the dual purpose of the USB charge port, many devices have been susceptible to having data exfiltrated from, or malware installed onto a mobile device by utilizing malicious charging kiosks set up in public places or hidden in normal charge adapters (5). SMS and MMS attacks have been unleashed, in December 2012 the Eurograbber SMS trojan intercepted SMS messages on Android phones containing Transaction Authentication Numbers (TAN), also Stagefright⁵, a group of software bugs that affect versions 2.2 ("Froyo") and newer of the Android operating system. The name is taken from the affected library, which among other things, is used to unpack MMS messages. Exploitation of these bugs allows an attacker to perform arbitrary operations on the victim's device through remote code execution and privilege escalation. In 2015, researchers at the French government agency Agence nationale de la sécurité des systèmes d'information (ANSSI) demonstrated the capability to trigger the voice interface of certain smartphones remotely by using "specific electromagnetic waveforms". The exploit took advantage of antenna-properties of headphone wires

¹ <http://www.pewinternet.org/fact-sheet/mobile/>

² https://en.wikipedia.org/wiki/List_of_countries_by_smartphone_penetration#2017_rankings

³ <https://www.theinquirer.net/inquirer/news/3025746/google-play-blocked-700-000-rogue-apps-in-2017>

⁴ <https://www.scmagazineuk.com/spyware-found-in-more-than-1000-apps-in-google-play-store/article/681506/>

⁵ <http://blog.zimperium.com/how-to-protect-from-stagefright-vulnerability/>

while plugged into the audio-output jacks of the vulnerable smartphones and effectively spoofed audio input to inject commands via the audio interface⁶.

In a more general aspect there are three major types of threats (6):

- Data Theft smartphones are devices for data management, and may contain sensitive data like credit card numbers, authentication information, private information, activity logs (calendar, call logs).
- Identity Theft smartphones are highly customizable, so the device or its contents can easily be associated with a specific person. For example, every mobile device can transmit information related to the owner of the mobile phone contract and an attacker may want to steal the identity of the owner of a smartphone to commit other offenses.
- Availability Prevention attacking a smartphone can limit access to it and deprive the owner of its use.

The types of threats for the smartphone industries are basically the following (7):

- Botnets attackers infect multiple machines with malware that victims generally acquire via e-mail attachments or from compromised applications or websites. The malware then gives hackers remote control of "zombie" devices, which can then be instructed to perform harmful acts.
- Malicious applications hackers upload malicious programs or games to third-party smartphone application marketplaces. The programs steal personal information and open backdoor communication channels to install additional applications and cause other problems.
- Malicious links on social networks an effective way to spread malware where hackers can place Trojans, spyware, and backdoors.
- Spyware hackers use this to hijack phones, allowing them to hear calls, see text messages and e-mails as well as track someone's location through GPS updates

All these made the manufacturers of smartphones and mobile software developers to try and secure platforms and apps. In every Android or Apple iOS version update, new security fixes or features are integrated in order for the systems and their users to be more secure. Also this led companies to create awareness to their employers but also to the users of the smartphones and mobile applications. Alpha Bank, a Greek Bank after logging to the web banking application, a pop up message appears stating that the bank will never ask for user credentials, and informing the users for various security matters, giving instructions and tips for things to do or don't do, or even a maintenance schedule.

1.2 Subject of Thesis

⁶ <https://www.wired.com/2015/10/this-radio-trick-silently-hacks-siri-from-16-feet-away/>

Two are the main subjects of this thesis, Secure Computation and Secure Key Storage (SKS). Key Storage is not like a key management system (KMS). KMS also known as a cryptographic key management system (CKMS), is an integrated approach for generating, distributing and managing cryptographic keys for devices and applications. Compared to the term key management, a KMS is tailored to specific use-cases such as secure software update or machine-to-machine communication. In a holistic approach, it covers all aspects of security - from the secure generation of keys over the secure exchange of keys up to secure key handling and storage on the client. Thus, a KMS includes the backend functionality for key generation, distribution, and replacement as well as the client functionality for injecting keys, storing and managing keys on devices. SKS only stores keys securely. SKS, functionalities provided by CKMS and the other aspects of a secure execution environment provided inside a device make the “Secure Computation”.

Inside this secure environment trusted apps can run and handle sensitive operations such as asking for a PIN-code or running a specific cryptographic algorithm. In order to make computation faster but also more secure different solutions were provided by the manufacturers. One commonly used solution for secure computation and secure key storage is the Secure Element. This is a smart card like tamper resistant platform that can be embedded in systems as a chip or integrated in SIM cards. Other solutions concern embedded chips on top of smartphone motherboards either inside the main CPU chip.

As far as concerning the last category, Arm, has provided a solution called Arm TrustZone. ARM architecture processors are used almost by all modern smartphones regardless the mobile OS. As ARM states to their website⁷, Arm TrustZone technology is a System on Chip (SoC) and CPU system-wide approach to security. TrustZone is hardware-based security built into SoCs by semiconductor chip designers who want to provide secure end points and a device root of trust. The family of TrustZone technologies added to ARMv6 processors and greater, such as ARM11, CortexA8, CortexA9 and CortexA15. It can be integrated into any Arm Cortex-A and the latest Cortex-M23 and Cortex-M33 based systems, from the smallest of microcontrollers, with TrustZone for Cortex-M processors, to high-performance applications processors, with TrustZone technology for Cortex-A processors.

ARM TrustZone Technology provides the basis for a Trusted Execution Environment (TEE). The TrustZone hardware features, together with some software, ensure that resources from the secure world and some specific devices cannot be accessed from the normal world. The TEE offers secure computation (and as a consequence also secure key storage). However, a TEE also provides a way to securely communicate with a user as we will see in [section 2.2](#). This is not possible for a secure element.

This thesis though will not emphasize much to different technologies providing TEE and SKS like Tim Cooijmans, but using the knowledge from this paper (8) and his thesis

⁷ <https://www.arm.com/products/security-on-arm/trustzone>

(9) we will examine the use cases for ARM Trust Zone for hardware solution and Android Secure functionalities for software solution, for devices without hardware support for Android versions 5, 6 and 7 and check if it the Key Storage solution provided by Android (Android Key Store) is vulnerable like in previous version.

1.3 Chapter Reference

Table 1: Chapter reference

Chapter Number	Title
1	Introduction
2	Background
3	Android Keystore
4	Experiment
5	Future Work & Conclusions
	Bibliography
Appendix A	Acronyms
Appendix B	Source Code

[Back to contents](#)

2. Background

In this chapter will be presented and analyzed the basics of what someone needs to know to comprehend the subject of the current thesis. In [Section 2.1](#) an introduction in cryptography and its primitives is taking place, where the reader understands what cryptographic keys are and what are they needed for. [Section 2.2](#) describes the principles of a Trusted Execution Environment and the requirements which needed to exist in order for an Execution Environment to be a trusted one. Also the ARM TrustZone technology and the functionalities of the Operating System related to it are discussed. Android OS will be covered in [Section 2.3](#) and its security features, for the reason that this thesis is concerned with mobile devices running Android OS.

2.1 Cryptography

Here a short introduction about cryptography is going to take place, but very shallow in order to cover only what is needed. We will focus on what cryptography is, where and why is needed but not so much on how it works. In this Section we will present only some kinds of algorithms which are the asymmetric and the symmetric ones. Also some of the uses of these algorithms will be referred to better realize the real life usage of them in digital systems. So what is Cryptography?

It is a method of using advanced mathematical principles in storing and transmitting data in a particular form so that only those whom it is intended can read and process them. More generally, cryptography is about constructing and analyzing protocols that prevent third parties or the public from reading private messages; various aspects in information security such as data confidentiality, data integrity, authentication, and non-repudiation are central to modern cryptography (10). Modern cryptography exists at the intersection of the disciplines of mathematics, computer science, electrical engineering, communication science, and physics.

- Encryption: It is the process of locking up information using cryptography. Information that has been locked this way is encrypted.
- Decryption: The process of unlocking the encrypted information using cryptographic techniques.
- Key: A secret like a password used to encrypt and decrypt information. There are a few different types of keys used in cryptography.
- Digital Signature: A digital signature gives the receiver reason to believe the message was sent by the claimed sender. Digital seals and signatures are equivalent to handwritten signatures and stamped seals.
- Public Key Certificate: Also known as a digital certificate or identity certificate, is an electronic document used to prove the ownership of a public key. The certificate includes information about the key, information about the identity of its owner, and the digital signature of an entity that has verified the certificate's contents⁸.

⁸ https://en.wikipedia.org/wiki/Public_key_certificate

2.1.1 Symmetric Cryptography

Symmetric-key cryptography refers to encryption methods in which both the sender and receiver share the same key. This is the simplest kind of encryption, it is an old and best-known technique. It uses a secret key that can either be a number, a word or a string of random letters. Also the key can be blended with the plain text of a message in certain ways to change the result of a hashing algorithm. This technique is known as Message Authentication Code (MAC) and it protects both a message's integrity and authenticity by allowing verifiers (who also possess the secret key) to detect any changes to the message content.

Why symmetric key ciphers are valuable (11):

- It is relatively inexpensive to produce a strong key for these ciphers.
- The keys tend to be much smaller for the level of protection they afford.
- The algorithms are relatively inexpensive to process.

Therefore, implementing symmetric cryptography (particularly with hardware) can be highly effective because you do not experience any significant time delay as a result of the encryption and decryption. The sender and the recipient should know the secret key that is used to encrypt and decrypt all the messages because data encrypted with one symmetric key cannot be decrypted with any other symmetric key. Therefore, as long as the symmetric key is kept secret by the two parties using it to encrypt communications, each party can be sure that it is communicating with the other as long as the decrypted messages continue to make sense.

The main disadvantage of the symmetric key encryption is that all parties involved have to exchange the key used to encrypt the data before they can decrypt it.

Some examples of Symmetric Encryption Algorithms are Blowfish, AES, RC4, DES, 3DES, RC5, and RC6. DES was published as a standard in 1977 (12). It originally used a 56-bit key. In 2006 a DES key could be broken in 9 days for under 9000 euro (13). The most widely modern used symmetric algorithm is AES-128, AES-192, and AES-256.

2.1.2 Asymmetric Cryptography

Asymmetric Cryptography or else Public Key Encryption is called asymmetric because it's functionality is based on two different keys of different length, called a key-pair one public and one private, which are impossible to derive one from another. The user cannot encrypt and decrypt the payload-message with the same key. If you use the private key for encryption you have to use the public key from the pair for decryption. This kind of cryptography maybe is the most significant evolvement in coding theory in the last 3000 years of its history but either way it does not replace conventional cryptography it just supplement it.

The two keys are used as described below:

- Public Key, is known to everyone and anyone can have it. It is being used to encrypt or decrypt messages and digital signature verification.
- Private Key, is known only to the owner and is being used for message encryption or decryption and digital signature creation.

For this reason the private key is binded together with the identity of the owner by a trusted third party, a Certificate Authority (CA). The CA checks the identity of the owner of a key pair with procedures to certify that he is the one that he claims to be, then the CA signs (we will see how later) a certificate that confirms that this certain person or entity has a certain private key. In Transport Layer Security (TLS) a certificate's subject is typically a computer or other device, though TLS certificates may identify organizations or individuals in addition to their core role in identifying devices. TLS, sometimes called by its older name Secure Sockets Layer (SSL), is notable for being a part of HTTPS, a protocol for securely browsing the web. The key generation is done by special mathematical functions that take as input a big random number. The more entropy exists in the random number the merrier secure are the keys.

Now let's see the classic Bob-Alice example... When Bob needs to communicate with Alice securely he sends an encrypted message with Alice's public key which is given as input to the asymmetric algorithm together with the message. The only way for someone to read the message is to use Alice's private key that was generated together with the public key as a key pair, that only Alice should possess it (else anyone can read the message). This gives Bob (and Alice of course) the assurance that only Alice can read his message providing confidentiality. If Bob cannot meet with Alice to take the public key from her, then this schema is not going to happen. However if Bob may take the public key of Alice from a third party let's say Eve, he must be sure for the authenticity of the public key in a way that the key belongs to Alice and not to Eve or someone else, because if it is used for encryption the other end could read the message if he has access to it. Here comes the CA to play its part... CA holds the public key of both and when Bob and Alice need to communicate they exchange their certificates which basically are their public keys and IDs encrypted with the private key of the CA. By knowing the public key of CA Bob can decrypt the certificate and verify that the key is Alice's or not. For confidentiality but also authentication, but with the condition that they have exchanged their public keys securely, Bob should first encrypt the message with his private key and then encrypt again the generated cipher with Alice's public key so that Bob will be sure that only Alice can read his message but also Alice will be sure that Bob sent the message. With this way they can then exchange a symmetric key and use only that for their communication which is a lot faster. This symmetric key is also called session key.

It is time we see how signatures are generated. Let's say that the key owner, Bob, wants to prove that a certain message was sent by him. Bob has to create the hash of the message generated by a hash function and encrypt it with his private key. This quantity is the so called digital signature. He then sends the signature together with the message and the recipient, Alice, then decrypts the encrypted hash (signature) with Bob's public key and compares it with the hash of the message. If the signature is valid Alice knows that it was signed by Bob and if the hashes are equal then she knows that the message is unchanged. This provides integrity and authenticity. If Alice can prove that only Bob has this private key then he cannot deny that he signed the message providing non-repudiation.

Hash functions are publicly known and can be executed by anyone. If the tiniest part of the input-message changes (a single bit) it has to assure that the output will change dramatically. The size of the output (hash) is fixed, as a result messages that have bigger

complexity than the hash can have the same output. Depending on the hash function and the output length though it is infeasible to find a message that will have the same hash with another message.

In general the best practice for using key pairs, is to have one for every job you need to do. One pair for encryption, one pair for authentication and one for signature generation. As we examined above encryption and decryption operations are user also for signature generation. For encryption we use the public key, for decryption the private key. For signature generation we use the private key and for signature verification the public key. If the key owner is convinced to encrypt for you hash of a forged message he unintentionally signs a forged message. So if he used other key for encryption and other for signing the recipient would realize that something is not good because the public key for signing would not decrypt the hash which was encrypted with the private key for message encryption

There are lot of public key encryption algorithms that are used for signature generation and encryption. RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key is public and it is different from the decryption key which is kept secret (private). In RSA, this asymmetry is based on the practical difficulty of the factorization of the product of two large prime numbers, the "factoring problem". To be secure RSA key pairs with 3072bits length are recommended⁹ by BIS (Bundesamt für Sicherheit in der Informationstechnik, the German Federal Office for Information Security) (14).

Another algorithm for signatures based on Elliptic Curve Cryptography (ECC) is Elliptic Curve Digital Signature Algorithm (ECDSA). It is more efficient than RSA and uses smaller key length and is also supported by OpenSSL library. For example as we stated above the minimum length for an RSA key is 2048 bit but the "equivalent" key length for ECDSA is 256bits. This is a factor 8 smaller, however this is only an estimation.

2.2 Trusted Execution Environment

In July 2010 GlobalPlatform first announced their own standardization of the TEE. In according to GlobalPlatform the TEE is a secure area of the main processor in a smart phone (or any connected device)¹⁰. It ensures that sensitive data is stored, processed and protected in an isolated, trusted environment. The TEE's ability to offer isolated safe execution of authorized security software, known as 'trusted applications', enables it to provide end-to-end security by enforcing protected execution of authenticated code, confidentiality, authenticity, privacy, system integrity and data access rights. Comparative to other security environments on the device, the TEE also offers high processing speeds and a large amount of accessible memory.

⁹ A minimum length of 2048-bit may remain in use until end 2021.

¹⁰ https://www.globalplatform.org/mediaguidetee.asp#_Toc419214135

There are two main components of platform security:

- Trusted Execution Environment
- Trusted Platform Module

They work in tandem; one is not designed as a replacement of the other. As an analogy, TEE is the bulletproof safe, while TPM is the 128-digit combination lock for the safe. Both are needed to ensure the safe is protected.

The TEE offers a level of protection against attacks that have been generated in the Rich OS environment. It assists in the control of access rights and houses sensitive applications, which need to be isolated from the Rich OS. For example, the TEE is the ideal environment for content providers offering a video for a limited period of time, as premium content (e.g. HD video) must be secured so that it cannot be shared for free. Vasudevan et al. provide a number of requirements that are needed to ensure a Trusted Execution Environment (15):

- Isolated execution: TEE should allow applications to be run in isolation from other applications. This ensures that malicious applications can not access or modify the code or data of an application while it is running.
- Secure storage: Secure storage of data should be provided to protect the secrecy and integrity of the binaries that represent the applications while they are not running. The same security properties should also be guaranteed for the application data. Note that application data can be even more sensitive than the binaries as passwords and secret keys may be stored in the application data.
- Remote attestation: For a service to verify that it is actually talking to the software on the device it intends to talk to the principle of remote attestation is invented. It allows parties communicating with the secure execution environment to check the authenticity of the software and/or hardware that implements the TEE.
- Secure provisioning: It should be possible to send data to a specific software module operating in the execution environment of a specific device while guaranteeing the integrity and secrecy of the data being communicated.
- Trusted path: It should be possible for the execution environment to communicate with the outside world and to receive communication from the outside world while guaranteeing the authenticity of the communicated data and optionally also the secrecy and availability. This should allow on one hand a party, either human or non-human, to verify that the communicated data originates from the execution environment. On the other hand the technology should ensure that data from peripherals received by the environment is authentic.

Some of the interfaces to communicate to the user can be used by both the secure and non-secure world. For example the secure world and the non-secure world can both control the display in TrustZone-enabled processors. This makes it hard for the user to verify that he is interacting with an application in the TEE. To solve this the TEE has to provide local attestation. Local attestation should enable a user to check that they are in fact interacting with the TEE. An example solution is the Secure Mode Indicator patented by Texas Instruments. This solution provides the user a LED that is hardware controlled and that is only lighted when the secure world is running.

Arguments were made that TPM is not necessary if the TEE is robust. Some vendors have chosen not to use external TPM and store the keys and protected data in a TEE-only addressable area. TEE can help with Binding and Sealing. ISO standards suggest using a full-fledged TPM. External TPM could be very useful in coordinating between several masters and other complex systems. On the other hand, solutions that only rely on TPM are very vulnerable for execution and boot attacks. It is easy to override the application run states and circumvent TPM.

2.2.1 ARM TrustZone®

ARM TrustZone technology has been around for almost a decade. It was introduced at a time when the controversial discussion about trusted platform-modules (TPM) on x86 platforms was in full swing (TCPA, Palladium). Similar to how TPM chips were meant to magically make PCs "trustworthy", TrustZone aimed at establishing trust in ARM-based platforms. In contrast to TPMs, which were designed as fixed-function devices with a predefined feature set, TrustZone represented a much more flexible approach by leveraging the CPU as a freely programmable trusted platform module. To do that, ARM introduced a special CPU mode called "secure mode" in addition to the regular normal mode, thereby establishing the notions of a "secure world" and a "normal world". The distinction between both worlds is completely orthogonal to the normal ring protection between user-level and kernel-level code and hidden from the operating system running in the normal world. Furthermore, it is not limited to the CPU but propagated over the system bus to peripheral devices and memory controllers. This way, such an ARM-based platform effectively becomes a kind of split personality. When secure mode is active, the software running on the CPU has a different view on the whole system than software running in non-secure mode. This way, system functions, in particular security functions and cryptographic credentials, can be hidden from the normal world. It goes without saying that this concept is vastly more flexible than TPM chips because the functionality of the secure world is defined by system software instead of being hard-wired.

These worlds are achieved by separating both software and hardware resources (16):

- All memory in the system is separated. This includes the system's RAM but also the registers of the CPUs. The RAM is split into two separate virtual memory spaces. One for the secure world and one for the normal world. This means that the normal world cannot access the memory used by the secure world. The persistent memory (such as ash memory) can be separated by using encryption.
- A dedicated cryptoprocessor and memory for storing keys can only be made accessible by the secure world. This prevents the normal world from accessing sensitive key material.
- The display controller can use both a section of the memory of the normal world and a section of the secure world as display buffer. This dual buffer setup allows the secure world to communicate information to the user without interference from the normal world. As the display buffer for the secure world is located in the secure world memory, the normal world applications cannot access it.

To ensure the requirements discussed to build a Trusted Execution Environment a concept called secure boot is normally used. This process verifies the integrity of the contents of the storage that contains the operating system and checks that the operating

system is issued by the device manufacturer by checking its signature. This prevents attackers from modifying or changing the operating system.

All these arise questions such as: Does TrustZone provide mechanisms for secure booting and secure storage? Isn't it a kind of virtualization technology? If yes, isn't it superseded by ARM's virtualization extensions? How does it work? Is it important to consider it when developing an operating system (17)?

How does it work

TrustZone technology is programmed into the hardware, enabling the protection of memory and peripherals. Since security is designed into the hardware, TrustZone avoids security vulnerabilities caused by proprietary, non-portable solutions outside the core. Security can be maintained as an inherent feature of the device, without degrading system performance, enabling device manufacturers to build security applications, such as DRM or mobile payment as protected applications that run on the secure kernel.

With TrustZone, user space applications operate in "normal" mode. The kernel runs "system" mode. The trusted kernel operates in "monitor" mode. Because of this architecture, even a "rooted" application cannot access protected regions within the trusted kernel. Any component can be designated as part of the trusted infrastructure, from regions of the PCI-E address space to NAND memory. Overall, TrustZone offers a secure and easy-to-implement trusted computing solution for device manufacturers, without requiring additional hardware.

One of the main hardware features of TrustZone technology is the Security bit on the communication bus (18). The ARM processor has a communication bus called the AXI-bus that is used by the main processor to communicate with peripherals (see figure below). These peripherals are located in the same package or chip based on the SoC architecture. The security bit is added to this bus to communicate to the peripherals whether the transaction they are receiving is either from the secure world or the normal world. All peripherals should check the security status of the transaction and ensure that they do not leak any sensitive information.

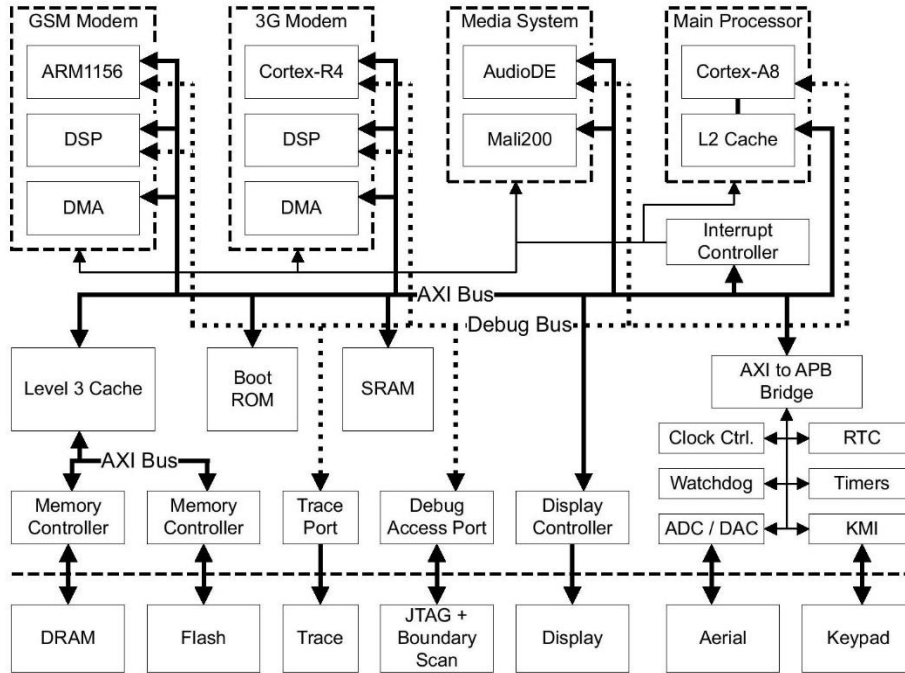


Figure 1: AXI-bus diagram

Another aspect of the TrustZone hardware is the separation of the two worlds in the processor itself. This is indicated by the NS -bit (Non-Secure-bit) in the Secure Configuration Register (SCR) of the processor. This bit can only be set in the Secure mode. When the NS-bit is 0 the processor is operating in secure world and when it is 1 the processor is operating in normal world. Two operating systems can run alongside each other using this architecture: One in the secure world and one in the normal world. As a result a special form of virtualization is created: There are two virtual environments that include virtual processors and virtual resources. Access to those virtual resources can be limited depending on the security status of the processor. The value of the NS-bit is used to signal the security status of communications on the AXI bus. This is in turn used by the peripheral to decide if it should act on a certain transaction.

Figure 2: NS Bit functionality normal world below illustrates the concept of the two worlds. The normal world is active (non-secure bit is set), the OS running on the platform can only access a subset of the physical resources.

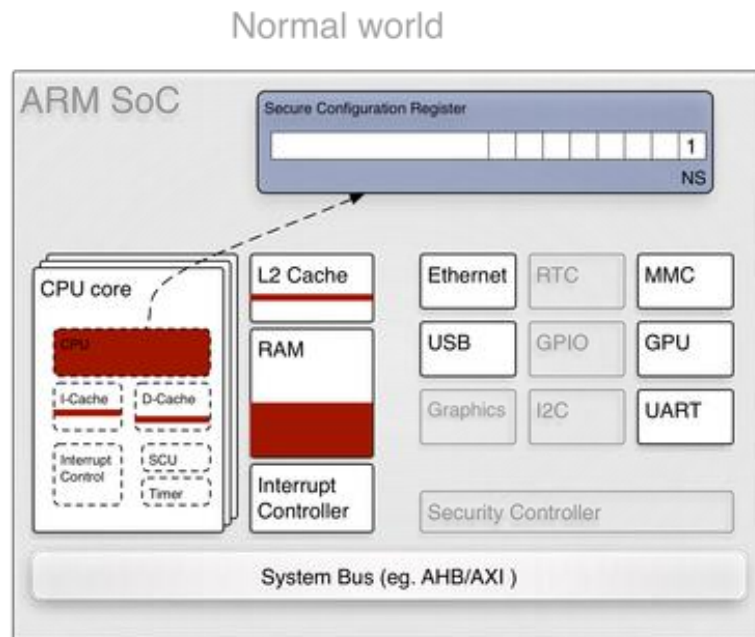


Figure 2: NS Bit functionality normal world

When a world switch takes place, the secure world comes into effect. The system software running in the secure world can access the devices hidden from the normal world. In Figure 2 below we can see clearly the switch to the secure world with NS Bit 0, but also the different areas of memory that are being used.

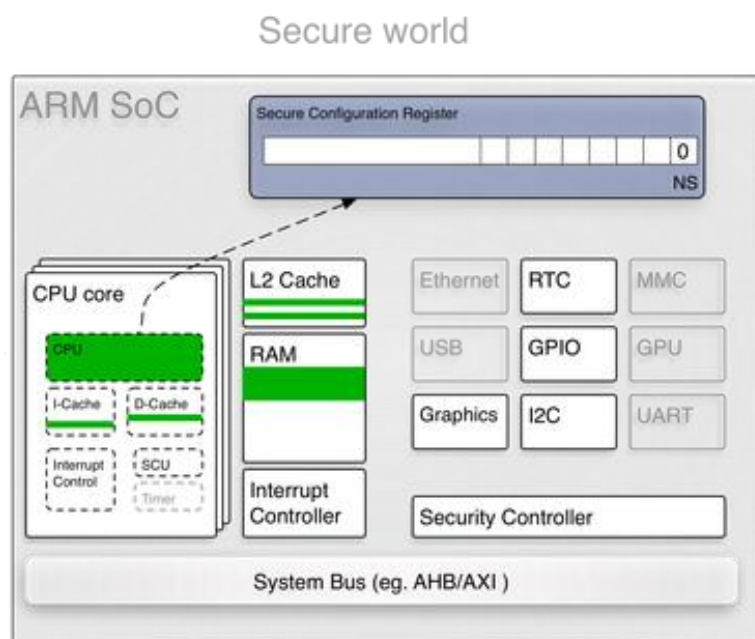


Figure 3: NS Bit functionality secure world

A special state is created in the secure world to facilitate switching between the worlds. This state is called the monitor mode. The normal world can start this monitor mode by calling the Secure Monitor Call (SMC) instruction. Hardware exceptions such as interrupts can also cause a switch to the secure world. When such switch due to the SMC instruction or exceptions happens the monitor mode of the secure world is enabled. The monitor mode ensures that the state of the world it is leaving is saved and

the state of the world it is entering is restored (16). State data includes all processor registers and optionally additional information depending on the peripherals in the SoC (16).

TrustZone software provides a minimal secure kernel which can be run in parallel with a more fully featured high-level OS-such as Linux, Android, or BSD-on the same core. It also provides drivers for the normal, rich OS (normal world) to communicate with the secure OS (secure world).

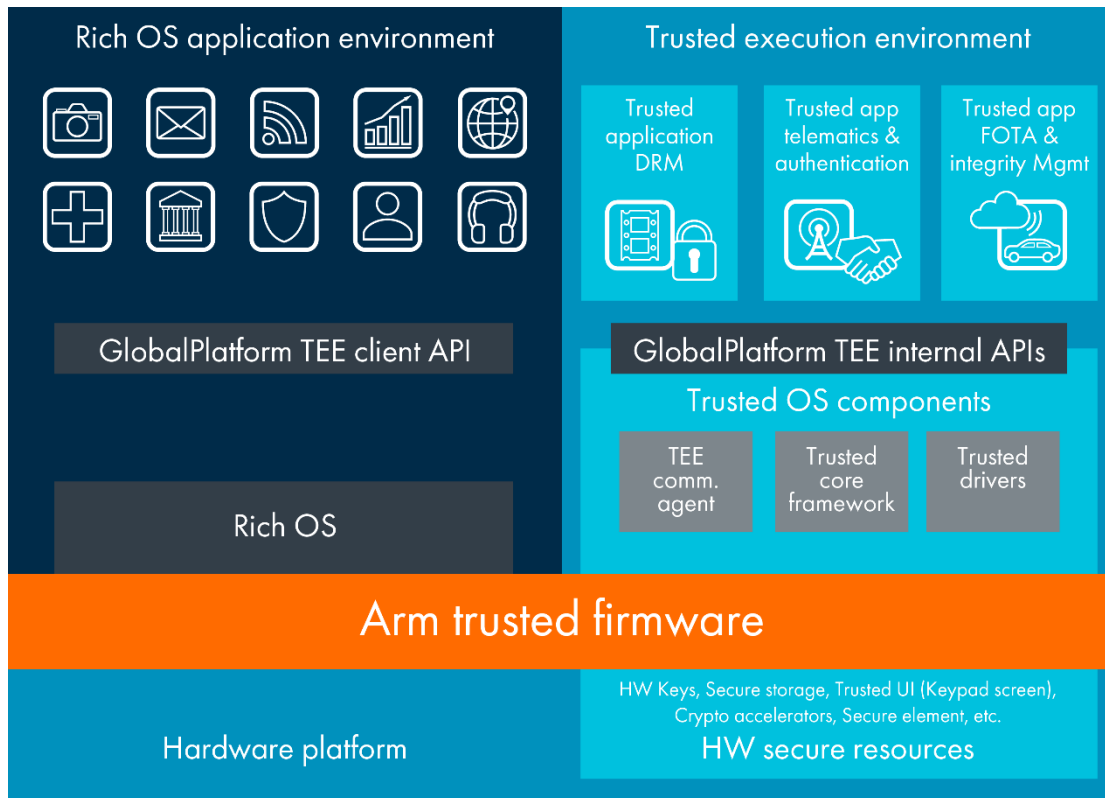


Figure 4: Arm Trusted Firmware

TrustZone software uses ARM TrustZone security extensions to completely protect the secure OS and any secure peripherals from code running in the normal world. It includes a secure monitor that switches between the secure and the normal world, and an example secure first-stage bootloader. Systems with a separate ARM processor dedicated for security can use the TrustZone software multicore, running the secure kernel on its own CPU.

Processors can have multiple processor cores to allow multiprocessing. This may present additional difficulties to ensure that all data is kept safe between context switches. To simplify the switching, the secure world can be fixed to a single core or a certain number of cores. Note that, in contrast to dedicated security processors, the TrustZone hardware does not by default include tamper protection [9]. Since ARM only sells the designs to create the processors and does not make the chip itself manufacturers may include additional hardware features to offer better tamper resistance. Also note that while TrustZone provides hardware-based security features, the security of the whole system is also depending on some software features such as the sanity of the switching between the secure and normal world implemented in the

monitor mode. A bad implementation could leak the register values that may contain sensitive information.

ARM Trusted Firmware Architecture

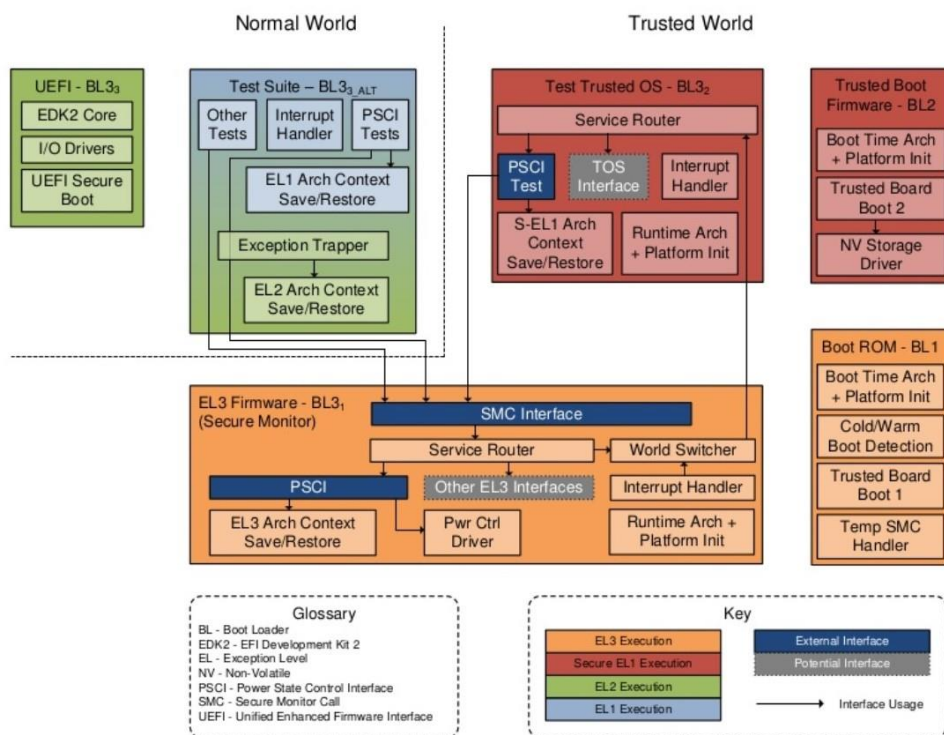


Figure 5: ARM Trusted Firmware Architecture

When an ARM processor with TrustZone support boots it starts by executing an application that is programmed in the on-chip-ROM in the secure world. This application is called the bootrom. The bootrom can be fixed by the processor manufacturer at design-time using a Masked ROM or by the customer of the manufacturer (the manufacturer of the system that incorporates the processor) by using write-locked ash. A public key is programmed in to the SoC using One-Time-Programmable (OTP) memory. This memory can only be written once. This is often achieved by burning fuses on the chip.

A TrustZone compatible bootloader starts the OS that runs in secure world. When it boots it starts the normal world OS. The use case of this procedure is described below.

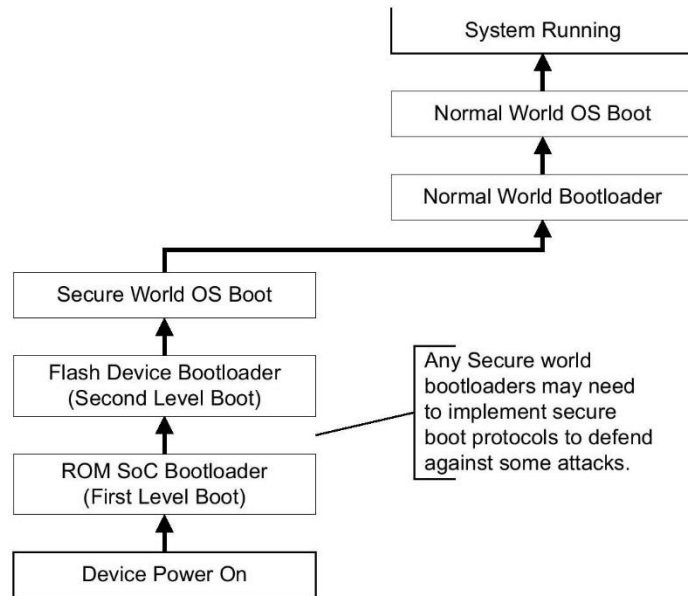


Figure 6: TrustZone boot procedure

However, hardware features such as the impossibility of the normal world to access the memory of the secure world allow this functionality to be implemented. To allow multiple applications to be run in the secure world a secure world operating system (secure world OS) has to be implemented. We consider the secure world OS as the software that provides the TEE: It provides an execution environment for applications to run in. As discussed previously, applications running in the TEE are called trustlets. The secure world OS schedules resources between both the trustlets running in the secure world and the operating system running in the normal world. The secure world OS should handle both context switches (between trustlets in the secure world and between the secure and normal world). It should also ensure that no data is leaked during the context switches. Note that if an untrusted user is allowed to run trustlets in the TEE, also the security of context switches between trustlets in the TEE should not leak any information. Even if all trustlets are created by the same issuer this is still a good property to ensure. The separation between the two worlds each with its operating system is pictured in Figure 6.

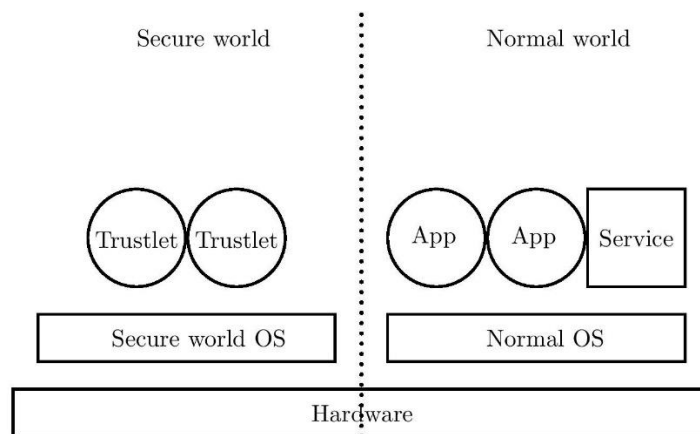


Figure 7: The separation of the hardware by TrustZone in two worlds.

A secure world OS is sometimes seen as a hypervisor. A term that is generally associated with running virtual machines; running multiple operating systems in their own environment on the same hardware. While the separation by TrustZone also provides two operating systems running in their own environment on the same hardware, the actual architecture is different. In the case of virtualization the hypervisor provides virtual hardware on which the (unlimited number of) operating systems run. The hypervisor sits in between the virtual hardware and the real hardware. In the case of TrustZone technology both the secure world OS and the OS running in the normal world can directly communicate with the hardware. However, some parts of the hardware are only accessible by the secure world. In principle there is no limit to the complexity and functionality of the secure world OS running in the secure world. Also, to reduce the attack-surface, the functionality is usually limited. There is only a small number of open source implementations of TrustZone compatible secure world OSes available (19) but ARM has opensourced the ARM Trusted Firmware¹¹.

2.3 Android

Initially developed by Android Inc., which Google bought in 2005, Android was unveiled in 2007, with the first commercial Android device launched in September 2008. The operating system has since gone through multiple major releases, with the current version being 8.1 "Oreo", released in December 2017. It is based on a LTS Linux Kernel and since 2017 version 3.18 or 4.4 are used.

Android has been the best-selling OS worldwide on smartphones since 2011 and on tablets since 2013. As of May 2017, it has over two billion monthly active users, the largest installed base of any operating system, and as of 2017, the Google Play store features over 3.5 million apps¹². Applications are written using Android SDK and Java. Java may be combined with C/C++. Also the Go language is supported and in May of 2017, Google announced support for Kotlin. Android SDK includes a lot of libraries and tools such as emulators, documentations, tutorials and sample code. The primary IDE since 2014 is Google's Android Studio which has integrate all the tools mentioned above in its GUI. Numbers of Android developers have grown big last years, building apps for almost everything which can be acquired by Android users from Google Play Store, or unofficial sources.

Source code of Android is available after the release of each version in AOSP, an open source initiative by Google. The AOSP code can be found mainly on Nexus and Pixel series of devices. Many OEMs customize and adapt the AOSP code to run on their hardware based on their needs and their peripheral devices. As a result most Android devices ship with a combination of open source and closed source software.

¹¹ <https://github.com/ARM-software/arm-trusted-firmware>

¹² <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>

The directory layout of the file system for Android is somewhat different from a usual Linux operating system:

- **/data** is used to store the data of all applications and services running on the operating system in.
- **/data/data** is the location for applications to store their data. Each application gets its own directory that is named using the application identifier (com.company.example).
- **/sdcard** is the location where the SD-card (if present in the system) is mounted. The internal storage is limited but faster on older Android devices so developers had to choose whether they stored the data internally or on the SD-card. Most recent Android devices have larger internal storage so they do not need a SD-card anymore. On systems that have internal storage and no SD-card slot the **/sdcard** path is symlinked to **/storage/emulated/legacy**.

The (emulated) SD-card directory can be accessed over USB to write and read all files from it. A number of directories in **/sdcard** are accessible by all applications such as **/sdcard/DCIM** where pictures are stored and **/sdcard/Music** for storing Music. The **/sdcard/Android/data** directory is where applications can store application specific data (like the **/data/data** directory). As with the internal application-data directory each application gets its own directory that only can be read by that application. However, all contents of the (emulated) SD-card can be read and modified over USB. This also includes the application data that is stored in **/sdcard/Android/data**, so caution should be used when storing data on **/sdcard**.

Applications have a file `AndroidManifest.xml` that describes the contents of the application and the permissions that the application requests. A user cannot deny a single permission in AOSP Android 5 and older in the set of permissions requested by the application. Some OEMs have implemented their own application permission restriction to applications. Starting from version 6 the user can choose almost all permissions that the application is allowed to have except if they are critical for the functionality of the app. The requested permissions are shown before the application is installed by the user or when an updated version of an application requests additional permissions. After installation there is no way for a user the denial of certain permissions other than un-installing the application in Android 5. As we mentioned since Android 6 this has changed. A list of all permissions available can be found in the Android API documentation¹³.

To control the access of the application to the file system, OS assigns automatically a UID to each name of the application. This is different than normal Linux where each user can run the applications under his UID. The mapping of UIDs and application names can be found in **/data/system/packages.list** and its structure is like that:

¹³ <http://developer.android.com/reference/android/Manifest.permission.html>

```
<pkg_name> <UID> <user_installed> <directory> <type_of_app>
```

An example:

```
com.example.user.rogue.app 10081 1  
/data/user/0/com.example.user.rogue_app default
```

We can clearly see that an application with package name `com.example.user.rogue.app` (this is the app we are going to use in the next chapter) has UID 10081.

Android OS uses its special virtual machine called Dalvic¹⁴ as basis for the apps and services. This virtual machine is based on Java resulting a number of Java APIs ported to Android. The ported APIs can be found under `java.*` packages which provide support for cryptographic operations through the `java.security` package. Additional functionality is implemented in the `android.security` package. The API version is indicated by API level. The first Android release has API 1 and the latest one (Oreo 8.1) has API level 27. An overview of Android versions associated with API levels can be found on the Android Developer web-page¹⁵

[Back to contents](#)

¹⁴ <https://source.android.com/devices/tech/dalvik/index.html>

¹⁵ <https://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels>

3. Android Keystore

In this Chapter the key storage solution provided by Android is going to be analyzed. In Android 6.0 M the security scheme changed dramatically over L version giving user more control over app permissions. Instead of granting all apps permissions when the user installs them, Android prompts the user to choose only which permissions he wants. It is also easy to revoke permissions even for apps designed for old versions. Also the big news was the Android Keystore which made its debut with Android API 18. Chad Brubaker, a senior engineer working on Android security, states in Androids website “Android has opened access to hardware security that used to be costly and difficult to use. Now with Keystore, any developer can use the best available hardware security features.” As a matter of fact if we check the AOSP page¹⁶ we will find out that many bug-fixes and security issues are resolved as time passes. We shall see though in [Chapter 4](#) that this is not true if someone gains root or system privileges with even SE Linux to not complain about anything...

3.1 APIs for Key storage

In Android API 1 developers were provided with cryptographic operations and key storage. For this the `java.security.KeyStore` class which provided an interface for key storage. It only provided the interface to store the keys and also to get the instance of the class. Each key store type is defined in a class that provides a `java.security.KeyStoreSpi` interface. The `KeyStore` class uses the methods of `KeyStoreSpi` interface which give the ability to developers to store keys.

There are multiple Key Store types given from Android API. The most knowns are the Bouncy Castle key store which is a cryptographic library for Java and C#, and the official Android Keystore. As we said above it was added to AOSP with API 18 and since then it is inside all API versions.

This Keystore has a service created for it named `KeyStore`, communicating with it using Inter Process Communication (IPC). The `KeyStore` service starts at boot time with the OS. Manufacturers can develop drivers for their hardware that communicates with `KeyStore` service, providing hardware based secure key storage. If no drivers are detected or not compatible or no hardware for TEE, Android defaults to a software implementation. The following variants of `AndroidKeyStore` will be evaluated:

- Software based: Not all Android devices support hardware-based key storage. Some old Arm chips simple do not have TrustZone support and some device manufacturers do not have the ability to create applications for the TEE. For this reason Android developers created a software based `KeyMaster` trustlet that runs in the TEE. If a device supports hardware based `AndroidKeyStore` though the software based is not available.

¹⁶ <https://android.googlesource.com/platform/system/security/>

- Hardware based: Qualcomm created a closed source keymanager trustlet in the TEE and driver for Android.

Note that while the `AndroidKeyStore` variants are called software-based and hardware-based all solutions are of course based on both software and hardware. With software-based and hardware-based we mean that hardware-based or only software-based security features are used to protect the key store.

Besides the key store type a provider can be optionally defined for some classes. This defines what library is used for cryptographic operations. The `Spongy Castle` library uses this to interact with the `Bouncy Castle` key store type. When creating an instance of the `Bouncy Castle` key store type `Spongy Castle` is defined as provider. Subsequently keys compatible with the `Spongy Castle` library are provided. Note that not all combinations of providers and key store types are supported, some key store types do not support providers at all. This can be the case for key store types that only provide an interface to manage the keys but do not provide the actual key material. An example of this is the `AndroidKeyStore` key store type that can run its cryptographic operations outside of application process on other hardware.

API level 14 added a new `KeyChain` class. This is a high-level API to provide asymmetric key storage. In contrast to the `KeyStore` class the `KeyChain` class interacts with the user using a graphical interface. For example the user can be asked confirm key generation. When a private key is required for operation the user is also asked to confirm. This class, however, also provides two interesting static methods. The `isKeyAlgorithmSupported(String algorithm)` can be used to test if the device supports a certain algorithm. In the API level 19 release a new static method was added to the class, `isBoundKeyAlgorithm(String algorithm)`. This method Returns true if the current device's `KeyChain` binds any `PrivateKey` of the given algorithm to the device once imported or generated according to documentation¹⁷.

In addition to private keys it is also possible to store symmetric keys in the `KeyStore`. To support this the key store has a `SecretKeyEntry`. Secret keys can be created using the `javax.crypto.spec.SecretKeySpec` class. Instances of this class are initialized by a byte-array for the secret key and a string indicating the algorithm that is associated with the secret key.

Also another problem which might concern the developers and also the end users is that until Android M when device security settings (device-lock) are set or changed by the user (No lock, PIN, Swipe etc) the keys inside `Keystore` are wiped. I realized it when testing the behavior of Android `Keystore` and Dorian Cussen confirmed it in his blog post¹⁸ stating that it was a mistake in `KeyPairGeneratorSpec` class “an undocumented-exception-throwing fail which requires you to reset the alias (`Keystore.deleteEntry()`)”

¹⁷ <http://developer.android.com/reference/android/security/KeyChain.html>

¹⁸ <https://doridori.github.io/android-security-the-forgetful-keystore/>

3.2 Methodology

As we said in the previous section a good way to evaluate and check the differences of Android Keystore is the open source code of Android (AOSP) in the git repositories. Most of the code is provided that is used by mobile phones running Android OS (Nexus and Pixel phones). Since licensing of the AOSP, code requires that changes made to code are also made public so the code for phones that are not developed under Google is available. Therefore some manufacturers also have donated part of their source code to the AOSP making it open.

To evaluate the condition and to analyze the system of Android OS, we will be using root privileges on the platforms. Root access disabled in the Android OS that is shipped with the mobile phones, so in order to gain root privileges, we root the device (in AVDs this is not needed). By unlocking the bootloaders of the phone users are allowed to install android images that are not signed by the manufacturer. Many tutorials exist for rooting an Android device on internet accessible by anyone¹⁹ using just Android Debug Bridge (ADB). ADB is a “bridge” for developers to work out bugs in their Android applications aka debugging. This is done by connecting a device that runs the software through a PC, and feeding it terminal commands. ADB lets you modify your device (or device’s software) via a PC command line. For the whole rooting process there is even a video in YouTube²⁰ giving instructions on how to extract the Stock Android Firmware that the phone shipped with in order not to lose the “clean” OS using the TWRP recovery tool²¹. This is not supported though for all device manufacturers and firmwares. The users then install an app called SuperSU that allows any app that is installed in the device to run with root privileges.

3.2.1 Criteria

As Tim Cooijmans refers (8), he defined three requirements for evaluation:

$R_{\text{device bound}}$ The private key material stored in the key store cannot be extracted by an attacker to be used outside of the device that generated it. This defines that keys are bound to the device.

$R_{\text{app bound}}$ The private key material stored in the key store can only be used by the application and on the device that generated it. This defines that keys are bound to the device and the application.

$R_{\text{user consent}}$ Using the private key material requires explicit permission from the user and cannot be used without explicit consent. This defines user consent.

¹⁹ <http://trendblog.net/how-to-root-your-google-nexus-device-4-5-7-10/>

²⁰ <https://www.youtube.com/watch?v=DyUainEJwLM>

²¹ <https://dl.twrp.me/>

The requirements $R_{\text{device bound}}$ and $R_{\text{app bound}}$ are evaluated using three assumed attacker models:

$A_{\text{outside no root}}$ The first attacker model is an attacker that is able to run and install his own application on the phone with any of the permissions that an application can request. This models a rogue application (or update of an application) that a user can install from an app store such as the Google Play Store.

$A_{\text{outside root no memory}}$ The second is an attacker that has root permissions and can use all data stored on the phone. We assume that these permissions are gained by either an exploit or are given to the attacker by using an application that asks for root permissions on a rooted phone. This models that either the attacker uses an exploit to gain root permissions on the device or that the attacker has the ability to run an application with root permissions. Note that this may seem as a situation that does not happen a lot, however, many users today enable the root permissions on their phone to work around the permission model. There are even applications in the Google Play Store that require root permissions. For example the Titanium Backup in the Play Store¹¹ application that is used to back up a phone requires root permissions.

$A_{\text{outside root memory}}$ The third attacker model assumes that an attacker has root permissions and can use all data stored on the phone but also has access to data that is only temporarily stored on the phone such as data stored in memory. For example a PIN that is used to unlock the phone but that is not stored on the phone.

For the requirement $R_{\text{user consent}}$ there is only one attacker model:

$A_{\text{inside no root}}$ This attacker model assumes that the actual developer of an app is or has become malicious. This is particularly interesting in the case where the developer of an application has interest in using the key pair that is stored for the app. An example of this can be an e-mail application that has a key pair to sign messages. The signature may have legal-effect that may be abused

For all criteria the described attacker models are evaluated. For each combination of a criteria and an attacker model the result is either:

- ✓ An attacker using the attacker model cannot violate the requirement.
- X An attacker using the attacker model can violate the requirement.

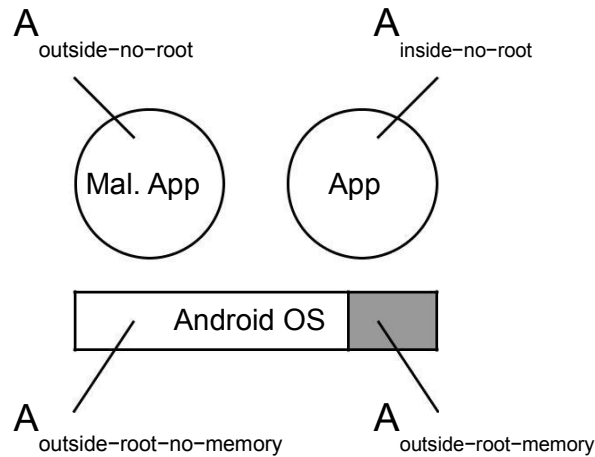


Figure 8: Schematic overview of the attacker models

All attacker models are shown in a schematic overview in the above figure. In this figure we see two applications, App is the genuine owner of the keys that need to be protected, Mal. App is an malicious application controlled by an attacker. The attackers that have root permissions can access the operating system controls and are therefore pictures to be inside the operating system. Although in fact attackers could also gain root credentials by using an application.

The solution that scores X on all requirements for all attacker models is clearly the best solution. Note that requirement $R_{app\ bound}$ is a more specific variant of requirement $R_{device\ bound}$. So if $R_{app\ bound}$ is satisfied then $R_{device\ bound}$ is also satisfied. Also note that the attacker models for $R_{app\ bound}$ and $R_{device\ bound}$ are increasing in the privileges the attacker has, $A_{outside\ no\ root}$ has the smallest number of permissions and $A_{outside\ root\ memory}$ the most. If a requirement can be violated by an attacker having attacker model $A_{outside\ no\ root}$ than it can also be violated by an attacker having attacker model $A_{outside\ root\ no\ memory}$ or $A_{outside\ root\ memory}$.

The case where a user is tricked to give for example his password is not considered an attack in this use case. However, this may be a realistic threat.

3.2.2 Evaluation

To test the Android KeyStore we use a modified version of Cooijmans KeyStorageTest application (9). It is modified so it will try and export the modulus of the RSA key and the Private exponent that we will be storing in Android Keystore which it is prohibited by the API for the software version²², and because there is a bug in Android code for versions 6 and 7 as described we cannot test its behavior²³. The code exists in [Appendix B](#), some classed are deprecated since Android 7, but they run correctly with backwards compatibility. So let's analyze what the code does.

²² <https://developer.android.com/training/articles/keystore.html>

²³ <https://issuetracker.google.com/issues/37091211>

On startup of the application the constructor checks if the cryptographic algorithms RSA, ECDSA and DSA are bound to the device if they are available. This check is done by function `isBoundKeyAlgorithm(String algorithm)` of the `KeyChain` class. When the algorithms are bound to device it returns `true`. Continuing the constructor gets all keys stored in the Keystore by alias. This alias is defined inside the application and creates the key with it which is for identification of the key pair. The other methods are for generating the key, signing it, delete it and also for the rogue app there is a special button that shows the modulus and try to show the private exponent.

This applications generates an RSA key pair using the code in table 3. This code generates a key pair using the RSA algorithm with a key with size of 2048bits. The `KeyPairGenerator` by default also generates a self-signed certificate. The subject the serial number, start and end dates and the validity period have to be defined on creation of the key pair. The `setKeySize` function is supported from API version 19 and above, while version 18 only supports RSA keys of 2048-bits in size.

Table 2: Generating an RSA key pair

```

KeyPairGenerator rsaKeyGen;

try {
    rsaKeyGen = KeyPairGenerator.getInstance("RSA",
"AndroidKeyStore");
} catch (Exception exception) {
    writeToLog(exception.toString());
    return;
}

KeyPairGeneratorSpec rsaKeyGenSpec;

try {
    rsaKeyGenSpec = new KeyPairGeneratorSpec.Builder(this)
        .setAlias(KEY_ALIAS)
        .setSubject(new X500Principal("CN=test"))
        .setSerialNumber(new BigInteger("1"))
        .setStartDate(new Date())
        .setEndDate(new GregorianCalendar(2019, 1,
1).getTime())
        .build();
} catch (Exception exception) {
    writeToLog(exception.toString());
    return;
}

try {
    rsaKeyGen.initialize(rsaKeyGenSpec);
} catch (InvalidAlgorithmParameterException exception) {
    writeToLog(exception.toString());
    return;
}

rsaKeyGen.generateKeyPair();

```

Since API version 21, Android KeyStore also supports symmetric keys generation which was the major change of it since its debut. We can see a short example of generating and retrieving a 256 AES key using the M API in Table 3.

This is Java Cryptography Architecture (JCA) code, very similar to our example for asymmetric key handling. What is new is that there are more parameters available for the developer to use when generating or importing a key. Now you can specify the exact usage type of each key (encryption/decryption, signing/verification), its block mode, padding, etc. These parameters are stored along with the key, and it is prohibited to use the key for other purpose than the one it was generated for. So you can use a key for encryption only for that reason. Key validity period for each purpose can also be specified.

Another major feature in this version it is that key usage needs authentication before actually using the key check Figure 9. The authentication validity period can also be set giving the key more protection even if the key use is authenticated. That way you can authenticate each time you use a key or once every 15 minutes. This is for both symmetric and asymmetric keys and as an extra, the system can check whether a user has authenticated within a given time period. This is a good way to verify user presence, especially if the application does not make use of cryptography.

Table 3: Symmetric Key generation²⁴

```
KeyGenParameterSpec.Builder builder = new
KeyGenParameterSpec.Builder("key1",
    KeyProperties.PURPOSE_ENCRYPT |
KeyProperties.PURPOSE_DECRYPT);
KeyGenParameterSpec keySpec = builder
    .setKeySize(256)
    .setBlockModes("CBC")
    .setEncryptionPaddings("PKCS7Padding")
    .setRandomizedEncryptionRequired(true)
    .setUserAuthenticationRequired(true)
    .setUserAuthenticationValidityDurationSeconds(5
* 60)
    .build();
KeyGenerator kg = KeyGenerator.getInstance("AES",
"AndroidKeyStore");
kg.init(keySpec);
SecretKey key = kg.generateKey();
KeyStore ks = KeyStore.getInstance("AndroidKeyStore");
ks.load(null);
KeyStore.SecretKeyEntry entry =
(KeyStore.SecretKeyEntry)ks.getEntry("key1", null);
key = entry.getSecretKey();
```

If we skip the details though key generation and storage work the same as previous android versions, the Keystore service provides an interface with almost the same methods and also a fallback implementation in opposing with the Keymaster which offers a completely different interface. All these have not mitigate the theme of this thesis though.

²⁴ <https://nelenkov.blogspot.gr/2015/06/keystore-redesign-in-android-m.html>

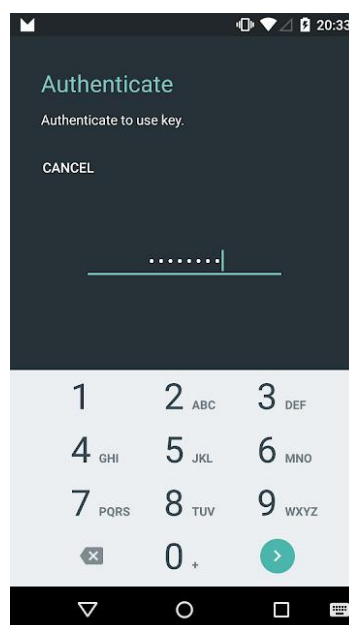


Figure 9: Per use key authentication

Keystore blobs are the files that keys are stored in the directory `/data/misc/keystore/user_X`, where X is the android user ID starting from 0 for primary user. Keystore blobs are variable size and they include a version byte, nonce, encrypted key, tag for authenticating the encrypted key and the key properties. This key blob can be encrypted for Android versions older than 6 as we will see in the next Chapter.

The Keystore data are stored on a per-application basis while the application is the one that gives the command to store and load the data. The application calls the load function of Keystore with an `InputStream` as argument which can be a `FileInputStream` that reads the contents of a file. As a second argument a byte array of the password can be provided as a password (as we will see later this will be the `.masterkey` of Android Keystore).

For our attack schema, two applications are installed on devices (`keystore_app`, `rogue_app`). The goal is to give the rogue app control over the key pair generated by the keystore app. Rogue app then can generate a valid signature over predefined data. If this is possible it violates $R_{app-bound}$. To test if $R_{device-bound}$ is violated we copy the key blobs and try to decrypt them or generate a valid signature on another phone. To verify $R_{user-consent}$ we look if the phone will ask for consent when we use the key (in Android 6 and later it asks but only if this API is used when creating the keys and providing the key purpose).

[Back to contents](#)

4. Experiment

In this Chapter we will experiment and evaluate if all assumptions that are real for Android 4.4 are also for 5, 6 & 7 versions. We will do the same tests in each Android version Keystore with or without TEE (software and hardware based) using the two applications keystore_app and rogue_app. In the beginning of each Section we will analyze each architecture and then we will provide the proof of concept for each one separately.

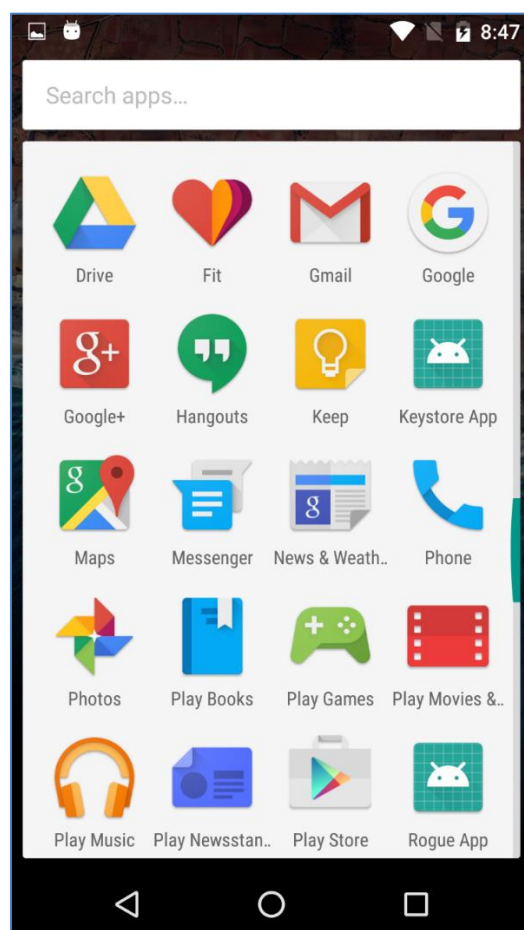


Figure 10: Legit app and Rogue app installed

Also for Android version 5 we will try to parse the private key of the keyblob exported from the device. To achieve that we will use the Nikolay Elenkov's java application keystore-decryptor²⁵ which decrypts the keystore files. To make it work I had to forcibly copy the bouncycastle library to the jar file of keystore-decryptor because the gradle script for an unknown reason did not put it inside the jar even when changing the version of bouncycastle to current.

²⁵ <https://github.com/nelenkov/keystore-decryptor>

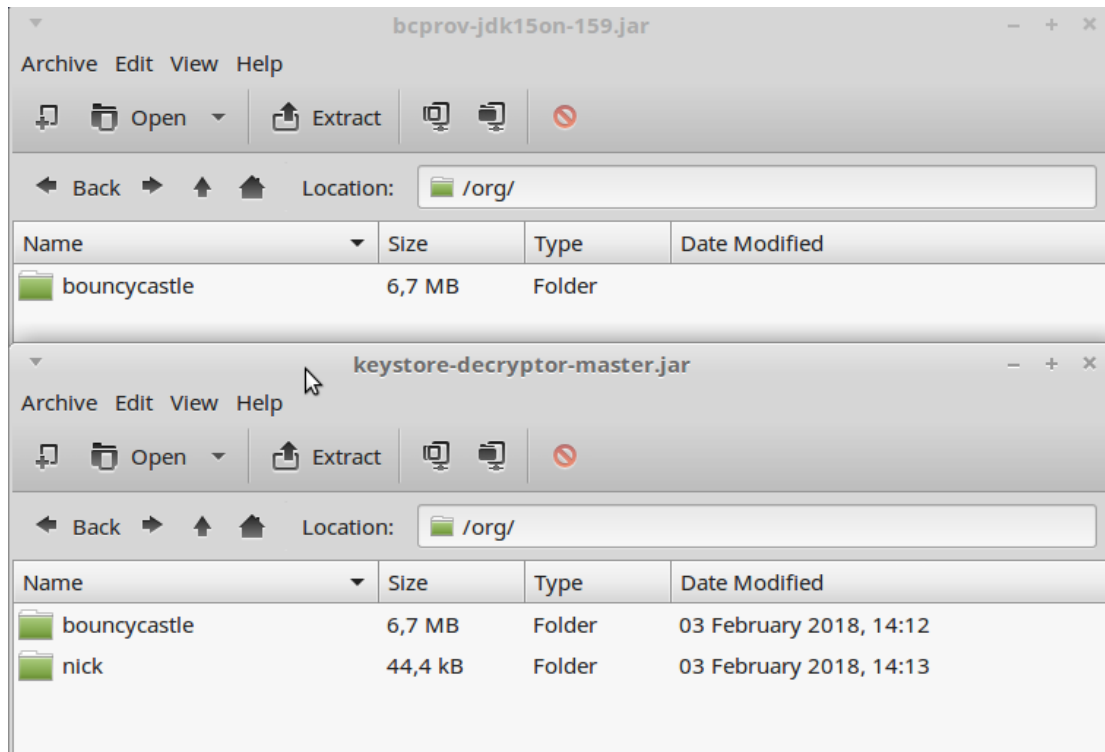


Figure 11: Forcibly inserting bouncycastle library to keystore-decryptor

The final step is to install each Android Version to Nexus 5 and then root it using the walkthroughs provided at the beginning of each Section.

4.1 Android KeyStore test using TEE on Qualcomm devices

Android Keystore on Qualcomm devices creates two key blobs in the directory we provided in [Section 3.2.2](#) which is `/data/misc/keystore/user_X`, two for each key pair created using the `keystore_app`.

- A USRPKEY key blob that stores the key pair parameters including the private key
- A USRCERT key blob that stores the self-signed certificate.

Both files have the following format upon creation given by the Android Keystore itself: `<UID of the app>_USRPKEY_<key alias given inside app>` and `<UID of the app>_USRCERT_<key alias given inside app>`. For example as we will see later our files will be like `10084_USRPKEY_TestKeyPair`. The UID of the app is the UID that the application is running under. The key alias is chosen by the developer using the `setAlias(String key_alias)` method of `KeyPairGeneratorSpec.Builder`.

The `/data/misc/keystore/user_X` is not accessible by a non-root user making other apps or services not to be able to read the key files. That way $R_{\text{device-bound}}$ and $R_{\text{app-bound}}$ are secured for $A_{\text{outside-no-root}}$. The part that make the Android Keystore vulnerable is the UID inside the name of the keyfiles. Using root permissions an attacker can rename or copy the files to new files in the same directory on the same device with the

UID of the rogue-app. For example in the next figure we show how we copy the key files of the legit up keystore_app to the rogue_app.

```

user@user-lp ~/Desktop
root@hammerhead:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 22:58 .masterkey
-rw----- keystore keystore     708 2018-02-13 22:37 10084_USRCERT_TestKeyPair
-rw----- keystore keystore     1652 2018-02-13 22:37 10084_USRPKEY_TestKeyPair
root@hammerhead:/data/misc/keystore/user_0 # cp 10084_USRPKEY_TestKeyPair 10
10084_USRCERT_TestKeyPair 10084_USRPKEY_TestKeyPair
p 10084_USRPKEY_TestKeyPair 10089_USRPKEY_TestKeyPair <
 10084_USRCERT_TestKeyPair 10089_USRCERT_TestKeyPair <
root@hammerhead:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 22:58 .masterkey
-rw----- keystore keystore     708 2018-02-13 22:37 10084_USRCERT_TestKeyPair
-rw----- keystore keystore     1652 2018-02-13 22:37 10084_USRPKEY_TestKeyPair
-rw----- root      root         708 2018-02-13 23:14 10089_USRCERT_TestKeyPair
-rw----- root      root         1652 2018-02-13 23:14 10089_USRPKEY_TestKeyPair
    
```

Figure 12: Copying key files with other UID

This way the private keys of legit app will be accessible to the other rogue app. Then we can generate a valid signature from the rogue_app using the key pair generated from the keystore_app. With this happening $R_{app-bound}$ is not satisfied for $A_{outside-root-no-memory}$. Copying the files to another device will not work because the private key can only be decrypted with the hardware-baked key that exists in the TEE. So $R_{device-bound}$ is satisfied for all attacker models. We observed that when no pin exists on the device there is no .masterkey file created giving us the suspicion that a standard key is used for keyblob encryption.

In Android 5 Android Keystore does not support user-key authentication when using a key as version 6 and 7 do. So an attacker can access the private key material at any time even for Android 6 and 7 if he knows the pin of the key. So $R_{user-consent}$ is not satisfied explicitly in Android 5 but only if the pin of the key is unknown is satisfied in Android 6 and 7.

For Android 5 this gives us the following conclusions:

- An attacker that has root permissions can easily use the keys of other apps on the same device by renaming the keystore files.
- Key pairs cannot be used outside of the device because the private data are encrypted with a device-specific key living inside the TEE that cannot be exported.
- There is no way to require any input from the user before key usage in Android 5.

The actual storage format of the PKEY is defined by the Qualcomm API 21. As we can see the file contains a number of fields according to the struct:

- Magic num: A fixed magic number is used to identify the les as key storage. The magic number is 0x4B4D4B42 or "KMKB" (KeyMaster Key Blob).
- Version num: The version number indicates the version of key.
- Modulus: Stores the modulus of the RSA key pair. The size of this field is fixed to KM KEY SIZE MAX which is 512 bytes (= 4096 bits).
- Modulus size: The actual size of the modulus. Since the modulus can be smaller than 4096 bits this field indicates the number of bytes that is actual used.

- Public exponent: Stores the public exponent of the RSA key pair. The size of this field is also fixed to KM KEY SIZE MAX.
- Public exponent size: The actual size of the public exponent.
- IV: The IV (initialization vector) that is used for the encryption of the encrypted private exponent using AES-CBC-128.
- Encrypted private exponent: Stores the private exponent of the RSA key pair in an encrypted form using AES-CBC-128 encryption. For the encryption the iv and an unknown key is used. This key is probably fixed in hardware to bind the private key to the device.
- Encrypted private exponent size: Stores the actual size of the private exponent in unencrypted form.
- HMAC: A HMAC is computed over the whole file using a SHA-2 and a key which is probably also fixed in hardware. This should ensure the integrity of the file.

The KeyStore service adds some data around the struct to identify the contents. The key used for encryption of the private exponent and for computing the HMAC cannot be found on the device. Also the trustlet is not open source according to the source code that is available the encryption is done in the TEE by the keymaster trustlet. We suspect that the keys are also stored in the TEE and are device specific. No method to access these keys from outside the TEE is documented, nor could we find one.

4.1.1 Evaluation in Android 5.0

Firstly we install the keystore app and we generate a key pair by pressing the “GEN” button and then we sign a byte array of string "Test Test Test". Then we install the roguw_app to our device and we read the file `/data/system/packages.list` which contains the information of which applications map with which UIDs, in order to find the roguw_app UID.

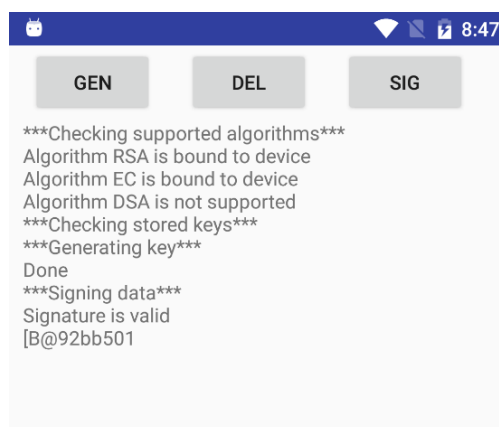
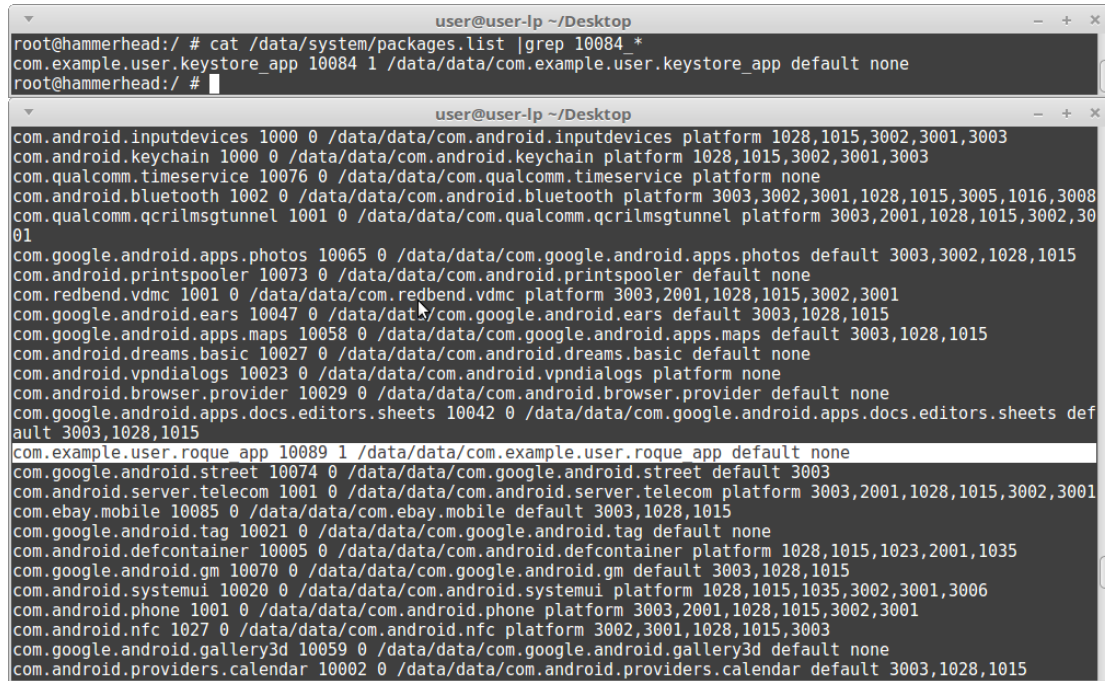


Figure 13: SHW key generation and validation of signature

Afterwards the only thing we have to do is to copy the files of the legit app with the UID of the rogue app (10089) as we can see in the figure below.

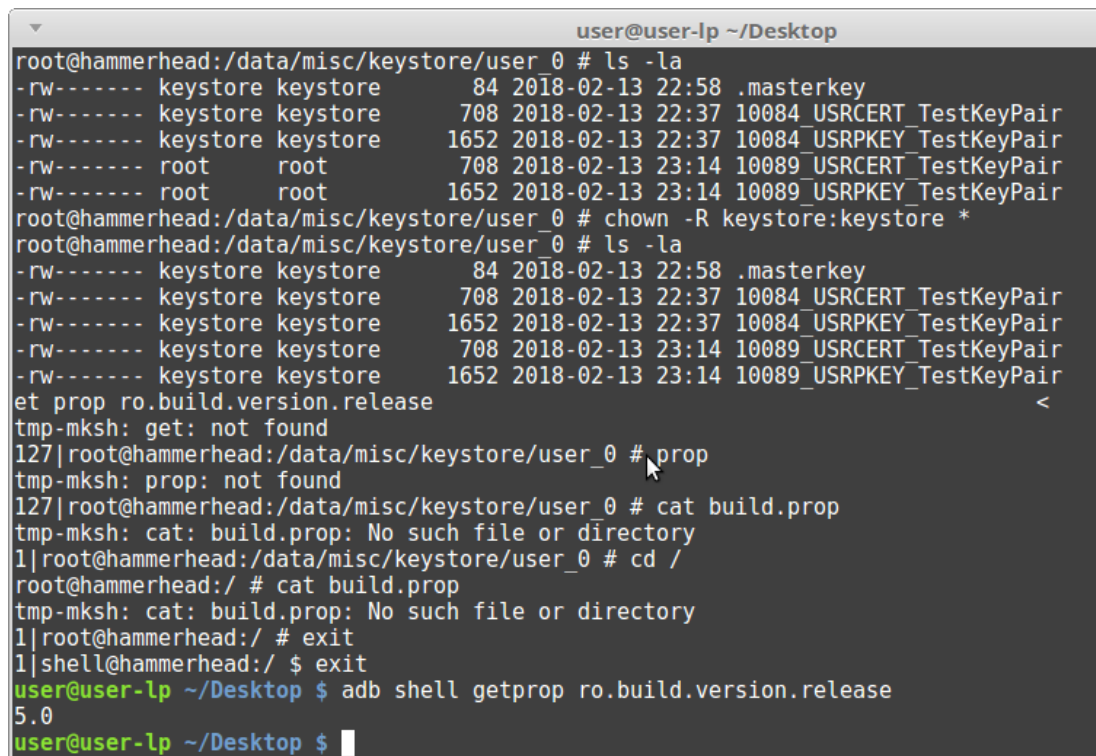
Security Evaluation of Android Keystore



```
user@user-lp ~/Desktop
root@hammerhead:/ # cat /data/system/packages.list | grep 10084 *
com.example.user.keystore_app 10084 1 /data/data/com.example.user.keystore_app default none
root@hammerhead:/ #

user@user-lp ~/Desktop
com.android.inputdevices 1000 0 /data/data/com.android.inputdevices platform 1028,1015,3002,3001,3003
com.android.keychain 1000 0 /data/data/com.android.keychain platform 1028,1015,3002,3001,3003
com.qualcomm.timeservice 10076 0 /data/data/com.qualcomm.timeservice platform none
com.android.bluetooth 1002 0 /data/data/com.android.bluetooth platform 3003,3002,3001,1028,1015,3005,1016,3008
com.qualcomm.qcrilmsgtunnel 1001 0 /data/data/com.qualcomm.qcrilmsgtunnel platform 3003,2001,1028,1015,3002,3001
com.google.android.apps.photos 10065 0 /data/data/com.google.android.apps.photos default 3003,3002,1028,1015
com.android.printspooler 10073 0 /data/data/com.android.printspooler default none
com.redbend.vdmc 1001 0 /data/data/com.redbend.vdmc platform 3003,2001,1028,1015,3002,3001
com.google.android.ears 10047 0 /data/data/com.google.android.ears default 3003,1028,1015
com.google.android.apps.maps 10058 0 /data/data/com.google.android.apps.maps default 3003,1028,1015
com.android.dreams.basic 10027 0 /data/data/com.android.dreams.basic default none
com.android.vpndialogs 10023 0 /data/data/com.android.vpndialogs platform none
com.android.browser.provider 10029 0 /data/data/com.android.browser.provider default none
com.google.android.apps.docs.editors.sheets 10042 0 /data/data/com.google.android.apps.docs.editors.sheets default 3003,1028,1015
com.example.user.roque_app 10089 1 /data/data/com.example.user.roque_app default none
com.google.android.street 10074 0 /data/data/com.google.android.street default 3003
com.android.server.telecom 1001 0 /data/data/com.android.server.telecom platform 3003,2001,1028,1015,3002,3001
com.ebay.mobile 10085 0 /data/data/com.ebay.mobile default 3003,1028,1015
com.google.android.tag 10021 0 /data/data/com.google.android.tag default none
com.android.defcontainer 10005 0 /data/data/com.android.defcontainer platform 1028,1015,1023,2001,1035
com.google.android.gm 10070 0 /data/data/com.google.android.gm default 3003,1028,1015
com.android.systemui 10020 0 /data/data/com.android.systemui platform 1028,1015,1035,3002,3001,3006
com.android.phone 1001 0 /data/data/com.android.phone platform 3003,2001,1028,1015,3002,3001
com.android.nfc 1027 0 /data/data/com.android.nfc platform 3002,3001,1028,1015,3003
com.google.android.gallery3d 10059 0 /data/data/com.google.android.gallery3d default none
com.android.providers.calendar 10002 0 /data/data/com.android.providers.calendar default 3003,1028,1015
```

Figure 14: SHW Finding UID of rogue and legit apps



```
user@user-lp ~/Desktop
root@hammerhead:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 22:58 .masterkey
-rw----- keystore keystore      708 2018-02-13 22:37 10084_USRCERT_TestKeyPair
-rw----- keystore keystore     1652 2018-02-13 22:37 10084_USRPKEY_TestKeyPair
-rw----- root root              708 2018-02-13 23:14 10089_USRCERT_TestKeyPair
-rw----- root root              1652 2018-02-13 23:14 10089_USRPKEY_TestKeyPair
root@hammerhead:/data/misc/keystore/user_0 # chown -R keystore:keystore *
root@hammerhead:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 22:58 .masterkey
-rw----- keystore keystore      708 2018-02-13 22:37 10084_USRCERT_TestKeyPair
-rw----- keystore keystore     1652 2018-02-13 22:37 10084_USRPKEY_TestKeyPair
-rw----- keystore keystore      708 2018-02-13 23:14 10089_USRCERT_TestKeyPair
-rw----- keystore keystore     1652 2018-02-13 23:14 10089_USRPKEY_TestKeyPair
et prop ro.build.version.release
tmp-mksh: get: not found
127|root@hammerhead:/data/misc/keystore/user_0 # prop
tmp-mksh: prop: not found
127|root@hammerhead:/data/misc/keystore/user_0 # cat build.prop
tmp-mksh: cat: build.prop: No such file or directory
1|root@hammerhead:/data/misc/keystore/user_0 # cd /
root@hammerhead:/ # cat build.prop
tmp-mksh: cat: build.prop: No such file or directory
1|root@hammerhead:/ # exit
1|shell@hammerhead:/ $ exit
user@user-lp ~/Desktop $ adb shell getprop ro.build.version.release
5.0
user@user-lp ~/Desktop $
```

Figure 15: SHW Copy key files and change ownership

Now we have to change also the ownership because the new files are owned by root user and group, or else keystore will not be able to read these files. After this step we run the rogue app and we can see that the key entry of the legit app is being read (Algorithm used, Subject, validity period) but also a validation of signature.

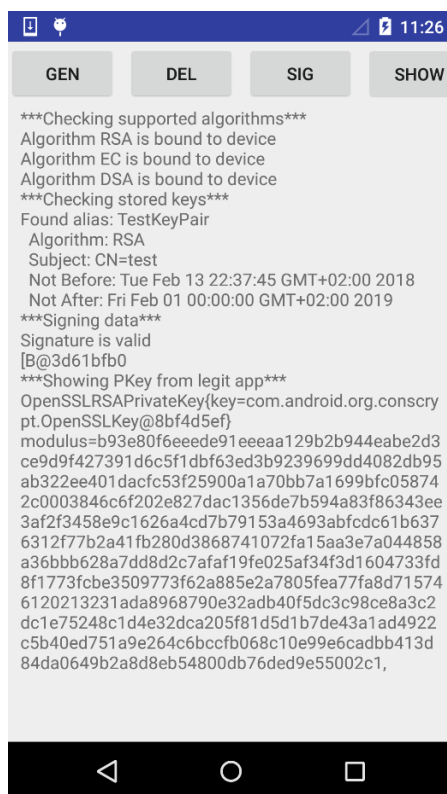


Figure 16: SHW rogue app parsing key of legit app

As we can see in Figure 16 the whole modulus of the RSA key can be exported on display. In the next step we will use keystore-decryptor to try to decrypt the private key from the file we pulled from device using ADB.

```
$ java -jar ksdecryptor-all.jar <master key file> <key
file> <password>
```

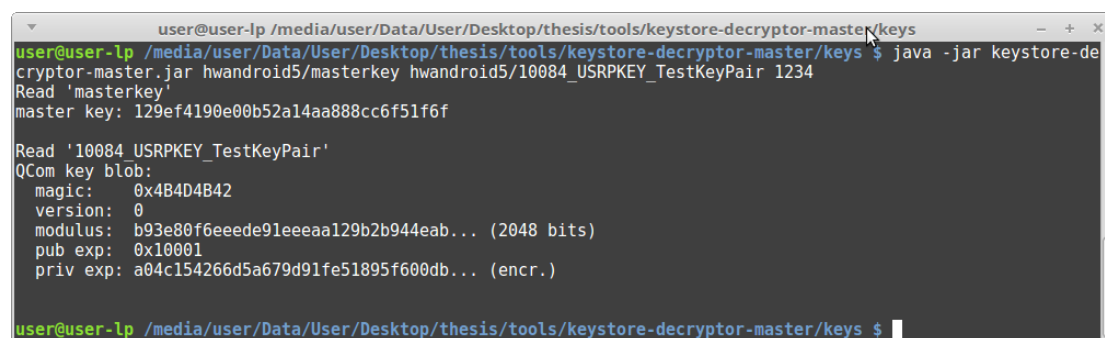


Figure 17: SHW using keystore-decryptor

The modulus that keystore-decryptor exports from the keyblob is the same as in Figure 16 but the private key cannot be exported because it is encrypted using the key provided by the TEE. The `.masterkey` file that encrypts the keyblob has its key decrypted. The structure of the keyblob is as described in page 30.

4.1.2 Evaluation in Android 6.0.1

Firstly we install the keystore app and when we run it we see that the algorithms are bound to device. Then we generate a key pair by pressing the “GEN” button and then

we push the “SIG” button to sign a byte array of string "Test Test Test" and verify the signature. Then we install the `rogue_app` to our device and we read the file `/data/system/packages.list` which contains the information of which applications map with which UIDs, in order to find the `rogue_app` UID.

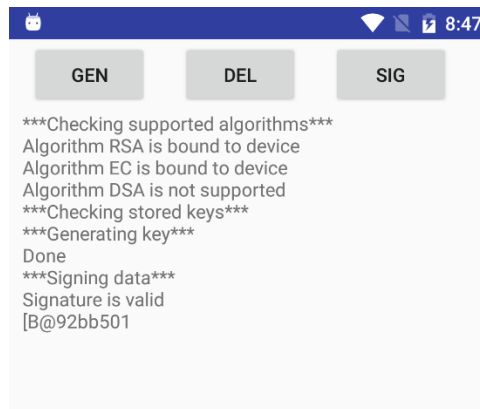


Figure 18: 6HW key generation and validation of signature

Afterwards the only thing we have to do is to copy the files of the legit app with the UID of the rogue app (10085) as we can see in the figure below. Now we have to change also the ownership because the new files are owned by root user and group, or else keystore will not be able to read these files.

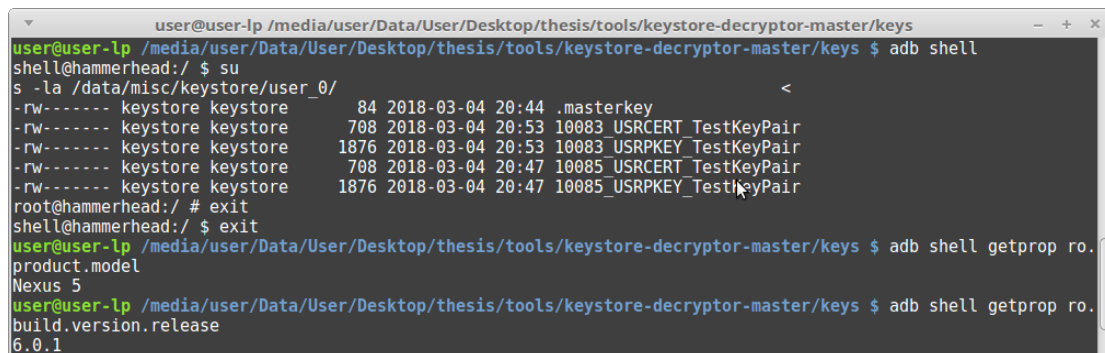


Figure 19: 6HW Copy key files and change ownership

The final step will be again to run the rogue app and check what will happen. We find out that nothing has changed and the key entries are accessible again by the rogue app. We can see again the algorithm, the subject and the validity period of the key pair and also the modulus of RSA but not whole. The system does not permit the full extraction of the modulus in the screen. Also when we try to export the private exponent of the key pair the bug which we referred in page 24 is causing a casting exception where an `RSAPrivateKey` cannot be casted to and `AndroidKeyStoreRSAPrivateKey` (funny huh?).

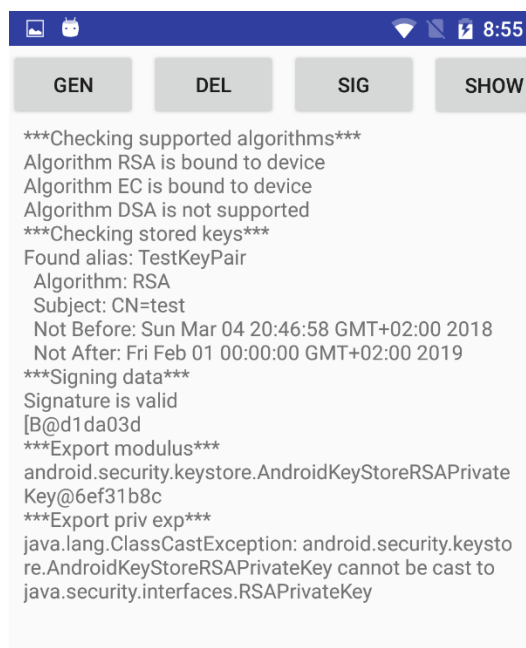


Figure 20: 6HW rogue app parsing key of legit app

4.1.3 Evaluation in Android 7.1.2

Again firstly we install the keystore app and when we run it we see that the algorithms are bound to device. Then we generate a key pair by pressing the “GEN” button and then we sign a byte array of string "Test Test Test" which is hardcoded in the device. Then we install the `rogue_app` to our device and we read the file `/data/system/packages.list` which contains the information of which applications map with which UIDs, in order to find the `rogue_app` UID.

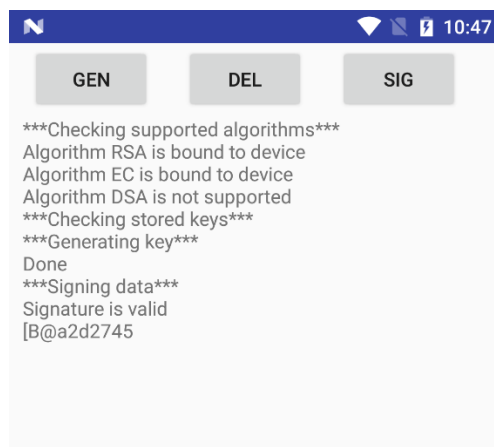


Figure 21: 7HW key generation and validation of signature

Afterwards we have to copy the files of the legit app with the UID of the rogue app (10085) as we can see in the figure below. Now we have to change also the ownership because the new files are owned by root user and group, or else keystore will not be able to read these files. In Android 7 we can see another entry which is hidden file `.10009_chr_USRPKEY_TestKeyPair`.

Security Evaluation of Android Keystore

```
user@user-lp ~
hammerhead:/ # cat data/system/packages.list | grep rogue app
com.example.user.rogue_app 10010 1 /data/user/0/com.example.user.rogue_app default none
hammerhead:/ # cd /data/misc/keystore/user_0/
hammerhead:/data/misc/keystore/user_0 # ls -la
total 48
drwx----- 2 keystore keystore 4096 2018-03-04 22:47 .
drwx----- 3 keystore keystore 4096 2018-03-04 22:47 ..
-rw----- 1 keystore keystore 228 2018-03-04 22:47 .10009_chr_USRPKEY_TestKeyPair
-rw----- 1 keystore keystore 84 2018-03-04 22:46 .masterkey
-rw----- 1 keystore keystore 708 2018-03-04 22:47 10009_USRCERT_TestKeyPair
-rw----- 1 keystore keystore 1892 2018-03-04 22:47 10009_USRPKEY_TestKeyPair
p 10009_USRPKEY_TestKeyPair 10010_USRPKEY_TestKeyPair <
p 10009_USRCERT_TestKeyPair 10010_USRCERT_TestKeyPair <
hammerhead:/data/misc/keystore/user_0 # chown keystore:keystore *
hammerhead:/data/misc/keystore/user_0 # ls -la
total 64
drwx----- 2 keystore keystore 4096 2018-03-04 23:03 .
drwx----- 3 keystore keystore 4096 2018-03-04 22:47 ..
-rw----- 1 keystore keystore 228 2018-03-04 22:47 .10009_chr_USRPKEY_TestKeyPair
-rw----- 1 keystore keystore 84 2018-03-04 22:46 .masterkey
-rw----- 1 keystore keystore 708 2018-03-04 22:47 10009_USRCERT_TestKeyPair
-rw----- 1 keystore keystore 1892 2018-03-04 22:47 10009_USRPKEY_TestKeyPair
-rw----- 1 keystore keystore 708 2018-03-04 23:03 10010_USRCERT_TestKeyPair
-rw----- 1 keystore keystore 1892 2018-03-04 23:03 10010_USRPKEY_TestKeyPair
hammerhead:/data/misc/keystore/user_0 # exit
hammerhead:/ # exit
user@user-lp ~ $ adb shell getprop ro.product.model
Nexus 5
user@user-lp ~ $ adb shell getprop ro.build.version.release
7.1.2
user@user-lp ~ $
```

Figure 22: 7HW Copy key files and change ownership

If we run the rogue app we can see again the algorithm, the subject and the validity period of the key pair and also the modulus of RSA but not whole. The system again does not permit the full extraction of the modulus in the screen. Also when we try to export the private exponent of the key pair the same exception with Android 6 is occurring.

```
GEN DEL SIG SHOW
***Checking supported algorithms***
Algorithm RSA is bound to device
Algorithm EC is bound to device
Algorithm DSA is not supported
***Checking stored keys***
Found alias: TestKeyPair
Algorithm: RSA
Subject: CN=test
Not Before: Sun Mar 04 22:47:33 GMT+02:00 2018
Not After: Fri Feb 01 00:00:00 GMT+02:00 2019
***Signing data***
Signature is valid
[B@d938654
***Export modulus***
android.security.keystore
.AndroidKeyStoreRSAPrivateKey@6f7055f3
***Export priv exp***
java.lang.ClassCastException: android.security.keysto
re.AndroidKeyStoreRSAPrivateKey cannot be cast to
java.security.interfaces.RSAPrivateKey
```

Figure 23: 7HW rogue app parsing key of legit app

There are not any particular differences from Android 6 and the result is typically the same.

4.2 Android KeyStore using software-based Keymaster

Again we will evaluate with the same methodology the Android KeyStore but now without using TEE and hardware-baked keys. The naming of the files when a software Keymaster, is the same as we have seen in Section 4.1 Android KeyStore test using TEE on Qualcomm devices. Also the attack schema is the same, if an attacker gains root permissions he can copy the keyblobs of the legit app to the keyblobs with the UID of the rogue app. This issue appears to be specific to the Android KeyStore and not to the actual implementation of the key storage that Android KeyStore uses. So $R_{\text{app-bound}}$ is not satisfied for $A_{\text{outside-root-no-memory}}$.

When a device does not require a PIN to unlock it no encryption is used for the private key. By parsing the USRPKEY file an attacker can learn all information that should be kept secret such as the private exponent and the two primes of the RSA key pair. However, when a PIN is required to unlock the device a random 128-bit AES master key is used for encryption. This master key is randomly generated and stored in the `.masterkey` file in the directory above. This file is encrypted using a key that is derived from the PIN using 8192 rounds of PKCS5 PBKDF2 HMAC SHA1. The master key is used to encrypt all key entries without any form of per-entry key-derivation. So if a password is used to unlock the device even if an attacker does not know the PIN or password, he can brute-force it outside of the device in case it has not much entropy or learn it from memory. Therefore we consider $R_{\text{device-bound}}$ as violated.

This changed in Android 6 whereas key blobs are wrapped inside keystore blobs, which are in turn stored as files in `/data/misc/keystore/user_X`, as before. Key material is encrypted using AES in OCB mode, which automatically authenticates the cipher text and produces an authentication tag upon completion. Each key blob is encrypted with a dedicated key encryption key (KEK), which is derived by hashing a binary tag representing the key's root of trust (hardware or software), concatenated with the key's authorization sets. Finally, the resulting hash value is encrypted with the master key to derive the blob's KEK. The current software implementation deliberately uses a 128-bit AES zero key, and employs a constant, all-zero nonce for all keys.

In this overview we see the following data (encoded as hexadecimal arrays):

1. The version number (=0)
2. The (public) modulus n (starts with E408)
3. The public exponent e (=010001)
4. The private exponent d (starts with BA4B)
5. The first prime p (starts with FD3E). This is one of the prime factors of the modulus.
6. The second prime q (starts with E683). This is the other prime factor of the modulus.
7. $d \bmod p-1$ (starts with 078C)
8. $d \bmod q-1$ (starts with C891)
9. $q-1 \bmod p$ (starts with CA12)

Items 2-4 represent all the data that is stored for an RSA key pair (including the private key). The additional data stored in parameters 5-9 are used for the Chinese Remainder Theorem (20). This allows for faster RSA operations compared to a native

implementation. Even when this data to use the Chinese Remainder Theorem is not present the private key can be used as described in the OpenSSL documentation for the RSA algorithm.

The software-based key store is called SoftKeyMaster or OpenSSLKeyMaster and can be found in the Android repositories²⁶. The communication between the application and the software-based is exactly the same as for the TEE-based implementation. The SoftKeyMaster acts as a kind of driver between the KeyMaster daemon and the filesystem.

For the needs of software based Android Keystore experiment we use the provided Android emulators AVDs. We firstly have to download Android Studio²⁷, download and install SDKs and images for Android versions 5, 6 & 7.

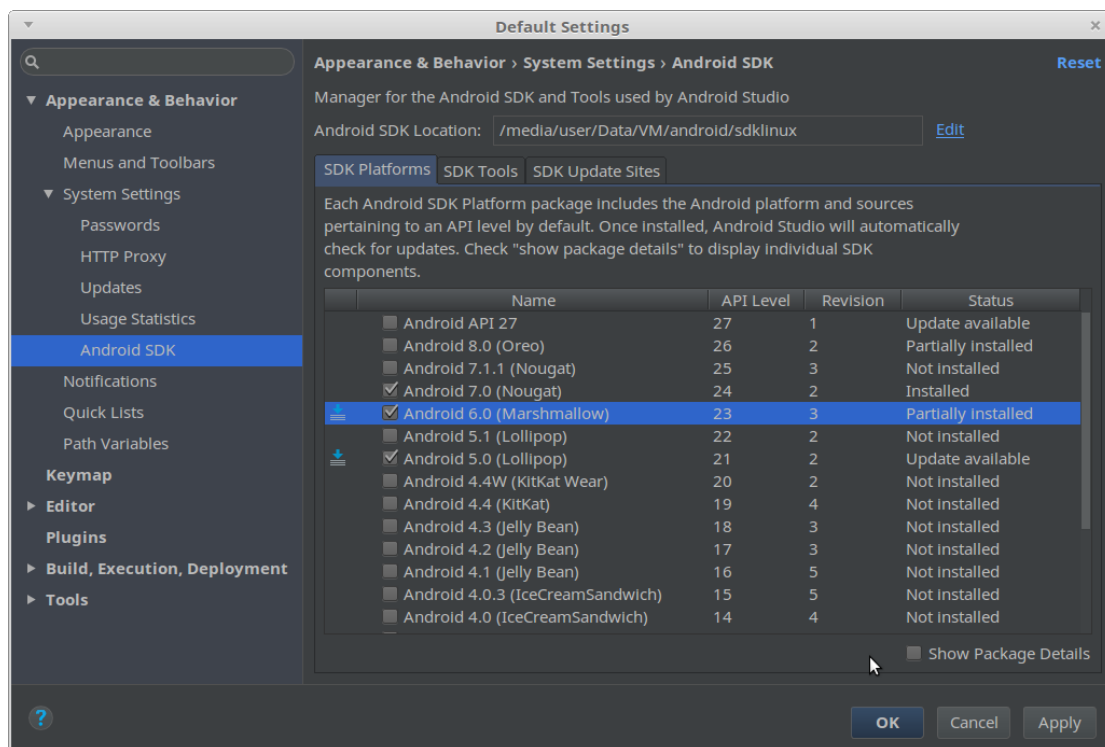
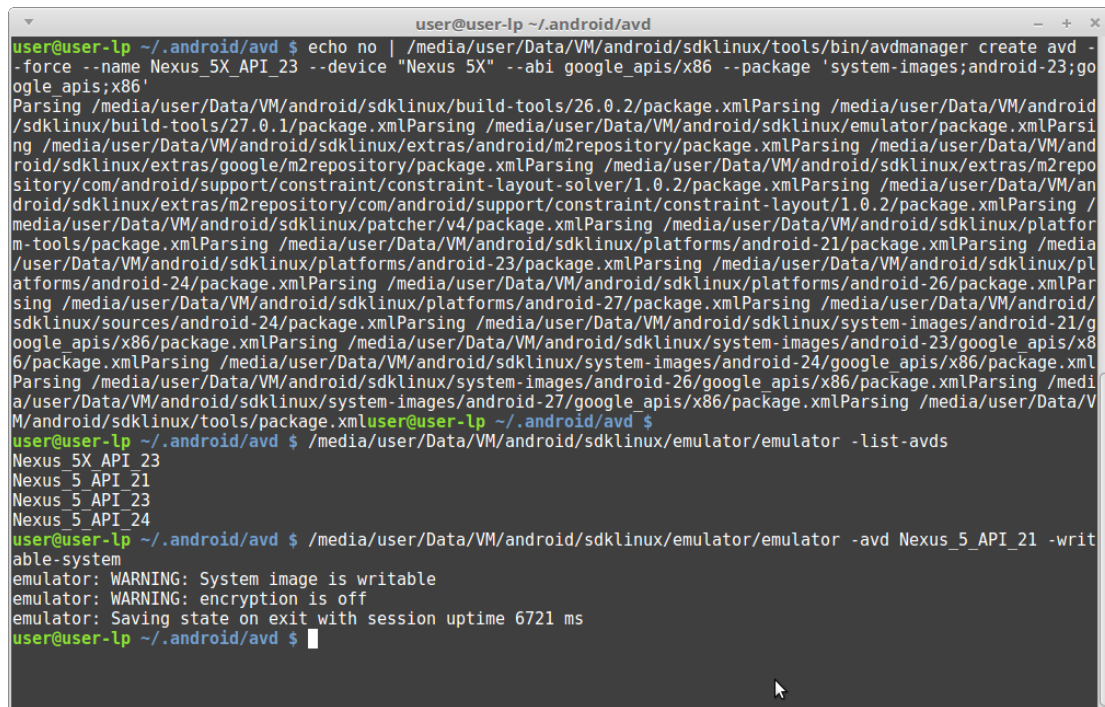


Figure 24: Android Studio downloading SDKs

When we have everything needed we can then create our AVDs using the avd manager provided by SDK.

²⁶ https://android.googlesource.com/platform/system/security/+/_master/softkeymaster/keymaster_openssl.cpp

²⁷ <https://dl.google.com/dl/android/studio/install/3.0.1.0/android-studio-ide-171.4443003-windows.exe>



```

user@user-lp ~/\.android/avd
user@user-lp ~/\.android/avd $ echo no | /media/user/Data/VM/android/sdklinux/tools/bin/avdmanager create avd -
-force --name Nexus_5X_API_23 --device "Nexus 5X" --abi google_apis/x86 --package 'system-images;android-23;go
ogle_apis;x86'
Parsing /media/user/Data/VM/android/sdklinux/build-tools/26.0.2/package.xmlParsing /media/user/Data/VM/android
/sdklinux/build-tools/27.0.1/package.xmlParsing /media/user/Data/VM/android/sdklinux/emulator/package.xmlParsi
ng /media/user/Data/VM/android/sdklinux/extras/android/m2repository/package.xmlParsing /media/user/Data/VM/and
roid/sdklinux/extras/google/m2repository/package.xmlParsing /media/user/Data/VM/android/sdklinux/extras/m2repo
sitory/com/android/support/constraint/constraint-layout-solver/1.0.2/package.xmlParsing /media/user/Data/VM/an
droid/sdklinux/extras/m2repository/com/android/support/constraint/constraint-layout/1.0.2/package.xmlParsing /
media/user/Data/VM/android/sdklinux/patcher/v4/package.xmlParsing /media/user/Data/VM/android/sdklinux/platfor
m-tools/package.xmlParsing /media/user/Data/VM/android/sdklinux/platforms/android-21/package.xmlParsing /media
/user/Data/VM/android/sdklinux/platforms/android-23/package.xmlParsing /media/user/Data/VM/android/sdklinux/pl
atforms/android-24/package.xmlParsing /media/user/Data/VM/android/sdklinux/platforms/android-26/package.xmlPar
sing /media/user/Data/VM/android/sdklinux/platforms/android-27/package.xmlParsing /media/user/Data/VM/android/
sdklinux/sources/android-24/package.xmlParsing /media/user/Data/VM/android/sdklinux/system-images/android-21/g
oogle_apis/x86/package.xmlParsing /media/user/Data/VM/android/sdklinux/system-images/android-23/google_apis/x8
6/package.xmlParsing /media/user/Data/VM/android/sdklinux/system-images/android-24/google_apis/x86/package.xml
Parsing /media/user/Data/VM/android/sdklinux/system-images/android-26/google_apis/x86/package.xmlParsing /medi
a/user/Data/VM/android/sdklinux/system-images/android-27/google_apis/x86/package.xmlParsing /media/user/Data/V
M/android/sdklinux/tools/package.xmluser@user-lp ~/\.android/avd $
user@user-lp ~/\.android/avd $ /media/user/Data/VM/android/sdklinux/emulator/emulator -list-avds
Nexus_5X_API_23
Nexus_5_API_21
Nexus_5_API_23
Nexus_5_API_24
user@user-lp ~/\.android/avd $ /media/user/Data/VM/android/sdklinux/emulator/emulator -avd Nexus_5_API_21 -writ
able-system
emulator: WARNING: System image is writable
emulator: WARNING: encryption is off
emulator: Saving state on exit with session uptime 6721 ms
user@user-lp ~/\.android/avd $

```

Figure 25: Creating the AVDs

For a statefull execution of the emulator we should append the parameter `--writable-system` when running the AVD. Next we can install google PlayStore in order to download and test apps for this vulnerability using open google apps project²⁸. We extract GmsCore.apk, GoogleServicesFramework.apk, GoogleLoginService.apk and Phonesky.apk from the archive and we push them in the AVD with ADB like below:

```
adb push PrebuiltGmsCore.apk /system/priv-app/ && adb push
GoogleServicesFramework.apk /system/priv-app/ && adb push
GoogleLoginService.apk /system/priv-app/ && adb push
Phonesky.apk /system/priv-app
```

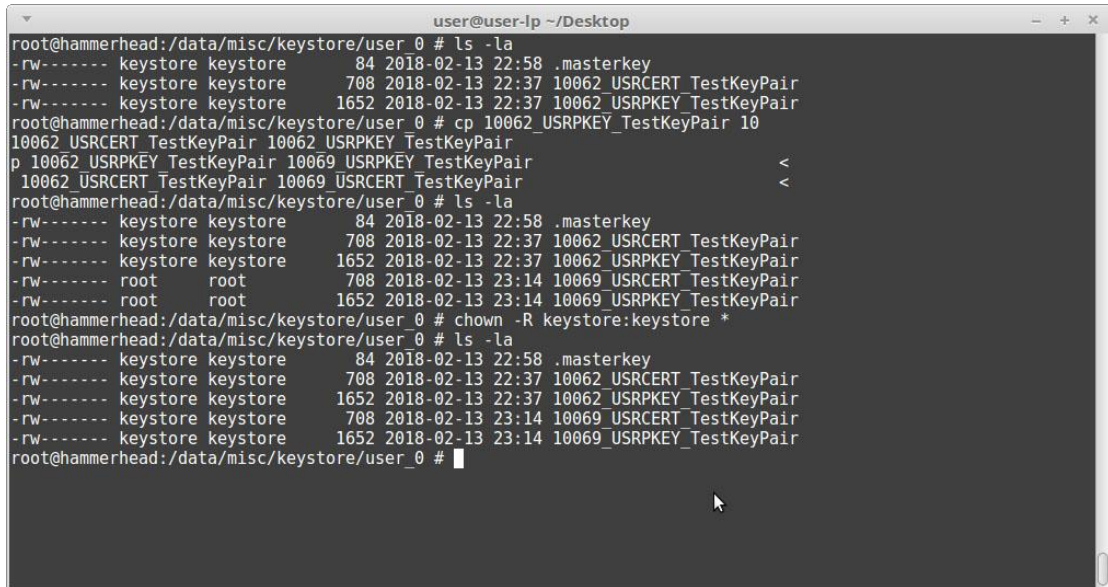
We are ready to go, we don't need to root the ADVs because they are unlocked and have root support from ADB natively.

4.2.1 Evaluation in Android 5.0

Firstly we install the keystore app and when we run it we see that the algorithms are **not** bound to device. Then we generate a key pair by pressing the "GEN" button and then we sign a byte array of string "Test Test Test" which is hardcoded in the device. Then we install the rogue_app to our device and we read the file `/data/system/packages.list` which contains the information of which applications map with which UIDs, in order to find the rogue_app UID.

²⁸ <http://opengapps.org/>

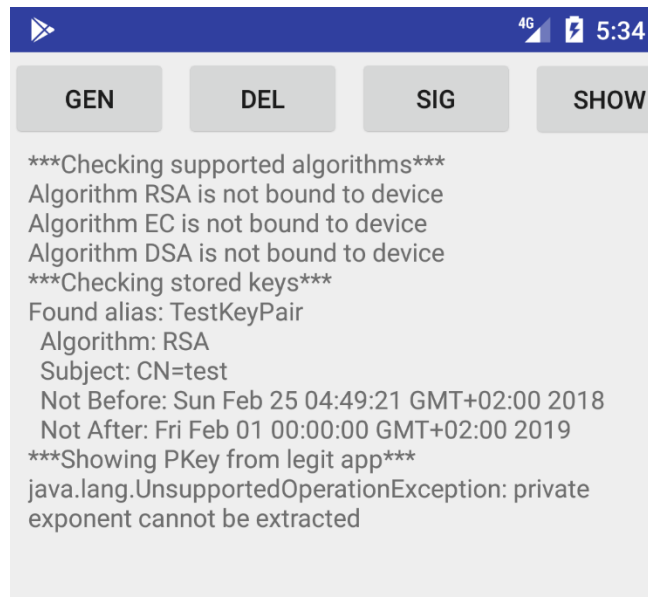
Security Evaluation of Android Keystore



```
user@user-lp ~/Desktop
root@hammerhead:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 22:58 .masterkey
-rw----- keystore keystore      708 2018-02-13 22:37 10062_USRCERT_TestKeyPair
-rw----- keystore keystore      1652 2018-02-13 22:37 10062_USRPKEY_TestKeyPair
root@hammerhead:/data/misc/keystore/user_0 # cp 10062_USRPKEY_TestKeyPair 10
10062_USRCERT_TestKeyPair 10062_USRPKEY_TestKeyPair
p 10062_USRPKEY_TestKeyPair 10069_USRPKEY_TestKeyPair <
10062_USRCERT_TestKeyPair 10069_USRCERT_TestKeyPair <
root@hammerhead:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 22:58 .masterkey
-rw----- keystore keystore      708 2018-02-13 22:37 10062_USRCERT_TestKeyPair
-rw----- keystore keystore      1652 2018-02-13 22:37 10062_USRPKEY_TestKeyPair
-rw----- root root              708 2018-02-13 23:14 10069_USRCERT_TestKeyPair
-rw----- root root              1652 2018-02-13 23:14 10069_USRPKEY_TestKeyPair
root@hammerhead:/data/misc/keystore/user_0 # chown -R keystore:keystore *
root@hammerhead:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 22:58 .masterkey
-rw----- keystore keystore      708 2018-02-13 22:37 10062_USRCERT_TestKeyPair
-rw----- keystore keystore      1652 2018-02-13 22:37 10062_USRPKEY_TestKeyPair
-rw----- keystore keystore      708 2018-02-13 23:14 10069_USRCERT_TestKeyPair
-rw----- keystore keystore      1652 2018-02-13 23:14 10069_USRPKEY_TestKeyPair
root@hammerhead:/data/misc/keystore/user_0 #
```

Figure 26: 5SW Copy key files and change ownership

Now we run the rogue app and we can see that the key entry of the legit app is being read (Algorithm used, Subject, validity period) but also a validation of signature.



```
GEN DEL SIG SHOW
***Checking supported algorithms***
Algorithm RSA is not bound to device
Algorithm EC is not bound to device
Algorithm DSA is not bound to device
***Checking stored keys***
Found alias: TestKeyPair
Algorithm: RSA
Subject: CN=test
Not Before: Sun Feb 25 04:49:21 GMT+02:00 2018
Not After: Fri Feb 01 00:00:00 GMT+02:00 2019
***Showing PKey from legit app***
java.lang.UnsupportedOperationException: private
exponent cannot be extracted
```

Figure 27: 5SW rogue app parsing key of legit app

As we can see in Figure 27: 5SW rogue app parsing key of legit app Android does not allow to extract the private exponent giving an `UnsupportedOperationException`. The next step is to try again to export the private key outside of the device. We use again the `keystore-decryptor` with the following command and let's see what happens.

```
$ java -jar ksdecryptor-all.jar <master key file> <key
file> <password>
```

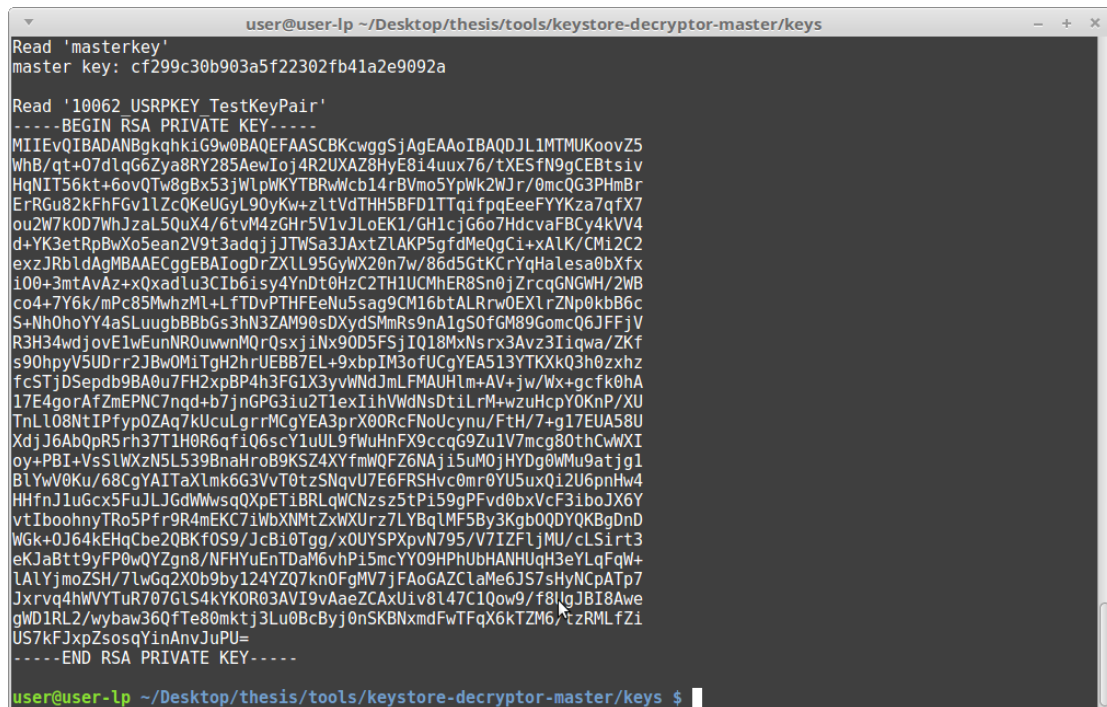


Figure 28: 5SW using keystore-decryptor

The Private Key entry has been fully exported in plain text!!!

4.2.2 Evaluation in Android 6.0

Again firstly we install the keystore app and then we generate a key pair by pressing the “GEN” button and then we sign a byte array of string "Test Test Test" which is hardcoded in the device. Then we install the rogue_app to our device and we read the file /data/system/packages.list which contains the information of which applications map with which UIDs, in order to find the rogue_app UID.

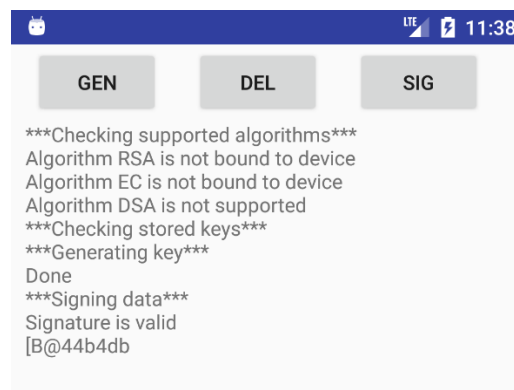
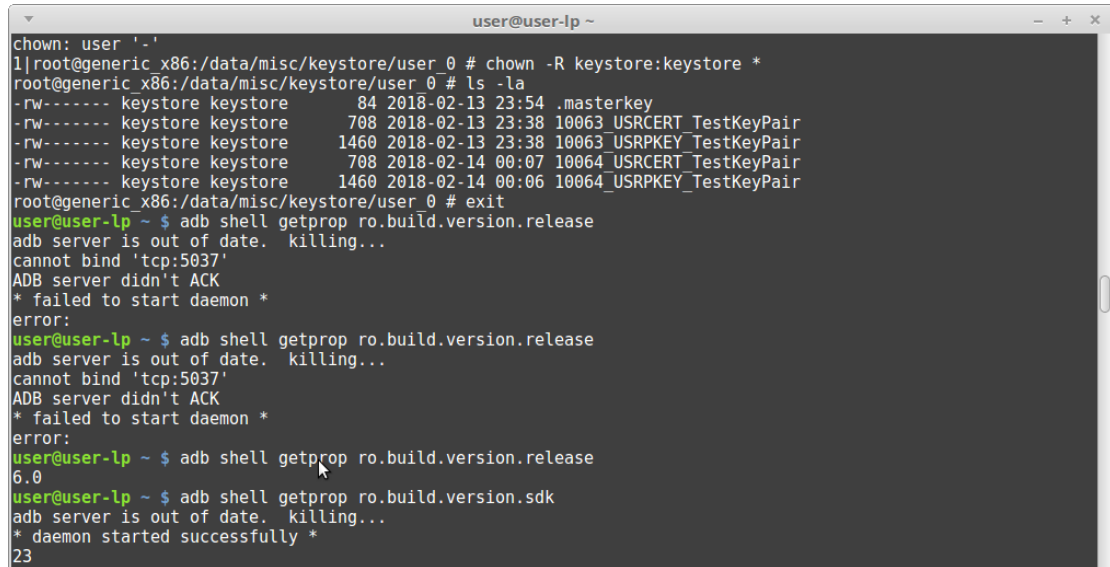


Figure 29: 6SW key generation and validation of signature

Afterwards we have to copy the files of the legit app with the UID of the rogue app (10064) as we can see in the figure below and change also the ownership.

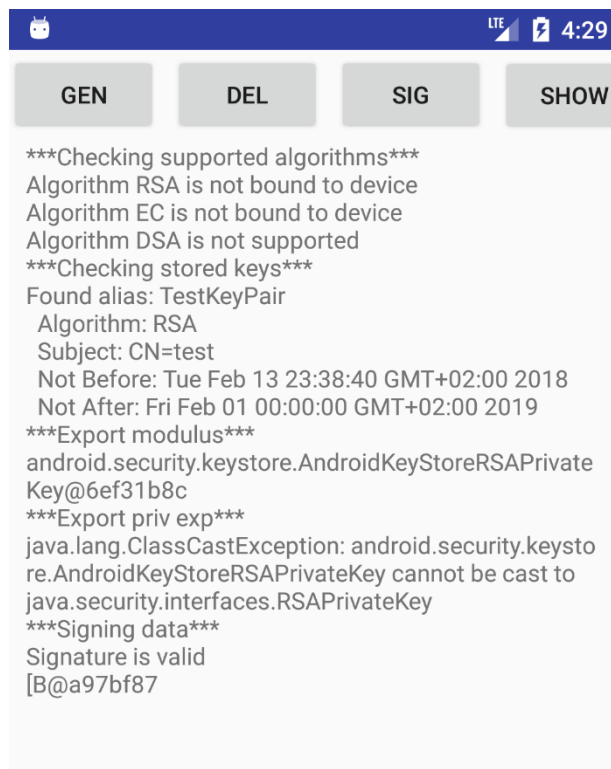
Security Evaluation of Android Keystore



```
user@user-lp ~
chown: user '-'
1|root@generic_x86:/data/misc/keystore/user_0 # chown -R keystore:keystore *
root@generic_x86:/data/misc/keystore/user_0 # ls -la
-rw----- keystore keystore      84 2018-02-13 23:54 .masterkey
-rw----- keystore keystore     708 2018-02-13 23:38 10063_USRCERT_TestKeyPair
-rw----- keystore keystore    1460 2018-02-13 23:38 10063_USRPKEY_TestKeyPair
-rw----- keystore keystore     708 2018-02-14 00:07 10064_USRCERT_TestKeyPair
-rw----- keystore keystore    1460 2018-02-14 00:06 10064_USRPKEY_TestKeyPair
root@generic_x86:/data/misc/keystore/user_0 # exit
user@user-lp ~ $ adb shell getprop ro.build.version.release
adb server is out of date. killing...
cannot bind 'tcp:5037'
ADB server didn't ACK
* failed to start daemon *
error:
user@user-lp ~ $ adb shell getprop ro.build.version.release
adb server is out of date. killing...
cannot bind 'tcp:5037'
ADB server didn't ACK
* failed to start daemon *
error:
user@user-lp ~ $ adb shell getprop ro.build.version.release
6.0
user@user-lp ~ $ adb shell getprop ro.build.version.sdk
adb server is out of date. killing...
* daemon started successfully *
23
```

Figure 30: 6SW Copy key files and change ownership

We then run the rogue app and we can see that the key entry of the legit app is being read (Algorithm used, Subject, validity period) but also a validation of signature.



```
***Checking supported algorithms***
Algorithm RSA is not bound to device
Algorithm EC is not bound to device
Algorithm DSA is not supported
***Checking stored keys***
Found alias: TestKeyPair
Algorithm: RSA
Subject: CN=test
Not Before: Tue Feb 13 23:38:40 GMT+02:00 2018
Not After: Fri Feb 01 00:00:00 GMT+02:00 2019
***Export modulus***
android.security.keystore.AndroidKeyStoreRSAPrivate
Key@6ef31b8c
***Export priv exp***
java.lang.ClassCastException: android.security.keysto
re.AndroidKeyStoreRSAPrivateKey cannot be cast to
java.security.interfaces.RSAPrivateKey
***Signing data***
Signature is valid
[B@a97bf87
```

Figure 31: 6SW rogue app parsing key of legit app

The same exception arises as in Android 6 hardware based keystore, also again modulus is not exported but the key of the legit app is parsed again without a problem. Software or hardware based it does not mitigate the problem for Android 6. Lastly let's check software based keystore of Android 7 if anything has changed with this version.

4.2.3 Evaluation in Android 7.0

We stuck to the procedure and we install the keystore app, then generate a key pair by pressing the “GEN” button and then we sign a byte array of string "Test Test Test" which is hardcoded in the device using the “SIG” button. Then we install the rogue_app to our device and we read the file /data/system/packages.list which contains the information of which applications map with which UIDs, in order to find the rogue_app UID.

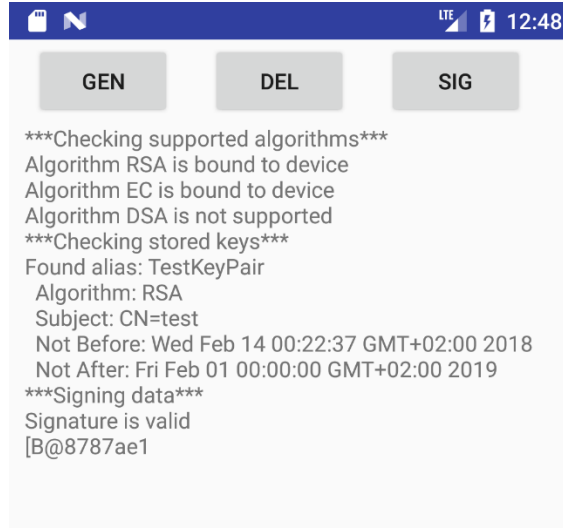


Figure 32: 7SW key generation and validation of signature

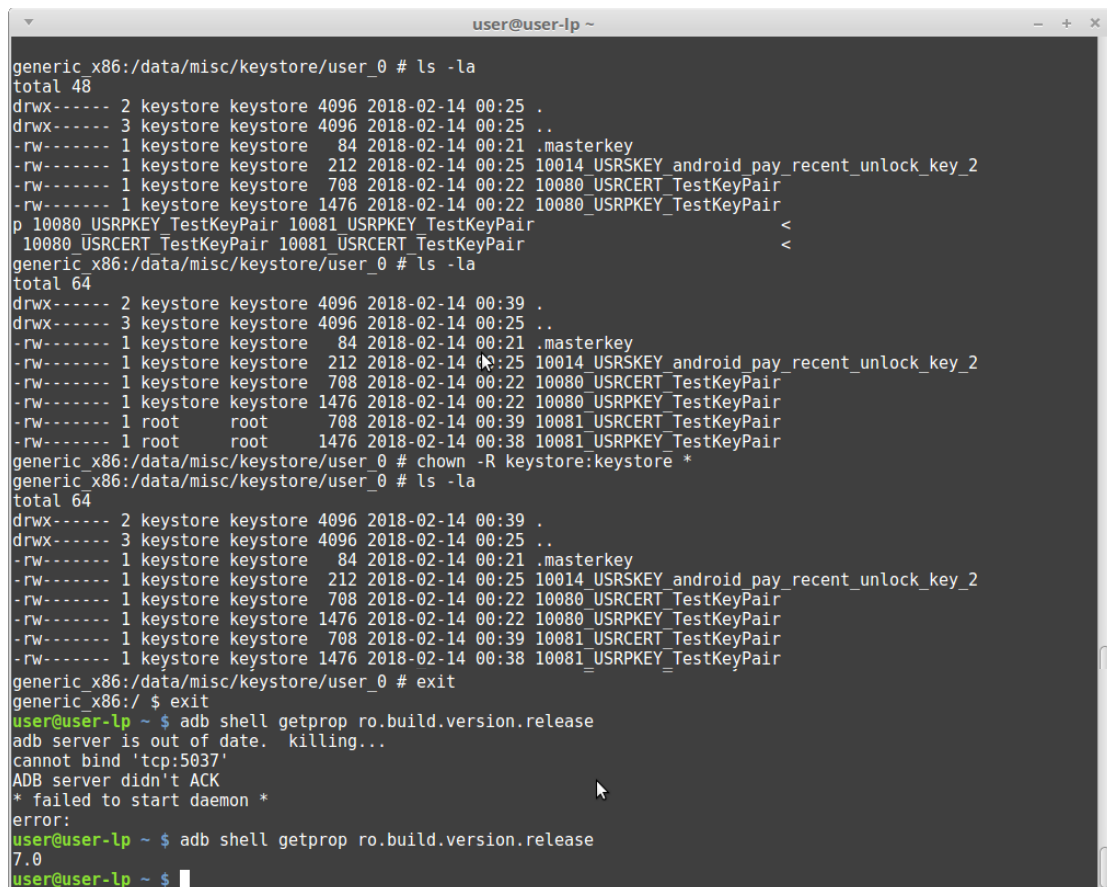


Figure 33: 7SW Copy key files and change ownership

Security Evaluation of Android Keystore

Afterwards we have to copy the files of the legit app with the UID of the rogue app (10081) as we can see in the figure above and change also the ownership. We run the rogue app and we can see that the key entry of the legit app is being read (Algorithm used, Subject, validity period) but also a validation of signature.

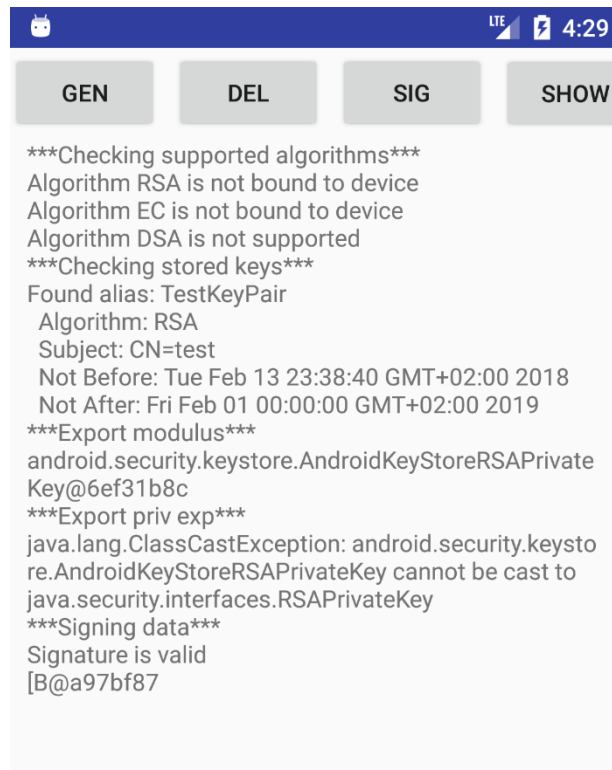


Figure 34: 6SW rogue app parsing key of legit app

The same exception arises as in Android 7 hardware based keystore, also again modulus is not exported but the key of the legit app is parsed again without a problem. Software or hardware based it does not mitigate the problem for Android 7.

[Back to contents](#)

5. Future Work & Conclusions

As time pass and Android OS is becoming more and more mature we still have to think about what level of security is provided to the end user. Users pay their money to buy a black box which will serve the needs they have to their everyday lives, not knowing most of the times the dangers but also what could happen if...

Even after 7 versions of API, Android developers still have not found a final solution to this major security hole giving the malicious app-user the ability to have unauthorized access to the secure stored key. In Android version 5 we also saw that a malicious user can even export the private key from the key blob of the software based Android KeyStore if he knows the pin and to make matters even worse, he can do it out of the device. In Android versions 6 and 7 if the keys are built with the new APIs then it asks for user consent when using the key but without fully protecting it again if the attacker know the password. Also when changing a PIN or password the device informs the user that it will delete the keys.

A solution to all these matters could be a custom kernel that will check everything that interacts with the Keystore directory, inform the user whenever this happens and provide choices of action. It will not mitigate the problem but at least the system will let the user know and decide what to do if he is not the one using the key entries.

The future work of this thesis is to research a way to obtain root access and use it to do all the things we saw automatically. Also to test the behavior of Android Keystore of versions 8 and 8.1.

[Back to contents](#)

References

1. **Android.** *Wikipedia.* [Online] [https://en.wikipedia.org/wiki/Android_\(operating_system\)](https://en.wikipedia.org/wiki/Android_(operating_system)).
2. **Linux.** *Wikipedia.* [Online] <http://en.wikipedia.org/wiki/Linux>.
3. **Computer Security.** *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Computer_security.
4. Haas, Peter D. *Ransomware goes mobile: An analysis of the threats posed by emerging methods.* May 2015.
5. Drake, Joshua, et al. *Android Hacker's Handbook.* March 2014.
6. **Mobile Security.** *Wikipedia.* [Online] https://en.wikipedia.org/wiki/Mobile_security.
7. *Mobile Security: Finally a Serious Problem?* Leavitt, Neal. 6, s.l. : IEEE, June 2011, Vol. 44. 0018-9162 .
8. Tim Cooijmans, Joeri de Ruiter, Erik Poll. Analysis of Secure Key Storage Solutions on Android. *Institute for Computing and Information Sciences (iCIS)* . [Online] 2014. Analysis of Secure Key Storage Solutions on Android.
9. Cooijmans, Tim. *Secure Key Storage and Secure Computation in Android.* s.l. : Radboud University Nijmegen, 2014.
10. Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone. Handbook of Applied Cryptography. *Centre for Applied Cryptographic Research (CACR).* [Online] October 1996. <http://cacr.uwaterloo.ca/hac/>. 0-8493-8523-7.
11. Symmetric cryptography. *IBM Knowledge Center.* [Online] https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.14/gtps7/s7symm.html.
12. National Bureau of Standards, US Department of Commerce. *Data Encryption Standard.* s.l. : Federal Information Processing Standards Publication 46, (1977).
13. Kumar, Sandeep. How to Break DES for €8,980. [Online] http://www.hyperelliptic.org/tanja/SHARCS/talks06/copa_sharcs.pdf.
14. Cryptographic Key Length Recommendation from organizations. *GRONAU IT CLOUD COMPUTING.* [Online] GRONAU IT CLOUD COMPUTING, November 21, 2016. <https://www.gronau-it-cloud-computing.de/en/cryptographic-key-length-recommendation-from-organizations/>.
15. Amit Vasudevan, Jonathan M. McCune, James Newsome. *Trustworthy Execution on Mobile Devices.* s.l. : Springer Science & Business Media, 2013. 9781461481904.
16. ARM. Building a Secure System using TrustZone Technology. [Online] April 2009. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. PRD29-GENC-009492C.
17. GENODE. An Exploration of ARM TrustZone Technology. [Online] <https://genode.org/documentation/articles/trustzone>.
18. *TrustZone Explained: Architectural Features and Use Cases.* Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho and Sarah Martin. s.l. : IEEE, 2016. 978-1-5090-4607-2/1.
19. *Trusted computing building blocks for embedded linux-based ARM trustzone platforms.* Winter, Johannes. 2008. 10.1145/1456455.1456460.
20. *Chinese remainder theorem: applications in computing, coding, cryptography.* C Ding, D Pei, and A Salomaa. 1996.

Appendix A - Acronyms

A	ADB	Android Debug Bridge
	AOSP	Android Open Source Project
	AVD	Android Virtual Device
C	CA	Certificate Authority
	CKMS	Cryptographic Key Management System
	CPU	Central Processing Unit
D	DSA	Digital Signing Algorithm
E	ECC	Elliptic Curve Cryptography
	ECDSA	Elliptic Curve Digital Signing Algorithm
G	GUI	Graphical User Interface
H	HMAC	Hashed Message Authentication Code
I	IDE	Integrated Development Environment
K	KMS	Key Management System
L	LTS	Long Term Support
M	MAC	Message Authentication Code
	MMS	Multimedia Messaging Service
O	OEM	Original Equipment Manufacturer
	OS	Operating System
R	RSA	Rivest Shamir Adleman
S	SCR	Secure Configuration Register
	SDK	Software Development Kit
	SGID	Set Group ID
	SKS	Secure Key Storage
	SMC	Secure Monitor Call
	SMS	Short Message Service
	SSL	Secure Socket Layer
	SUID	Set User ID
T	TAN	Transaction Authentication Number
	TEE	Trusted Execution Environment
	TLS	Transport Layer Security
	TPM	Trusted Platform Module
U	UID	User ID

Appendix B – Source Code

MainActivity.java

```
1
2 package com.example.user.roque_app;
3
4 import android.app.Activity;
5 import android.os.Build;
6 import android.os.Bundle;
7 import android.security.KeyChain;
8 import android.security.KeyPairGeneratorSpec;
9 import android.util.Log;
10 import android.text.method.ScrollingMovementMethod;
11 import android.view.View;
12 import android.widget.TextView;
13 import android.support.v7.app.AppCompatActivity;
14
15 import javax.security.auth.x500.X500Principal;
16
17 import java.math.BigInteger;
18 import java.security.*;
19 import java.security.cert.*;
20 import java.security.interfaces.RSAPrivateKey;
21 import java.util.*;
22
23 import static com.example.user.roque_app.R.id.consoleTextView;
24
25
26 public class MainActivity extends Activity {
27     //Tag used for identifying this Activity in the logs
28     private final static String TAG = "KeyStorageTest";
29     //Asymmetric algorithm to use for testing
30     private final static String KEY_ALGORITHM = "RSA";
```


Security Evaluation of Android Keystore

```
31     //Signature algorithm to use
32     private final static String SIGNATURE_ALGORITHM = "SHA512WithRSA";
33     //Alias of the key to be used
34     private final static String KEY_ALIAS = "TestKeyPair";
35     //Define the algorithms to check if they are supported by the app
36     private final static ArrayList<String> algorithmsToCheck = new ArrayList<String>() {{
37         add("RSA");
38         add("EC");
39         add("DSA");
40     }};
41     //The reference to the view in the activity that shows the log
42     private TextView logView;
43     //Reference to the AndroidKeyStore
44     private KeyStore androidKeyStore;
45
46     /**
47      * Called when the activity class is constructed.
48      */
49
50     @Override
51     public void onCreate(Bundle savedInstanceState) {
52         super.onCreate(savedInstanceState);
53         setContentView(R.layout.main);
54         logView = findViewById(consoleTextView);
55         logView.setMovementMethod(new ScrollingMovementMethod());
56         try {
57             androidKeyStore = KeyStore.getInstance("AndroidKeyStore");
58             androidKeyStore.load(null);
59         } catch (Exception exception) {
60             writeToLog(exception.toString());
61         }
62
63         doStartUpChecks();
64     }
65
66     /**
```

```
67     * Do a number of checks when the Activity is launched.
68     * <p>
69     * Bibliography
70     */
71 private void doStartupChecks() {
72     writeToLog("***Checking supported algorithms***");
73     for (String algorithm : algorithmsToCheck) {
74         if (KeyChain.isKeyAlgorithmSupported(algorithm)) {
75             if (KeyChain.isBoundKeyAlgorithm(algorithm)) {
76                 writeToLog("Algorithm " + algorithm + " is bound to device");
77             } else {
78                 writeToLog("Algorithm " + algorithm + " is not bound to device");
79             }
80         } else {
81             writeToLog("Algorithm " + algorithm + " is not supported");
82         }
83     }
84
85
86     writeToLog("***Checking stored keys***");
87
88
89     try {
90         Enumeration<String> keyStoreAliases = androidKeyStore.aliases();
91         while (keyStoreAliases.hasMoreElements()) {
92             String keyStoreAlias = keyStoreAliases.nextElement();
93             writeToLog("Found alias: " + keyStoreAlias);
94             try {
95                 X509Certificate certificate = (X509Certificate)
androidKeyStore.getCertificate(keyStoreAlias);
96                 PrivateKey privateKey = (PrivateKey) androidKeyStore.getKey(keyStoreAlias, null);
97                 writeToLog("\tAlgorithm: " + privateKey.getAlgorithm());
98                 writeToLog("\tSubject: " + certificate.getSubjectDN().toString());
99                 writeToLog("\tNot Before: " + certificate.getNotBefore().toString());
100                writeToLog("\tNot After: " + certificate.getNotAfter().toString());
101            }
```

Security Evaluation of Android Keystore

```
102         } catch (Exception exception) {
103             writeToLog(exception.toString());
104         }
105     }
106     } catch (Exception exception) {
107         writeToLog(exception.toString());
108     }
109 }
110
111 /**
112  * Write a message to the log.Writes to both the the view and
113  * the log cat log
114  *
115  * @parammessage
116  */
117 private void writeToLog(String message) {
118     logView.append(message + "\n");
119     Log.d(TAG, message);
120 }
121
122
123 /**
124  * Called when the "GenerateKey" button is clicked.
125  *
126  * @paramview
127  */
128 public void onGenerateKeyButtonClick(View view) {
129     KeyPairGenerator rsaKeyGen;
130     writeToLog("***Generating key***");
131
132     try {
133         rsaKeyGen = KeyPairGenerator.getInstance(KEY_ALGORITHM, "AndroidKeyStore");
134     } catch (Exception exception) {
135         writeToLog(exception.toString());
136         return;
137     }
```

```
138
139     KeyPairGeneratorSpec rsaKeyGenSpec;
140     try {
141         if (Build.VERSION.RELEASE.startsWith("4.4")) {
142             rsaKeyGenSpec = new KeyPairGeneratorSpec.Builder(this)
143                 .setAlias(KEY_ALIAS)
144                 .setSubject(new X500Principal("CN=test"))
145                 .setSerialNumber(new BigInteger("1"))
146                 .setStartDate(new Date())
147                 .setEndDate(new GregorianCalendar(2019, 1, 1).getTime())
148                 .setKeySize(2048)
149                 .build();
150         } else {
151             rsaKeyGenSpec = new KeyPairGeneratorSpec.Builder(this)
152                 .setAlias(KEY_ALIAS)
153                 .setSubject(new X500Principal("CN=test"))
154                 .setSerialNumber(new BigInteger("1"))
155                 .setStartDate(new Date())
156                 .setEndDate(new GregorianCalendar(2019, 1, 1).getTime())
157                 .build();
158         }
159     } catch (Exception exception) {
160         writeToLog(exception.toString());
161         return;
162     }
163
164     try {
165         rsaKeyGen.initialize(rsaKeyGenSpec);
166     } catch (InvalidAlgorithmParameterException exception) {
167         writeToLog(exception.toString());
168         return;
169     }
170     rsaKeyGen.generateKeyPair();
171     writeToLog("Done");
172 }
173 /**
```

Security Evaluation of Android Keystore

```
174     * Called when the "Delete" button is clicked.
175     */
176     public void onDeleteKeyButtonClick(View view) {
177         try {
178             androidKeyStore.deleteEntry(KEY_ALIAS);
179         } catch (KeyStoreException exception) {
180             writeToLog(exception.toString());
181         }
182     }
183     /**
184     * Called when the "SignData" button is clicked.
185     *
186     * @paramview
187     */
188     public void onSignDataButtonClick(View view) {
189         writeToLog("***Signing data***");
190         try {
191             Signature testKeySignature = Signature.getInstance(SIGNATURE_ALGORITHM);
192             PrivateKey testPrivateKey = (PrivateKey) androidKeyStore.getKey(KEY_ALIAS, null);
193             testKeySignature.initSign(testPrivateKey);
194             testKeySignature.update("Test Test Test".getBytes());
195             byte[] signature = testKeySignature.sign();
196
197             Signature testKeySignatureVerf = Signature.getInstance(SIGNATURE_ALGORITHM);
198             testKeySignatureVerf.initVerify(androidKeyStore.getCertificate(KEY_ALIAS));
199             testKeySignatureVerf.update("Test Test Test".getBytes());
200             if (testKeySignatureVerf.verify(signature)) {
201                 writeToLog("Signature is valid");
202                 writeToLog(signature.toString());
203             } else {
204                 writeToLog("Invalid signature");
205             }
206
207         } catch (Exception exception) {
208             writeToLog(exception.toString());
209         }
```

```
210     }
211 }
212
213 public void showPKey(View view){
214     try {
215
216         writeToLog("***Export modulus***");
217         PrivateKey testPrivateKey = (PrivateKey) androidKeyStore.getKey(KEY_ALIAS, null);
218         writeToLog(testPrivateKey.toString());
219         writeToLog("***Export priv exp***");
220         androidKeyStore.load(null);
221         KeyStore.PrivateKeyEntry keyEntry =
222 (KeyStore.PrivateKeyEntry) androidKeyStore.getEntry(KEY_ALIAS, null);
223         RSAPrivateKey privKey = (RSAPrivateKey) keyEntry.getPrivateKey();
224         writeToLog(privKey.getPrivateExponent().toString());
225     } catch ( Exception ex){
226         writeToLog(ex.toString());
227     }
228 }
229 }
```