OULUN AMMATTIKORKEAKOULU

Samuli Rissanen

COMMON CRITERIA SECURITY AUDIT FOR MOBILE DEVICES

COMMON CRITERIA SECURITY AUDIT FOR MOBILE DEVICES

Samuli Rissanen
Bachelor's Thesis
Fall 2018
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology

Author: Samuli Rissanen
Title of thesis: Common Criteria Security Audit for Mobile Devices
Supervisor: Juha Mäki-Asiala, Gabriel Byman, Teemu Korpela
Term and year of completion: Fall 2018 Pages: 70

The aim of this bachelor's thesis was to study the requirements set for security auditing of mobile devices by Common Criteria, to determine how the current system fulfills these requirements and to make the needed modifications to fulfill the requirements. This thesis was commissioned by Bittium Oyj.

The requirements set for the security auditing for the mobile devices were studied and a list of all the requirements that needed to be fulfilled were composed. After the requirements were clear, the actual logging system of Android was studied to understand how it has been implemented. The logging system was studied very thoroughly to get the complete picture on how it functions at different levels of the system and how it is currently being utilized in the Android platform.

After the Android logging system has been presented, a few solutions on how audit logging functionality could be implemented and integrated as a part of the Android logging system are presented.

The actual analysis of the system was the last part of this thesis. Analyzing certain components of the system took a significant amount of time as they were very complex, but understanding their operation and functionality was crucial for making a proper analysis. The analysis itself was a surprisingly demanding work, as I had to cover a lot of individual components and had to learn how they function in a very short time.

As was found during the analysis, some requirements were not initially met, but after making the necessary modifications to the responsible components, most of the requirements were met in the end. Due to time restrictions set at the beginning of this thesis, some of the requirements were identified to require a great effort to make the necessary modifications and therefore, they were left out of the scope of the implementation part of this thesis.

Keywords: Common Criteria, Mobile Devices, Android, Security, Audit Logging

PREFACE

This thesis was carried out during the summer holidays of 2018 for Bittium Oyj.

First, I would like to thank Bittium for hiring me as an intern prior to the making this thesis and for providing the possibility to make this thesis.

Second, I would like to thank all of my colleagues at Bittium for providing a wonderful and friendly working environment.

Special thanks go to Juha Mäki-Asiala and Gabriel Byman as my supervisors here at Bittium and to Teemu Korpela as my supervisor from OUAS for providing guidance during the whole process.

Oulu, 11.8.2018
Samuli Rissanen

CONTENTS

VOCABULARY

CCRA	Common Criteria Recognition Agreement
API	Application Programming Interface
ART	Android Runtime
DVM	Dalvik Virtual Machine
SELinux	Security-Enhanced Linux
ADB	Android Debug Bridge
MDM	Mobile Device Management
HAL	Hardware Abstraction Layer
DEK	Data Encryption Key
KEK	Key Encryption Key

1 INTRODUCTION

The objective for this thesis was planned together with Bittium Oyj.

Bittium Oyj was branched out of Elektrobit Oyj in 2015, after the company sold its automotive related business activities and the rights to the Elektrobit brand name. Elektrobit Oyj was originally founded in 1985. (1.)

Bittium employed a little over 600 people in 5 countries at the end of 2017. The majority of the employees is spread across 5 offices located in Finland. (1.)

Bittium specializes in developing reliable and secure communication solutions and equipment. In addition, Bittium provides medical technology solutions for bio signal measuring and high-quality information security solutions for mobile devices and computers. (1.)

Bittium additionally provides consulting and R&D services for external parties. (1.)

The primary aim that was set for this thesis was to familiarize with the Common Criteria security audit requirements for mobile devices, to analyze the current Android system and its compatibility with the requirements. Then if necessary, to implement the needed changes to comply with the Common Criteria security requirements.

The system that was analyzed is an undisclosed mobile device using the version 8.1 of Android. Further details about the system will not be presented in this thesis due to the proprietary and secret nature of the information. This is also the reason why very little inner details are revealed from proprietary components and certain implementation details.

If implementation modifications were necessary to fulfill the requirements, the implementations that were made were small scale modifications. As the time was very limited and it was necessary to learn the system, bigger scale modifications or complete overhauls of the components were not made.

The Common Criteria organization is introduced at the beginning of the document along with Protection Profiles, which are created and maintained by the Common Criteria organization.

After the introduction of the general format of the Protection Profile, a deeper look is taken into the Mobile Device Fundamentals Protection Profile, which is the applicable Protection Profile for this thesis.

The Mobile Device Fundamentals Protection Profile is analyzed and the applicable security requirements are composed together into this document to help the reader get a better understanding of the overall requirements and their meaning.

The Android logging system is presented in great detail as understanding the inner workings of the logging system and how the logging system is utilized at different levels was crucial for analyzing the system and its logging capabilities.

The Android logging system consists of many different levels of operation and many different components that together form the overall logging system. All of these individual components are presented during the coverage of the system and their purpose and certain implementation details are presented.

A few components that are not directly related to the logging system are also presented together with the logging system. Although they are not directly part of the logging system, these components are used to interact with the logging system in various ways.

After the logging system is covered, a few different solutions for implementing audit logging for the Android platform and integrating it as a part of the Android logging system are presented.

The final chapter covers the actual analysis of the system and presents the results of the analysis. In this chapter, the system will be analyzed by one security requirement at a time with implementation modifications being presented to some extent, depending on the component being analyzed.

2 COMMON CRITERIA

Common Criteria and the accompanying Common Methodology for Information Technology Security Evaluation are a technical basis for an international agreement, the Common Criteria Recognition Arrangement (CCRA). (2.)

The purpose of this arrangement is to ensure that individual products are evaluated by competent and neutral licensed laboratories. These laboratories determine on what level the product fulfills the security properties that have been set by the Common Criteria. Fulfilling the security requirements set by the Common Criteria grants the evaluated product a certificate that is issued by one of the many Certificate Authorizing Schemes. (2.)

The Certificate Authorizing Schemes are government bodies that are responsible for IT security certification in their respective countries. The Certificate Authorizing Schemes only reside in the countries that are Certificate Authorizing Members of the CCRA. The licensed laboratories mentioned above also reside in the countries that are Certificate Authorizing Members of the CCRA.

Section 2.2 covers all of the member countries and their role in the CCRA.

2.1 Purpose of the Common Criteria

The main purpose of the Common Criteria is to ensure the quality and high standards of evaluation for Information Technology products and their respective protection profiles. The evaluations themselves are also expected to contribute to the confidence in the security of the products and their protection profiles. (3.)

The secondary purposes are to improve the availability of evaluated and security enhanced Information Technology products and their protection profiles, reduce duplicate evaluations of Information Technology products and their protection profiles and to improve the efficiency and cost-effectiveness of the evaluation and the certification processes. (3.)

2.2 CCRA members

The participants in the CCRA are government organizations or agencies that represent their respective countries. The members can be divided into two categories: producers of Evaluation certificates and consumers of Evaluation certificates. If a member country is a producer of Evaluation certificates, they of course can also participate in consuming the Evaluation certificates. On the other hand, if a member country is only a consumer of Evaluation certificates, they may not maintain an IT security Evaluation capability and therefore cannot produce Evaluation certificates. (4, p. 6)

Below are two tables of the CCRA member countries. Table 1 contains the member countries that produce Evaluation certificates. Table 2 contains the member countries that consume Evaluation certificates.

TABLE 1. Evaluation certificate producing CCRA members (5.)

Australia	Australasian Information Security Evaluation program (AISEP)
Canada	Communications Security Establishment (CSE)
France	Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI)
Germany	Bundesamt für Sicherheit in der Informationstechnik (BSI)
India	Indian Common Criteria Certification Scheme
Italy	Organismo di Certificazione della Sicurezza Informatica (OCSI)
Japan	Japan IT Security Evaluation and Certification Scheme (JISEC)
Malaysia	CyberSecurity Malaysia
Netherlands	The Netherlands scheme for Certification in the Area of IT Security (NSCIB)
New Zealand	Defence Signals Directorate (DSC)
Norway	SERTIT
Republic of Korea	IT Security Certification Center (ITSCC)
Spain	Organismo de Certificación de la Seguridad de las Tecnologías de la Información
Sweden	Swedish Certification Body for IT Security (CSEC)
Turkey	Turkish Standards Institution (TSE)
United Kingdom	UK IT Security Evaluation and Certification Scheme (NCSC)
United States	National Information Assurance Partnership (NIAP)

TABLE 2. Evaluation certificate consuming CCRA members (5.)

Austria	Federal Chancellery of Austria
Czech Republic	National Security Authority of the Czech Republic (NBU)
Denmark	Center for Cyber Security (CFCS)
Ethiopia	Information Network Security Agency (INSA)
Finland	Finnish Communications Regulatory Authority (FICORA)
Greece	National Intelligence Service (NIS)
Hungary	Ministry of National Development
Israel	The Standards Institution of Israel (SII)
Pakistan	Ministry of Defence
Qatar	Ministry of Transport and Communications (MOTC)
Singapore	Cyber Security Agency of Singapore (CSA)

2.3 Protection profiles

The Protection Profiles, which were mentioned previously, are documents that contain the specific requirements a product must fulfill in order to be qualified for the Common Criteria certification. The protection profiles also contain test cases assuring that the security requirements are met. The overall structure of the protection profiles is presented in more depth in the next subsections.

The protection profiles are defined based on the target product's category. The categories range from printer software to whole operating systems.

The full list of currently approved protection profiles can be found from the link in the references. (6.)

2.3.1 Structure of a protection profile

The main part of the protection profile consists of three main chapters: Security Problem Definition, Security Objectives and Security Requirements.

The overall structure of the Protection Profiles may differ based on the target category, but the main chapters are always present in the Protection Profiles. Each of the chapters is covered in more detail in the upcoming subsections.

Protection Profile for Mobile Device Fundamentals, the version 3.1 is used for the examples as it was the relevant Protection Profile for this thesis and it was the latest available Protection Profile for the Mobile Device category. (7.)

In addition to the previously mentioned chapters of the Protection Profile, the Mobile Device Fundamentals Protection Profile also contains documentation on the design of proper entropy, use case templates and other relevant documentation which are specific to the Mobile Device category. Other categories might contain similar documentation on information that is relevant to the category.

These specific chapters are not covered here, as they were not relevant for the completion of this thesis.

2.3.1.1 Security problem definition

The security problem definition chapter contains information about possible threats that the target product may face, conditions that are assumed as the normal operating conditions for the target product and other possible organizational security policies for the target product.

The threats that are defined are possible cases on how the target product and its operations could be compromised.

T.PHYSICAL
Physical Access

An attacker, with physical access, may attempt to access user data on the Mobile Device including credentials. These physical access threats may involve attacks, which attempt to access the device through external hardware ports, impersonate the user authentication mechanisms, through its user interface, and also through direct and possibly destructive access to its storage media. Note: Defending against device re-use after physical compromise is out of scope for this protection profile.

FIGURE 1. Threat defined in a Protection Profile (7, p. 14)

The assumptions are the assumed normal operating conditions of the target product. These can vary depending on the target product and its category.

A.CONFIG
It is assumed that the TOE's security functions are configured correctly in a manner to ensure that the TOE security policies will be enforced on all applicable network traffic flowing among the attached networks.

A.NOTIFY
It is assumed that the mobile user will immediately notify the administrator if the Mobile Device is lost or stolen.

A.PRECAUTION
It is assumed that the mobile user exercises precautions to reduce the risk of loss or theft of the Mobile Device.

FIGURE 2. Assumptions defined in a Protection Profile (7, p. 15)

2.3.1.2 Security objectives

Security objectives that have been defined are defined as such that they will counter the corresponding threats that were defined in the Security Problem Definition chapter.

Each of the security objectives are divided into smaller security requirements. Once the individual security requirements are all addressed, the security objective can be assumed to be capable of preventing the corresponding threats.

The security objective also contains a written description on how it addresses the corresponding threats.

O.STORAGE

Protected Storage

To address the issue of loss of confidentiality of user data in the event of loss of a Mobile Device (T.PHYSICAL), conformant TOEs will use data-at-rest protection. The TOE will be capable of encrypting data and keys stored on the device and will prevent unauthorized access to encrypted data.

Addressed by: FCS_CKM_EXT.1, FCS_CKM_EXT.2, FCS_CKM_EXT.3, FCS_CKM_EXT.4, FCS_CKM_EXT.5, FCS_CKM_EXT.6, FCS_CKM_EXT.7(OPTIONAL), FCS_COP.1(1), FCS_COP.1(2), FCS_COP.1(3), FCS_COP.1(4), FCS_COP.1(5), FCS_IV_EXT.1, FCS_RBG_EXT.1, FCS_RBG_EXT.2(OPTIONAL), FCS_RBG_EXT.3(OPTIONAL), FCS_STG_EXT.1, FCS_STG_EXT.2, FCS_STG_EXT.3, FDP_ACF_EXT.2(OPTIONAL), FDP_ACF_EXT.3(OPTIONAL), FDP_DAR_EXT.1, FDP_DAR_EXT.2, FIA_UAU_EXT.1, FPT_KST_EXT.1, FPT_KST_EXT.2, FPT_KST_EXT.3, FPT_JTA_EXT.1

FIGURE 3. Security objective in a Protection Profile (7, p. 16)

The security objectives chapter also contains objectives for the assumptions mentioned earlier. These objectives are mainly for setting the correct operational environment for the target product.

OE.CONFIG

TOE administrators will configure the Mobile Device security functions correctly to create the intended security policy

OE.NOTIFY

The Mobile User will immediately notify the administrator if the Mobile Device is lost or stolen.

OE.PRECAUTION

The Mobile User exercises precautions to reduce the risk of loss or theft of the Mobile Device.

FIGURE 4. Objectives for Operational Environment in a Protection Profile (7, p. 17-18)

The final part of the security objectives chapter is a security objectives rationale. This section is used to demonstrate how all of the above-mentioned threats, assumptions and organizational security policies relate to the security objectives.

The purpose of this segment is to rationalize how the security objectives counter the targeted threat. The segment contains a specific written description of the relation of objective and threat.

T.PHYSICAL	O.STORAGE, O.AUTH	The threat T.PHYSICAL is countered by O.STORAGE as this provides the capability to encrypt all user and enterprise data and authentication keys to ensure the confidentiality of data that it stores. The threat T.PHYSICAL is countered by O.AUTH as this provides the capability to authenticate the user prior to accessing protected functionality and data.

FIGURE 5. Threat rationale in a Protection Profile (7, p. 18)

The first column contains the threat or an assumption. The second column contains the security objectives which, when completed, will counter the relevant threat or realize the assumption. The last column contains the written rationale.

The security objectives rationale also contains the rationalization on how the certain security objectives realize the assumptions that were set in the security problem definition section of the document.

An example of the rationalization of these assumptions can be found in figure 6 below.

A.CONFIG	OE.CONFIG	The operational environment objective OE.CONFIG is realized through A.CONFIG.
A.NOTIFY	OE.NOTIFY	The operational environment objective OE.NOTIFY is realized through A.NOTIFY.
A.PRECAUTION	OE.PRECAUTION	The operational environment objective OE.PRECAUTION is realized through A.PRECAUTION.

FIGURE 6. Assumption rationale in a Protection Profile (7, p. 19)

2.3.1.3 Security requirements

Security requirements are the individual security features that the target product has to implement in order to qualify for certification.

All security requirements are sorted under specific classes. Below in figure 7 is an example of a class for security audit related security requirements. All individual security requirements share the same identifier as the base class. The identifier is specified in the brackets after the class name.

5.1.1 Class: Security Audit (FAU)

FIGURE 7. Security requirement class in a Protection Profile (7, p. 20)

The security requirements themselves can also be divided into smaller separate requirements. These small separate requirements are related to each other and supplement each other to fulfill the main security requirement.

Each security requirement also contains an Assurance Activity. This is used to determine how to verify the functionality and to be assured that the security requirement is met. The Assurance Activity contains very specific test steps on how the verification of the security requirement is to be done.

FAU_STG.1 Audit Storage Protection

FAU_STG.1.1 The TSF shall protect the stored audit records in the audit trail from unauthorized deletion.

FAU_STG.1.2 The TSF shall be able to prevent unauthorized modifications to the stored audit records in the audit trail.

Assurance Activity

The evaluator shall ensure that the TSS lists the location of all logs and the access controls of those files such that unauthorized modification and deletion are prevented.

- ***Test 1:*** *The evaluator shall attempt to delete the audit trail in a manner that the access controls should prevent (as an unauthorized user) and shall verify that the attempt fails.*
- ***Test 2:*** *The evaluator shall attempt to modify the audit trail in a manner that the access controls should prevent (as an unauthorized application) and shall verify that the attempt fails.*

FIGURE 8. Security requirement in a Protection Profile (7, p. 27)

2.4 Certified products

The Common Criteria organization has a list of products that have been certified and are compliant with either of the following conditions:

The product is compliant with a collaborative Protection Profile that has been developed and maintained in accordance with CCRA Annex K (4, p. 46) with a selected Evaluation Assurance level (EAL) of at least 4, including flaw remediation (ALC_FLR). (7.)

The product is compliant with the EAL levels of 1 and 2, including the flaw remediation. (7.)

The Evaluation Assurance Levels are covered in great detail in part 3 of the currently latest Common Criteria release. (9, chapter 8.)

The flaw remediation is also covered in part 3 of the currently latest Common Criteria release (9, section 15.5.)

The certified products themselves cover a lot of different categories. A list of the key categories can be seen in figure 9 below.

After 1st of June 2019, products that have an expired certificate will be moved to an Archive list that is also available on the CCRA portal. (8.)

FIGURE 9. List of Common Criteria product categories (8.)

2.4.1 Example certified product

Figure 10 below contains an example of a certified product found in the Common Criteria Certified Products list.

Product	Vendor	Product Certificate	Date Certificate Issued	Certificate Validity Expiration Date	Compliance	Scheme
Apple iOS 11 Certification Report Security Target Extended Package for Mobile Device Management (MDM) Agents Protection Profile for Mobile Device Fundamentals, Version 3.1 Extended Package for Wireless Local Area Network (WLAN) Clients	Apple Computer, Inc.	CCRA Certificate	2018-03-30		PP Compliant	US

FIGURE 10. Certified product found in the Common Criteria Certified Products list (8.)

The product listing itself contains the name of the product and the vendor of the product. In this case the product is Apple's iOS 11 operating system. In addition, the listing contains the product certificate itself, the date it was issued and the date it will expire. The actual certificate can be seen in figure 11 below.

There is also a mention about the compliance that was met to be qualified for the certification. In this case the product was compliant with the relevant Protection Profiles. The scheme, which granted the certificate, is also listed.

And lastly below the product's name are all the relevant references to files such as the Protection Profiles that were used for the compliance validation.

National Information Assurance Partnership

Common Criteria Certificate

is awarded to

Apple Inc

for

Apple iOS 11

Common Criteria

The IT product identified in this certificate has been evaluated at an accredited testing laboratory using the Common Methodology for IT Security Evaluation (Version 3.1) for conformance to the Common Criteria for IT Security Evaluation (Version 3.1). This certificate applies only to the specific version and release of the product in its evaluated configuration. The product's functional and assurance security specifications are contained in its security target. The evaluation has been conducted in accordance with the provisions of the NIAP Common Criteria Evaluation and Validation Scheme and the conclusions of the testing laboratory in the evaluation technical report are consistent with the evidence adduced. This certificate is not an endorsement of the IT product by any agency of the U.S. Government and no warranty of the IT product is either expressed or implied.

Date Issued: 2018-03-30

Validation Report Number: CCEVS-VR-VID10851-2018

CCTL: atsec information security corporation

Assurance Level: PP Compliant

Protection Profile Identifier:
Extended Package for Mobile Device Management Agents Version 3.0
Protection Profile for Mobile Device Fundamentals Version 3.1
Extended Package for Wireless LAN Client Version 1.0

Original Signed By
Director, Common Criteria Evaluation and Validation Scheme
National Information Assurance Partnership

Original Signed By
Deputy National Manager National Security Systems
National Security Agency

FIGURE 11. Apple iOS 11 Common Criteria certificate (8.)

3 SECURITY AUDIT REQUIREMENTS

This chapter will present all of the individual security requirements that need to be implemented to cover the Security Audit requirements of the Protection Profile.

The Protection Profile, which is used for determining the requirements, is the Protection Profile for Mobile Device Fundamentals, Version 3.1. (7.)

In certain security requirements, the Protection Profile provides a possibility to select from a limited number of implementation options. An example of this kind of selection can be seen in figure 12 below.

FCS_CKM_EXT.5	*[**selection**: Failure of the wipe, None].*	*No additional information.*

FIGURE 12. Selection in a Protection Profile security requirement. (7, p. 22)

In this case the selection is whether or not to implement audit recording for the event in which the wipe of protected data fails.

Both options are as valid as the other and the choice will not affect the validation of the product or the possibility of certification.

For the safety of the product it would, of course, be better to include the auditing of as much of security related events as possible, but it is not always viable or even possible in some cases to include all the relevant events.

Due to strict time constraints of this thesis, during these kinds of selections where it was possible to decide whether or not to implement certain functionality, it was decided not to implement the functionality.

Regardless of the selection choices made during this thesis, it is possible to include the auditing for more of the events at a later date.

3.1 Audit Data Generation – FAU_GEN.1

This security requirement defines requirements for the generation of audit data and the specific format in which the audit data is to be recorded in the audit records.

3.1.1 FAU_GEN.1.1

This requirement defines events during which the security functionality should generate an audit record of the event.

Below is a compiled list of all the events that the security functionalities will need to be able to audit in order to fulfill the security requirement.

The requirement required referencing other sections of the Protection Profile for locating certain auditable events, all of which have also been included in the list. This was done to help the reader better understand the overall requirements and to remove the need for the user to study the actual Protection Profile.

Compiled list of auditable events required by the security requirement:

1. Start-up and shutdown of the audit functions (7, p. 20)
    - Audit record will be generated at the start-up and the shutdown of the audit functionalities

2. All administrative actions (7, p. 20)
    - An audit record will be generated in the event of an applicable administrative action
    - Applicable administrative actions: (7, p. 93)
        - Configuring of password policy
        - Configuring of session locking policy
        - Configuring of application installation policy

3. Start-up and shutdown of the Rich OS (7, p. 20)
    - An audit record will be generated at the start-up and the shutdown of the Android operating system (Rich OS)

4. Insertion or removal of removable media (7, p. 20)
    - If the target supports external removable media, an audit record will be generated in the event of insertion or removal of that media

5. Specifically defined auditable events (7, p. 20)
    - An audit record will be generated in cases of specifically defined auditable events
    - List of the defined auditable events can be found in table 3

6. Audit records reaching certain percentage of the audit capacity (7, p. 20)
    - An audit record will be generated in the event that the audit records are reaching a set threshold of the full capacity.

7. Specifically defined optional auditable events (7, p. 20)
    - An audit record will be generated in cases of specifically defined optional auditable events
    - List of the included optional auditable events can be found in table 4

TABLE 3. Compiled list of the specifically defined auditable events (7, p. 22-23)

Requirement	Auditable event	Additional information
FCS_STG_EXT.1	Import or destruction of key	Identity of key, Role and identity of requestor
FCS_STG_EXT.3	Failure to verify integrity of stored key	Identity of key being verified
FDP_DAR_EXT.2	Failure to encrypt / decrypt data	No additional information
FDP_STG_EXT.1	Addition or removal of certificate from Trust Anchor Database	Subject name of certificate
FIA_X509_EXT.1	Failure to validate X.509v3 certificate	Reason for failure of validation
FPT_TST_EXT.1	Initiation / Failure of self-test	No additional information
FPT_TXT_EXT.2	Start-up of the target product	No additional information

TABLE 4. Compiled list of included optional auditable events (7, p. 23-26)

Requirement	Auditable event	Additional information
FAU_SEL.1	Modifications to audit configuration while audit collection is operating	No additional information
FMT_SMF_EXT.1	Initiation of application installation or update	Name and version of application

3.1.2 FAU_GEN.1.2

This requirement defines how the data from the auditable events listed in the previous subsection is to be stored in the audit records.

Each event that is stored into the audit records needs to contain at least the following information: (7, p. 20-21)

- Date and time of the event
- Type of the event
    - Type of event is indicated by the severity level of the event
    - Severity levels: “Info”, “Warning” or “Error”
- Subject identity
    - Process name or the process ID
    - Other unique identifier
- Outcome of the event
    - Indicating whether the event was a success or a failure
- Additional information
    - Additional information is required in the case that the event is one of the specifically defined auditable events in Table 3 or one of the optional auditable events in Table 4.

3.2 Audit Storage Protection – FAU_STG.1

This requirement states that the security functionality should be able to protect the stored audit records from unauthorized deletion. The requirement also states that the security functionality will be able to prevent unauthorized modifications of the audit records. (7, p. 27)

3.3 Prevention of Audit Data Loss – FAU_STG.4

This requirement states that the security functionality should overwrite the oldest audit record if the full capacity of the audit records is reached. (7, p. 27)

3.4 Audit Review – FAU_SAR.1

This security requirement is currently optional and is not required to be fulfilled but is expected to be included as soon as possible and may become mandatory in the future. (7, p. 145)

As it is possible to include the required functionality, it should be included so that it will not have to be included separately in the future.

The requirement states that the security functionality should provide an administrator the capability to read all auditable events from the audit records. This access to audit records can be for example granted through an API or via a Mobile Device Management (MDM) agent. (7, p. 145)

The audit records should also be provided in a manner that is suitable for the administrator to interpret. (7, p. 145)

The requirement, however, does not set any assumptions on the knowledge or expertise level of the administrator interpreting the audit records. Taking this into consideration, the audit records should be kept as simple as possible and they should not include any technical information that would require extensive knowledge or expertise of the subject.

3.5 Selective Audit – FAU_SEL.1

This security requirement is also optional similar to the previous one, which defined requirements for audit record reviewing.

The requirement states that the security functionality should be able to select a subset of events to be audited from the all of the auditable events. This should be achieved by selecting the auditable events based on the following attributes: (7, p. 145)

1. Event type
2. Success of auditable events
3. Failure of auditable events

4 ANDROID LOGGING SYSTEM

This chapter will present Android's logging system and the individual components that are involved. The chapter will also present certain relevant implementation details from some of the logging system's key components.

The logging system is covered starting from the perspective of a traditional Java and a native application all the way down to the core of the whole logging system, which is the Logd logging daemon.

This chapter will also cover Android's log viewing tool Logcat and the communication method between an Android device and a host PC.

An overview of the whole Android logging system can be found below in figure 13. This overview will be referenced multiple times in the upcoming sections and it is shortly explained below.

The components of the Android logging system are presented in unique colors to help better understand the hierarchy of the system. The components sharing the same color are part of the same structure.

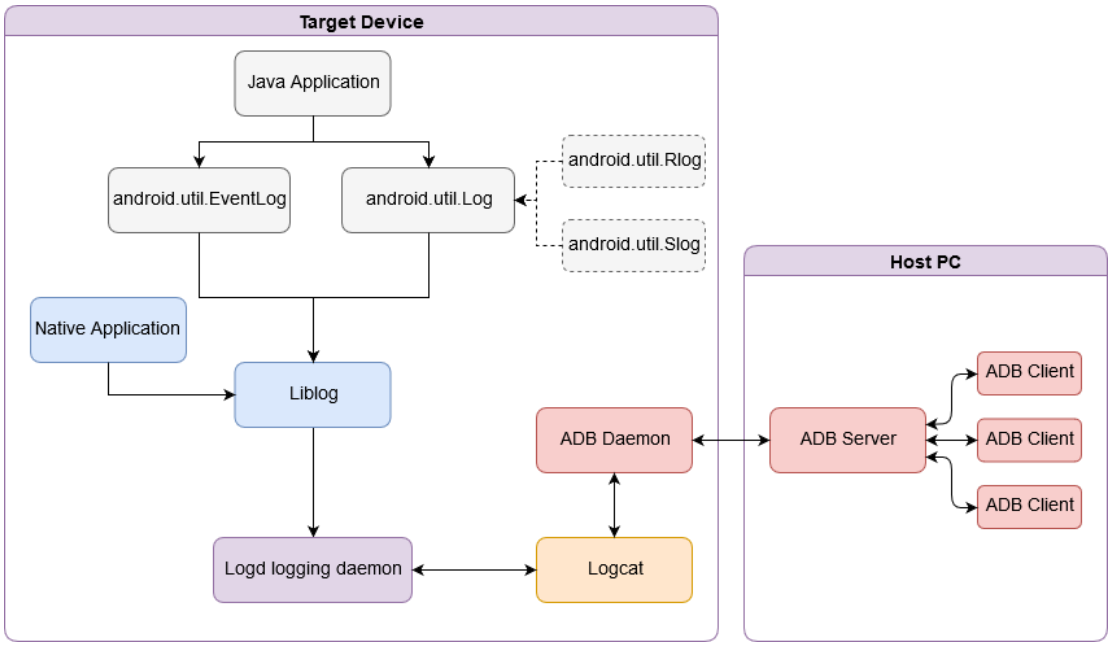

FIGURE 13. Overview of the Android Logging System

4.1 Java application

Traditional Java applications run under the Android operating system and within an Android Runtime (ART) environment. ART has replaced the previously used Dalvik Virtual Machine (DVM) in the Android platform. ART is used to translate Dalvik bytecode, previously used by DVM, into native instructions. Dalvik bytecode in turn is translated from Java bytecode, which in turn is compiled from the Java files of the application. (10.)

Topics, such as how Android handles its applications, how Java code is compiled into Java bytecode and converted into Dalvik bytecode, or how the Dalvik bytecode is translated into native instructions by ART, are beyond the scope of this thesis.

At the application level the logging system can be easily utilized by using Android's logging APIs for sending log messages. Android currently provides four separate logging APIs for logging messages.

The reason for the existence of four separate APIs is due to the implementation of the underlying logging system. The logging system has five separate buffers for different kinds of log events. The five main buffers are: main, radio, event, system and crash buffers. These buffers are managed by the Logd logging daemon, which was shown in figure 13 at the beginning of this chapter. The logging daemon will be presented on its own later in section 4.3.

The four APIs, which Android offers, are used to log message to the main, radio, event and system buffers.

There is a way to log messages to the crash buffer but as Android has no mentioning of it in its documentation, it is most likely not officially supported and therefore not presented.

Java also offers an API for logging, but as the use of Android's logging APIs is preferred over the use of Java's API, the Java API is not presented in further detail. For potential further future study, the documentation on the Java API can be found from the references. (11.)

The buffer names themselves are a good indication of the type of log messages they are mainly used to store:

- Main buffer
    - Logging messages sent by normal applications
- System buffer
    - Logging messages sent by Android's system services
- Radio buffer
    - Logging of radio and telephony related messages
- Event buffer
    - Logging of system diagnostic events (12.)
- Crash buffer
    - Logging of crash events and stack traces

Although Android offers multiple separate API's, which can be used, the actual implementation of, different APIs is very similar to one and another when disregarding a few exceptions.

The system and radio logging classes are mostly wrappers for the base *Log* class. The base *Log* class in question here is the main logging API, which is used for normal applications to log messages to the main buffer.

The base Log class contains definitions of identification numbers for each of the five different buffers mentioned earlier. The wrapper implementations then reference the appropriate identification number. The logging system uses these identification numbers to select the appropriate buffer for the log messages.

```
367     /** @hide */ public static final int LOG_ID_MAIN = 0;
368     /** @hide */ public static final int LOG_ID_RADIO = 1;
369     /** @hide */ public static final int LOG_ID_EVENTS = 2;
370     /** @hide */ public static final int LOG_ID_SYSTEM = 3;
371     /** @hide */ public static final int LOG_ID_CRASH = 4;
```

FIGURE 14. Android Log.java log buffer identification numbers (13.)

The implementation for the EventLog class is a bit more complex than the implementation of the radio and the system logging classes.

This is mainly because the event buffer stores the messages in a binary format instead of the normal text message format which the other buffers use. (14.)

The actual use of the four APIs corresponding with the target buffers is straight forward. The APIs can be utilized by importing the appropriate classes. The import of these classes can be seen in figure 15 below.

```
import android.util.Log;
import android.util.Slog;
import android.telephony.Rlog;
import android.util.EventLog;
```

FIGURE 15. Importing of Android's logging classes

The most relevant of these logging classes is the base *Log* class. The rest are used under very specific circumstances and some are not even meant to be used for standard applications and are made to be used by device manufacturers for logging of Android system services.

As this subsection is about logging from a standard application, the focus is on the base logging class.

After importing the main logging class, it can be easily used by calling the class methods for sending log messages. The class contains methods for several different severities of log events. This can be seen in figure 16 below.

The different severity levels are: *ERROR, WARN, INFO, DEBUG* and *VERBOSE*. (15.)

```
Log.v("Tag","Message"); //Verbose level log entry
Log.d("Tag","Message"); //Debug level log entry
Log.i("Tag","Message"); //Info level log entry
Log.w("Tag","Message"); //Warning level log entry
Log.e("Tag","Message"); //Error level log entry
```

FIGURE 16. Android main Log class methods used for sending log messages

The tag parameter is used to identify the source of the log message. The message parameter is the message that is logged. (15.)

4.2 Native application

The difference between a Java application and a Native application on Android is that the native application is running natively as the name suggests. Running natively means that the application is written in native programming languages, such as C/C++, which is then compiled straight into machine code whereas Java code is compiled into Java bytecode, then converted into Dalvik bytecode and then finally translated into native instructions.

There are both advantages and disadvantages to writing a native application over a Java application.

The main advantage of writing a native application over a Java application is the boost in the performance of the application. As the application is running natively on the processor instead of being translated into native instructions from the Dalvik bytecode, the removal of this translation process provides a performance boost to the application.

Another big advantage is the ability to manually manage the application's memory. Native programming languages provide support for manually allocating and deallocating memory for the application. On Java this kind of memory manipulation is not possible as Java uses automatic memory management.

The manual memory manipulation can also be a disadvantage. If the memory is managed incorrectly, the application can become vulnerable to external threats and cause further security threats to the whole system. This problem is nonexistent in Java as the programmer has no manual control over the memory and the human error in memory management is not possible.

In the right hands the manual memory management provided with native programming languages is an advantage as it increases the efficiency of the memory handling and in turn the performance of the application. Using native languages also reduces the overall memory footprint of the application.

As mentioned previously, writing native applications does not come without any drawbacks.

When writing native applications, the applications need to be compiled for multiple different processor architectures, as there are various Android devices running with multiple different processor architectures. All of the different versions need to be included in the Android Application Package (APK) that the user installs on the Android device. This will of course increase the overall size of the APK.

Another drawback of using native languages over Java is that Android does not provide access to all the same APIs that Java has access to. This can be a deal breaker for certain kinds of applications that need to be able to access APIs that are only available for Java applications.

Usual cases when native applications are written over Java applications are performance demanding applications, such as games or physics-based applications, as using native languages provides access to various high-performance libraries such, as OpenGL or Vulkan.

The logging library used on the native level of the Android platform is the Liblog library.
The Liblog library can be taken into use by including the library header and linking the dynamic logging library during the compilation of the application.

```
LOCAL_LDLIBS := -llog
```

FIGURE 17. Linking the Liblog library (16.)

```
#include <android/log.h>

#define LOG_VERBOSE(...) __android_log_print(ANDROID_LOG_VERBOSE,"Tag",__VA_ARGS__)
#define LOG_DEBUG(...) __android_log_print(ANDROID_LOG_DEBUG,"Tag",__VA_ARGS__)
#define LOG_INFO(...) __android_log_print(ANDROID_LOG_INFO,"Tag",__VA_ARGS__)
#define LOG_WARNING(...) __android_log_print(ANDROID_LOG_WARNING,"Tag",__VA_ARGS__)
#define LOG_ERROR(...) __android_log_print(ANDROID_LOG_ERROR,"Tag",__VA_ARGS__)
```

FIGURE 18. Liblog logging to main buffer

The macros in the figure 18 above are for logging to the main buffer. The included header does not provide functionality for logging to other buffers besides the main buffer. In order to log to the other buffers in the logging system, a different header of the Liblog library needs to be included. The header in question can be seen in figure 19 below.

```
#include <log/log.h>

#define LOG_TAG "Tag"

ALOGV("Main Buffer - Verbose");
ALOGD("Main Buffer - Debug");
ALOGI("Main Buffer - Info");
ALOGW("Main Buffer - Warning");
ALOGE("Main Buffer - Error");

SLOGD("System Buffer - Debug");
RLOGD("Radio Buffer - Debug");

LOG_EVENT_INT(tag,value);
LOG_EVENT_LONG(tag,value);

...
```

FIGURE 19. Liblog logging to multiple buffers

The library has predefined macros for logging to the previously presented buffers. The library uses the user defined *LOG_TAG* as the tag for the log message. The severity level is set automatically based on the used macro.

The *ALOGV, ALOGD, e.g.* macros use the same logging functionality, but with different severity levels. The severity is set in the macro definition, as can be seen in figure 20 below. In this case the severity level is set to *LOG_DEBUG*.

The first letter of the macro indicates the target buffer in question: *ALOG* for main buffer, *SLOG* for system buffer and *RLOG* for radio buffer. The last letter indicates the severity level. Logging to the event buffer has its own separate macros: *LOG_EVENT_INT* and *LOG_EVENT_LONG*.

```
202 #ifndef ALOGD
203 #define ALOGD(...) ((void)ALOG(LOG_DEBUG, LOG_TAG, __VA_ARGS__))
204 #endif
```

FIGURE 20. Liblog implementation of ALOGD macro (17.)

4.3 Logd logging daemon

As was shown earlier in figure 13, Logd is the core of the whole Android logging system.
Logd is a logging daemon that ultimately handles all of the incoming log messages and stores them in their respective buffers. The logging daemon is a service that runs in Linux user space.

Current implementation of the logging daemon has five buffers for different categories of log messages. These buffers were introduced earlier in the previous sections where the logging at the application level was presented.

The logging daemon creates three sockets at the start of its program. These sockets are used to receive logs, transfer the contents of the log buffers and to provide a control interface for the daemon itself. The logging daemon also interacts with multiple other interfaces and sockets but as they are not created by the logging daemon, they exist independently of the logging daemon.

The three sockets that are created by the logging daemon:

- Socket for receiving logs from the Liblog library
    - */dev/socket/logdw/*
- Socket for sending log buffer contents
    - */dev/socket/logdr/*
- Socket for controlling the logging daemon itself
    - */dev/socket/logd/*

The log messages, which are stored in the five main buffers, are received from the first of the created sockets: */dev/socket/logdw/*. This socket is where the majority of the log messages come from, as all of the log messages that were sent by applications and Android's system services are received through this socket.
The Liblog library writes the logs it receives from the upper level Java APIs to this socket. The library also writes the logs that are logged at the native level, by the library itself, to this socket.

In addition to the five buffers mentioned earlier, the logging daemon also has support for a sixth optional buffer, which can be used to store kernel log messages. When kernel logging is enabled, the logging daemon reads kernel log messages from the kernel's ring buffer through an interface in: */proc/kmsg/.*

Each individual buffer, which was mentioned earlier, has a certain capacity of how much data it can store. The buffers, which the logging daemon manages, are called ring buffers or circular buffers. This means that when the full capacity is reached, the oldest log messages are removed to make room for new log messages. The buffer can be thought of as a circle that is connected from one end to the other.

By default, the size for each of the buffers is set at 256 kilobytes. The logging daemon also provides the possibility to manually configure the size of each buffer. The maximum size, which can be configured for an individual buffer, is restricted at 256 megabytes.

In figure 21 below is an example of the buffer sizes. In the example, the buffers have been configured to a maximum size of 4 megabytes per buffer.

When inspecting the buffer sizes, the logging daemon also reports the current status of each buffer. The status indicates the current consumption level of each buffer. With this information, the user can easily calculate the remaining free space in the buffers.

```
main: ring buffer is 4Mb (2Mb consumed), max entry is 5120b, max payload is 4068b
radio: ring buffer is 4Mb (188Kb consumed), max entry is 5120b, max payload is 4068b
events: ring buffer is 4Mb (154Kb consumed), max entry is 5120b, max payload is 4068b
system: ring buffer is 4Mb (174Kb consumed), max entry is 5120b, max payload is 4068b
crash: ring buffer is 4Mb (2Kb consumed), max entry is 5120b, max payload is 4068b
kernel: ring buffer is 4Mb (153Kb consumed), max entry is 5120b, max payload is 4068b
```

FIGURE 21. Example of logging daemon buffer sizes

The buffer sizes and their current status can be viewed using Logcat, as was done in the above figure. Logcat will be covered separately in the next section. The buffers can also be manually configured using Logcat.

The logging daemon also provides the possibility to store log messages of Security-Enhanced Linux (SELinux).

SELinux is a Linux Security Module that is used to enforce mandatory access control over all running processes. More information on SELinux, its concept and how it has been incorporated into Android can be found from the documentation on Android Security. (18.)

Log messages from SELinux are read through a Netlink socket using *NETLINK_AUDIT* protocol to access the audit logging subsystem of Linux.

Netlink sockets are asynchronous sockets which are used for Inter-process communication (IPC) between kernel and user space processes. These sockets can also be used for IPC between user space processes but normal UNIX sockets are preferred to be used in this case. (19.)

Netlink currently has support for multiple protocols, including the *NETLINK_AUDIT* protocol, which is used for reading the SELinux log messages from the kernel's ring buffer.

The Netlink sockets were first implemented to provide access to the network control plane of the Linux kernel from regular user-space programs.

The log messages from SELinux can be stored in the main and the system buffers, depending on the configuration of the logging daemon. The logging daemon can also be configured to write the SELinux log messages back to the kernel's ring buffer. The messages are written to a character device that belongs to the audit logging subsystem: */dev/kmsg/.*

The possibility to write the SELinux log messages back to the kernel's ring buffer is provided, because when the logging daemon reads the SELinux log messages from the Netlink socket, they are removed from the kernel's ring buffer. This provides the functionality to keep the SELinux log messages both in the logging daemons own buffers and the kernel's ring buffer.

The contents of the log buffers can be read from another one of its created sockets: */dev/socket/logdr/*. This socket is used by Logcat to receive the log buffer contents from the logging daemon.

When reading logs from the above mentioned socket, the reader first needs to instruct the logging daemon with certain parameters to define the logs that the reader wants to receive from the logging daemon. The instructions are sent to the logging daemon using the same aforementioned socket.

There are a total of six parameters from which the logging daemon determines to send the requested logs. Currently, only five of the six parameters are implemented to be configurable.

The parameters, their corresponding Logcat flags and their meaning can be seen in table 5 below.

TABLE 5. Logging daemon log reading instruction parameters

Parameter	Logcat Flag	Description
tail=<value>	-t, -T <count>	Number of log lines to read. Selects the logs starting from the most recent one
start=<value>	-t, -T <time>	Start time of logs to dump. Default is set to EPOCH time
timeout=<value>	-	Feature not yet implemented
lids=<value>	-b <value>	Log ID's to send. This refers to the buffers from which the logs should be sent.
pid=<value>	--pid <value>	Logs from specific process ID
dumpAndClose	-d	Exit from the reader thread when log dumping is done

The last of the three sockets, which the logging daemon controls, is the control interface for the logging daemon itself. The socket in question is: */dev/socket/logd*.

The logging daemon can be manually configured by sending certain commands to the command interface socket. Most of the manual configurations, which can be done to the logging daemon through the control interface, are for the buffers that the logging daemon manages.

The configuration parameters can be seen in table 6 below.

TABLE 6. Logcat flags used to configure the logging daemon (20.)

Logd command	Logcat Flag	Description
Clear	-c, --clear	Clear the selected buffers and exit logcat. Default buffers are main, system and crash buffer.
GetBufSize GetBufSizeUsed	-g, --buffer-size	Print the size of the specified log buffer and exit logcat. Default buffers are main, system and crash buffer.
SetBufSize	-G <size>	Set the size of the log buffer. Default buffers are main, system and crash buffer. K or M can be added after the size to indicate kilo- or megabytes.
GetStatistics	-S	Include statistics in the output, number of logs by UID, PID, TID, etc.
GetPruneList	-p, --prune	Read current white- and blacklists
SetPruneList	-P, --prune <list>	Write new white- and blacklists

The overall logging functionality of the logging daemon that has been presented in this section is visually demonstrated in figure 22 below.

The figure contains all of the previously presented components, sockets and interfaces that interact with the logging daemon.

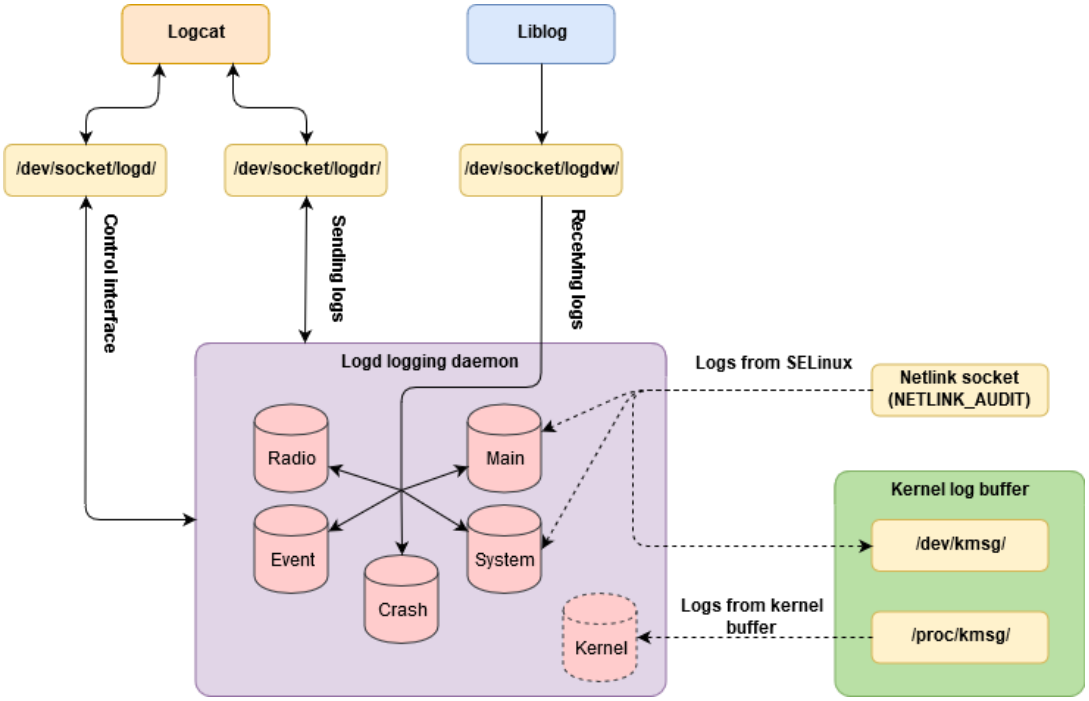

FIGURE 22. Overview of the logging daemon and its interaction with various interfaces

4.4 Logcat command line tool

Logcat is a command line tool that can be used to retrieve log messages which have been logged by the logging system. (20.)

Logcat also acts as the control interface for the Logd logging daemon, which was presented in the previous section.

Regardless of this, as was seen earlier in figure 13, Logcat is a completely separate program from the logging daemon.

In order to use Logcat, an Android Debug Bridge (ADB) connection to the target device is required. ADB is a versatile command line tool that provides functionality for a host PC to communicate with a target Android device.

ADB will be covered separately in the next section.

The Logcat tool can be utilized from an ADB client by issuing a *logcat* command and passing the arguments desired accompanied with the corresponding flags.

In addition to using the *logcat* command directly from the ADB client, the command can be used after creating a shell connection to the device. The shell connection can be created by using a *shell* command in the ADB client as seen in figure 23 below.

```
$ adb shell
# logcat
```

FIGURE 23. Creating shell connection with the target device (20.)

By executing the shell command, the ADB client provides the user access to a UNIX shell that can be used to run various commands on the device. All of the standard UNIX commands are not necessarily available to be executed on the device, as the Android shell only contains a subset of the standard UNIX commands.

As Logcat is mainly used for retrieving the contents of the logging daemons log buffers, Logcat provides extensive configurations on the formatting of its output and filtering of the actual log contents.

An example of filtering Logcat output based on selected buffers can be seen in figure 24 below. The buffers to be read are chosen with the *–b* flag.

```
logcat -b main -b radio -b events
```

FIGURE 24. Filtering logcat output based on selected buffers (20.)

The regular format of Logcat output can be seen in figure 25 below.

```
06-04 09:52:22.373  2426  2426 D KeyguardViewBase: show()
06-04 09:52:22.374  2426  2426 V KeyguardSecurityView: showPrimarySecurityScreen(turningOff=false)
06-04 09:52:22.374  2426  2426 D KeyguardSecurityView: showSecurityScreen(None)
```

FIGURE 25. Standard logcat output format

Logcat provides certain options from which the user can select the type of output format for the retrieved logs.

```
adb logcat -v thread
```

FIGURE 26. Logcat output formatting example (20.)

In the example shown in figure 26, the output format for the logs is set as *thread* with the *–v* flag.

In this output format, the output will print the severity level of the log message, the process ID, the thread ID and the log message itself. (20.)

An example of this output format can be seen in figure 27 below.

```
I( 1706: 1743) User[0] Flushing usage stats to disk
I( 1039: 1039) Enters wifigbk_ageOutHistory
I( 1039: 1039) Enters wifigbk_ageOutHistory
```

FIGURE 27. Logcat output format set as thread

4.5 Android Debug Bridge

Android Debug Bridge (ADB) is a command-line tool that consists of three individual components, all of which were visible in figure 13 shown earlier.

The first of the three components of ADB is an ADB client. The ADB client is a program running on the host PC. The ADB client is the interface through which the user can interact with a target device.

The second component is an ADB server, which is a background service running also on the host PC. The purpose of the ADB server component is to provide access to an ADB daemon service running on the target device. The ADB server is also responsible for managing the connections between multiple ADB clients and the ADB daemon running on the target device.

The last component of the ADB communication structure is the ADB daemon. The ADB daemon is a service that is running on the target device. The ADB daemon is ultimately the one responsible for executing the commands user has entered through the ADB client.

ADB can also be used to communicate with a device that is running inside an emulator on the host PC.

When an ADB client is first started, the client will check if an ADB server is running on the host PC. If the ADB server is not running prior to starting the ADB client, the client will automatically start the ADB server process. (21.)

When the ADB server is started, it will bind itself to a local TCP socket listening on port 5037. All of the ADB clients that connect to the ADB server will connect by using the port 5037. (21.)

After the ADB server has started, the server will try to find running ADB daemons over TCP or USB. The ADB Server is set to USB mode by default. This means that when the ADB server looks for the running ADB daemons, it will look for them initially over USB instead of over TCP.

ADB server can be switched between USB and TCP modes.

In order for the ADB server to be able to connect to the ADB daemon running on the target device, USB debugging option must be enabled on the device. USB debugging can be enabled from the target device's developer options.

To verify that the ADB daemon of the target device is properly connected to the ADB server, the ADB clients can list devices that are currently connected to the ADB server.

The command used by ADB clients to list the connected devices is *adb devices.*

An example of the *adb devices* command output can be seen in figure 28.

```
List of devices attached
XX123456789      device
```

FIGURE 28. List of connected devices

As the result of the command, the ADB server lists all of the currently connected devices and their unique serial numbers.

A visual illustration of the ADB communication flow, which has been presented so far, can be seen in figure 29 below.

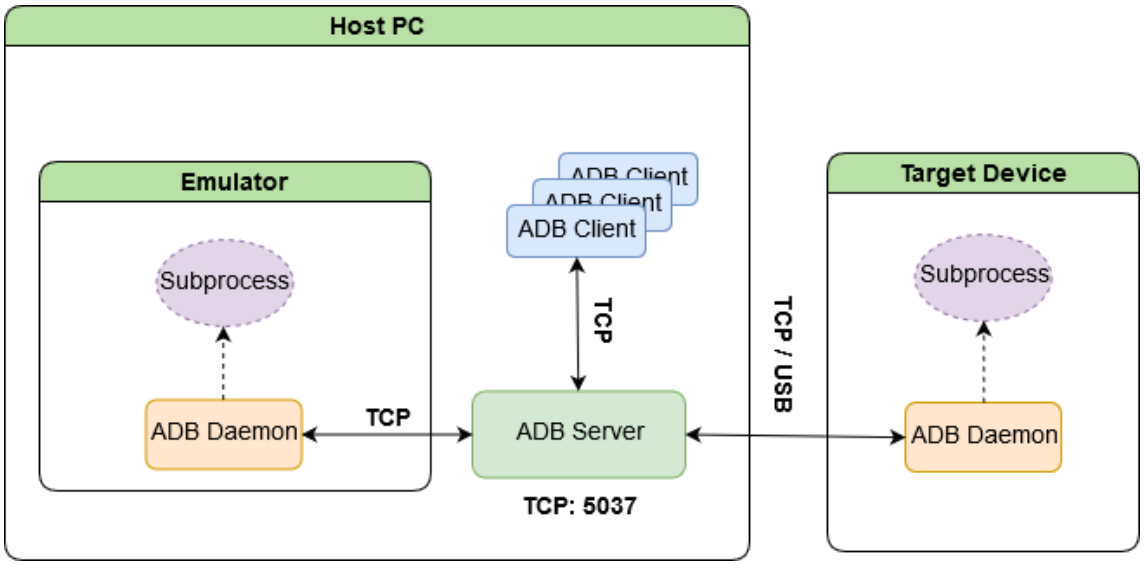

FIGURE 29. ADB communication flow

After a connection has been created between the ADB server and the target device's ADB daemon, the connection can then be used by multiple ADB clients

to transfer data between the host and the target device and to execute various commands on the target device.

One of the ways to execute commands on the target device is to create a shell connection to the device, as was briefly mentioned in the previous section.

Figure 30 shows a sequence chart, which illustrates the steps taken when executing a shell command *ls* on the target device. *ls* is a basic UNIX command that lists files in the current working directory.

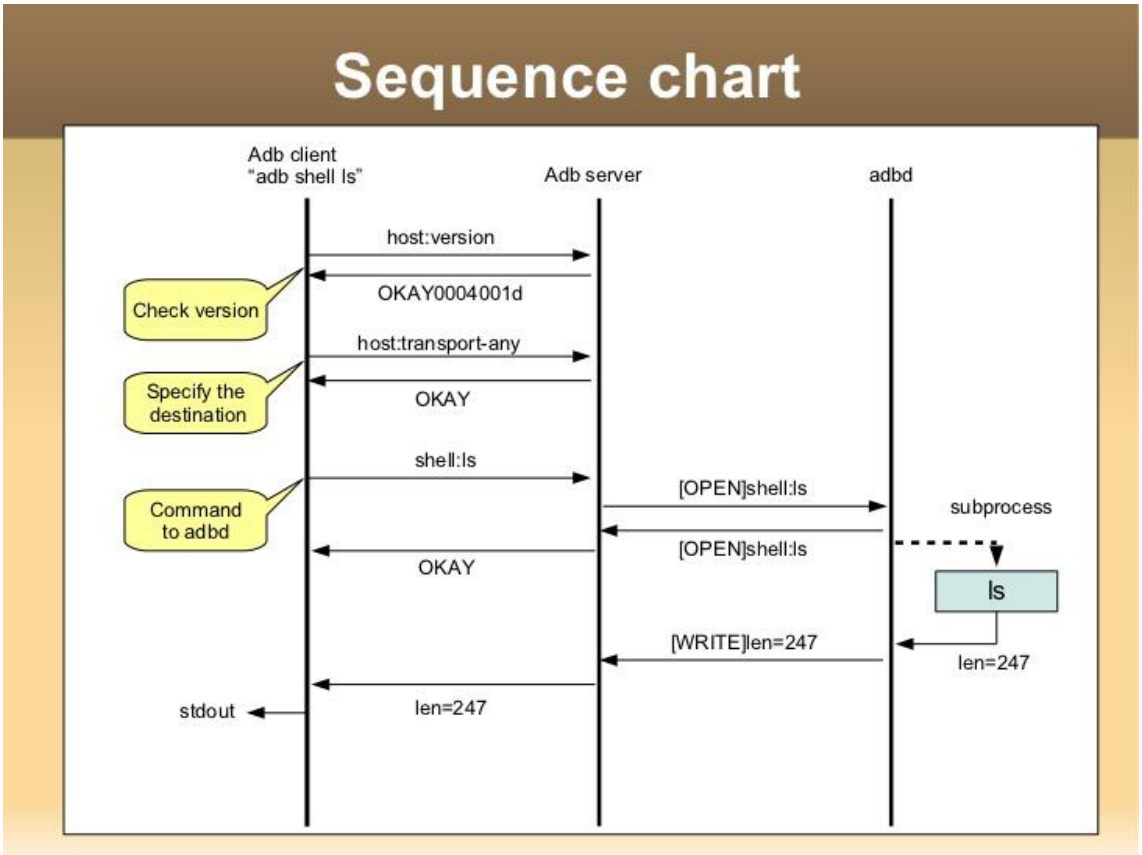

FIGURE 30. Sequence chart of shell command execution on a target device (22, p. 17)

Adbd in the above figure is the ADB daemon running on the target device.

A comprehensive list of all the different commands that can be used with ADB can be found from the ADB documentation. (21.)

5 AUDIT LOGGING FUNCTIONALITY

The audit logging term is used to describe the functionality to be able to store a record of the audit events on the local device.

This kind of audit logging functionality is not part of the original implementation of the Android logging system. The original logging daemon of Linux called Auditd, which the Android's logging daemon Logd is based on, provides the functionality to store the logs to an external file.

This crucial feature has been left out of the logging daemon due to strict SELinux security policies set for Android.

Auditd is the user space component of the Linux auditing system. Kauditd in turn is the kernel space component of the auditing system. Auditd is mainly responsible for receiving the audit events and writing them to the log files. Kauditd is running as a kernel process and it is responsible for handling kernel audit events and communicating with the auditd daemon.

There are countless standards and requirements for multiple certifications that most likely require safe and uncompromised storage of the audit logs. As an example of the standards, which require this kind of functionality, are the Common Criteria requirements for audit logging presented in this thesis.

A temporary buffer solution, which is used by the logging daemon, is in some cases insufficient to satisfy these requirements.

This leaves the task of implementing the audit logging features to the device manufacturers, which are in the need of this functionality, for the sake of certification for example.

There are a few different approaches the device manufacturers can take to implement the audit logging functionality for their systems and incorporating it into the existing Android logging system.

The simplest and most logical solution would be to modify the implementation of the logging daemon and to add the functionality to write the logs to an external file. This approach however is not viable, because Android has implemented strict SELinux security policies that prevent the logging daemon from writing to files.

The next solution would be to implement a new service that will handle the file writing. This will not conflict with Android's SELinux security policies as the new service is separate from the logging daemon and therefor is not restricted by the same SELinux security policies.

Figure 31 below demonstrates an example of how this could be implemented into the logging system.

In the illusration, the logging daemons implementation is modified to send the logs that it receives onward through a socket or an interface to a new daemon. This daemon will then receive the logs and write them to an external file.

Another possible solution would be to make a daemon that reads the log contents through the socket */dev/socket/logdr/* similar as to what Logcat does. This would remove the need to modify the functionality of the logging daemon.

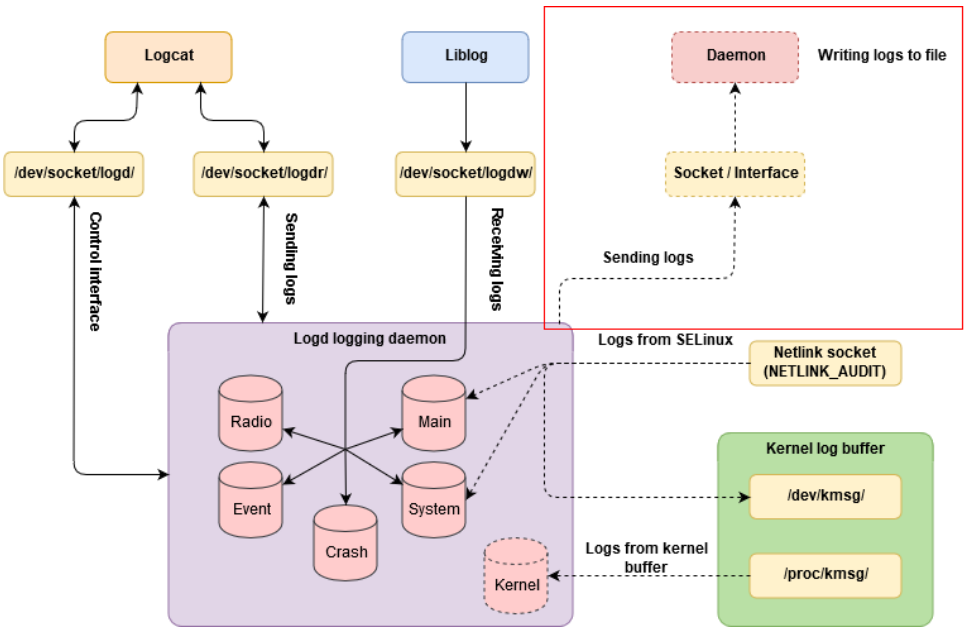

FIGURE 31. Audit logging functionality in the logging system

6 ANALYSIS OF THE SYSTEM

This chapter contains the analysis on how the system was revised to fulfill the requirements set for Security Audit by the Protection Profile that was presented earlier in chapter 3.

The chapter will cover the security requirements in the order of which they were analyzed and the possible modifications that were made to the system.

If the analysis of the security requirements revealed that the system did not fulfill the set requirements, the possible modifications that were made to the components to fulfill the requirements will be presented in some extent. In some cases, modifications were not made even though the requirements were not met. This was due the time limit of this thesis and the overall complexity of the component in question.

In most instances it is not possible to reveal all details regarding the results of the analysis and the possible modifications that were made to fulfill the security requirements as the source code and its functionality are proprietary information of the company and therefore cannot be revealed to the public.

When source code or small snippets of it have been included in this thesis, this is due to the source code being open source and therefore freely available to the public. When code snippets are included, the source of the snippets is included in the references. In cases where modifications had to be made to code that is originally open source in nature, the details of the implementations are still omitted due to the proprietary nature of the modifications.

6.1 Auditable Events - FAU_GEN.1.1

The following subsections present the individual auditable events that needed to be recorded in order to fulfill this security requirement.

6.1.1 Start-up and shutdown of the audit functions – FAU_GEN.1.1 (1)

The existing functionality of the audit functions did not include logging of its start-up and shutdown events. The existing functionality only logged audit events that it received from the Logd logging daemon.

As this was the case, the functionality to be able to log the start-up and the shutdown events of the audit functionality itself had to be implemented.

The implementation required some modifications to the existing program since the program originally only wrote the data it received from the logging daemon.

The added functionality for logging of start-up and shutdown of the audit functions itself is fairly simple. At the start of the audit logging program, the program writes the start-up event of the audit functions itself to the log records. The similar shutdown event is logged at the shutdown of the program.

However, during the shutdown of Android the audit logging program gets killed with a *SIGKILL* termination signal. Processes killed with *SIGKILL* are terminated immediately and processes are not provided any possibility to handle the incoming signal or perform any kind of cleaning up procedures.

For this reason, the logging of the shutting down event of the audit functions during the shutdown of Android is not possible.

This on the other hand is not relevant, since Android shutting down is enough of an indication that the audit logging program is also shutting down.

The appropriate log format also needed to be formed to fulfill the requirement set by *FAU_GEN.1.2*. The explanation of the security requirement itself and the log format can be found from section 6.2.

The logs of the start-up and shutdown events can be seen in figure 32.

```
25-06-2018 05:33:35:746748 I/auditlogger: AuditLoggerd shutdown
25-06-2018 05:33:35:812332 I/auditlogger: AuditLoggerd start-up
```

FIGURE 32. Start-up and shutdown events of audit functions

6.1.2 Start-up and shutdown of Android OS – FAU_GEN.1.1 (4)

A program called *init* handles the start-up of Android by starting processes necessary for operation and managing them during the runtime of Android. Init also handles the termination of the running processes during shutdown.

The *init* program is similar to the regular Linux environment *init* program, as the main purpose of the program is to handle the initialization of the system.

Android provides system properties that indicate the current status of the system. These properties do not exist in a standard Linux environment.

Init checks these system properties and performs actions that have been assigned to be performed during certain events and based on the status of the system properties.

The completion of the Android start-up is indicated by the system property *sys.boot_completed*.

Below in figure 33 is an example of *init* log messages where it processes actions assigned to the aforementioned system property. These log messages can be used to identify the start-up of Android, as is required by the security requirement.

```
26-06-2018 11:09:28:708454 I/init: processing action (sys.boot_completed=1)
26-06-2018 11:09:28:708517 I/init: processing action (sys.boot_completed=1)
26-06-2018 11:09:28:710257 I/init: processing action (sys.boot_completed=1)
26-06-2018 11:09:28:717083 I/init: processing action (sys.boot_completed=1)
```

FIGURE 33. Init processing actions for sys.boot_completed system property

An example of the contents of an *init* configuration file and an action to be performed when the *sys.boot_completed* system property is set can be seen in figure 34 below. In this example, *init* will stop the *bootchart* process that was started by *init* earlier in the boot cycle of Android.

```
704 on property:sys.boot_completed=1
705     bootchart stop
```

FIGURE 34. Example of an action in an init configuration file (23.)

As for the shutting down of Android, a new implementation was not needed to be able to record the shutdown event either.

The same *init* program also clearly indicates and sends log messages when Android is shutting down and when it is processing the actions assigned to the shutdown event of Android.

An example of the log messages *init* sends during the shutdown of Android can be seen in figure 35.

```
26-06-2018 07:02:04:006893 I/init: Clear action queue and start shutdown trigger
26-06-2018 07:02:04:006990 I/init: processing action (shutdown_done) from (<Builtin Action>:0)
26-06-2018 07:02:04:010850 I/init: Reboot start, reason: shutdown,userrequested, rebootTarget:
26-06-2018 07:02:04:012474 I/init: Shutdown timeout: 0 ms
```

FIGURE 35. Init log messages during Android shutdown

Init additionally handles actions assigned for a system property related to the shutting down of Android. The system property for monitoring shutdown requests for Android is *sys.shutdown.requested*. *Init* logs the events when it processes actions assigned to this system property. These log messages can be used as another indication of the shutdown of Android.

The resulting log messages that *init* sends during the processing of actions assigned to *sys.shutdown.requested* system property can be seen in figure 36.

```
26-06-2018 11:24:42:744648 I/init: processing action (sys.shutdown.requested=*)
```

FIGURE 36. Init processing action for sys.shutdown.requested system property

As *init* clearly logs the events when it processes actions assigned for the system properties regarding the start-up and shutdown of Android, no modifications needed to be made to fulfill the security requirement.

6.1.3 Start-up of the target product - FPT_TXT_EXT.2

With Android as the core OS of the target product, the logging of the start-up of Android is considered to be enough indication on the start-up of the target product and therefore no individual logging functionality was implemented for this security requirement.

This was rationalized with the reasoning that Android could not possibly start without the target product starting first.

The actual implementation of this feature would have been a great endeavour, since the audit logging functionality operates in Linux user-space. If logs were to be recorded from the bootloader of the target product, extensive implementation modifications would have been required to be able to achieve this.

6.1.4 All administrative actions – FAU_GEN.1.1 (3)

Administrative actions are a part of Android's device policy management system. The device policies can be changed by administrators, usually using some kind of Mobile Device Management (MDM) application.

MDM applications are used to remotely manage and monitor the device. MDM applications usually provide administrators features such as configuring password policies, inspecting the contents of the device and other administration related features.

The auditable events that needed to be generated in the event of an applicable administrative action were presented earlier in subsection 3.1.1, however this subsection will go into a bit more detail concerning the individual actions that the security requirement requires to be recorded into the audit records.

Android provides a huge number of options to configure device policies but the focus of this thesis was only on the mandatory administrative actions.

A detailed list of the policies that can be configured can be found from Android's documentation. (24.)

The actual functionality to control and change these device policies is already a part of Android. The actual logging of the events, however, was not implemented as a part of the original functionality.

The security requirement defines certain events that needed to be audited from each of the administrative actions. Below is a list of the individual events that needed to be audited to fulfill the requirement.

Configuring of password policy (7, p. 93):

- Minimum password length
- Minimum password complexity
- Maximum password lifetime

Configuring of session locking policy (7, p. 93):

- Screen lock enabled / disabled
- Screen lock timeout
- Number of authentication failures

Configuring of application installation policy (7, p. 93):

- Restriction sources of applications
- Denying installation of applications
- Application whitelisting

As the device policy manager of Android is running at the Java application level, it was possible to utilize the S*log* class of Android to handle the logging of the events. The system logging class was used since the DevicePolicyManager can be considered a system service of Android.

Specific log tags were created for each of the main administrative actions so that the individual administrative actions could easily be recognized and separated from the other administrative actions.

These created tags can be seen in figure 37 below.

```
private static String PASSWORD_POLICY_TAG = "PasswordPolicy";
private static String SESSION_LOCKING_POLICY_TAG = "SessionLockingPolicy";
private static String APPLICATION_INSTALLATION_POLICY_TAG = "ApplicationInstallationPolicy";
```

FIGURE 37. Tags for administrative action audit events

The majority of the time of adding the logging for the events went into looking through the source code of the device policy functionality, finding the functionalities corresponding with the individual auditable events and then finally adding the logging of those events.

Below in figure 38 is an example of functionality to set the minimum password length. This was one of the functionalities that had to be modified to include the logging of the event.

```
public void setPasswordMinimumLength(@NonNull ComponentName admin, int length) {
    if (mService != null) {
        try {
            mService.setPasswordMinimumLength(admin, length, mParentInstance);
        } catch (RemoteException e) {
            throw e.rethrowFromSystemServer();
        }
    }
}
```

FIGURE 38. Function for setting minimum password length (25.)

The DevicePolicyManager provides functionalities for all of the individual auditable events, except whitelisting applications. Since it is not implemented by Android, it was not possible to add logging for that event. The application whitelisting functionalities can however be provided by an MDM application for example.

Below in figure 39 is an example of the resulting log messages in the audit records. The figure contains examples of the resulting log messages for each of the administrative actions.

```
27-06-2018 11:52:08:108002 I/PasswordPolicy: SetPasswordQuality: 393216
27-06-2018 11:52:08:109631 I/PasswordPolicy: SetPasswordMinimumLength: 4
27-06-2018 11:52:08:121252 I/PasswordPolicy: SetPasswordHistoryLength: 5
27-06-2018 11:52:08:122019 I/SessionLockingPolicy: SetMaximumFailedPasswordsForWipe: 5
27-06-2018 11:52:08:122641 I/SessionLockingPolicy: SetMaximumTimeToLock: 10000
27-06-2018 11:52:08:167419 I/ApplicationInstallationPolicy: SetApplicationInstallationPolicy: no_install_unknown_sources
```

FIGURE 39. Auditable events visible in the audit records

As seen in the figure, the individual administrative action log messages can easily be identified by the tags that were shown earlier.

6.1.5 Insertion or removal of removal media – FAU_GEN.1.1 (5)

The security requirement requires logging of the insertion and removal events of external media (7, p. 20)

Android uses Volume Daemon (VOLD) to handle mounting and managing of external media such as SD cards. When an external media is inserted, *VOLD* handles the basic mounting of the partition. After that, *VOLD* hands the control over to *sdcardfs* to handle the final mounting procedures.

Sdcardfs is a stackable wrapper filesystem, which is part of the Linux kernel. *Sdcardfs* handles the final mounting of the external media by mounting the media to multiple locations in the Linux filesystem using the *sdcardfs* filesystem.

The mounts that occur when an external media is inserted into the device can be seen in figure 40 below. The *sdcardfs* filesystem type that is used for the external media is also visible in the figure.

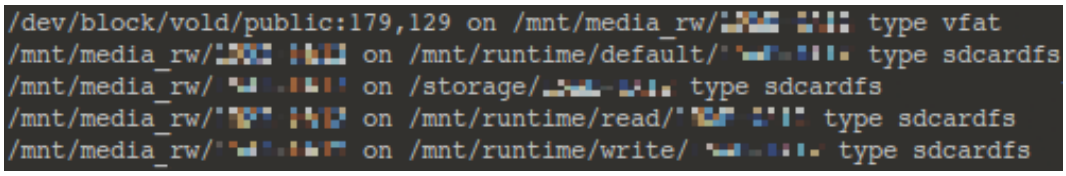

FIGURE 40. Mounting of external media

The actual logging of the mounting and unmounting events had to be implemented because a clear logging of both mounting and unmounting events did not exist in either *VOLD* or *sdcardfs*.

The implementation of the logging consisted of going through *VOLD*'s source code and determining where the mounting and unmounting of the external media is handled and adding the logging of those events.

The resulting log events that are recorded to the audit records can be seen in figure 41 below.

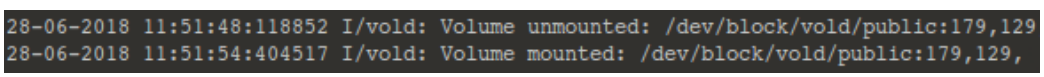

FIGURE 41. Insertion and removal of external media log messages

6.1.6 Audit records reaching (%) of full capacity (7)

The security requirement requires the recording of the event, in which the audit records reach a selected percent of the full storage capacity. (7, p. 20)

The audit functions have a limited number of audit records that it will store. This is done to prevent the audit functions from filling the whole memory of the device with the audit records. After the last audit record is full, the audit functions will delete the oldest of the audit records and create a new one in its place.

The actual logging of the audit records reaching a certain capacity was not in the existing implementation of the audit functions.

The percent of the full capacity at which the audit functions will create an audit event was chosen to be at 75%. New functionality was added to log the audit event when the threshold is reached.

The logging of the audit functions itself was implemented during the implementation phase regarding the security requirement about the start-up and shut-down auditable events of the audit functions.

As the ground work for the logging had been done earlier, the implementation of the logging for this security requirement was simple. The logging of the event was done using the previously implemented logging functionality to log events from the audit functions.

The functionality to check if the set threshold has been reached had to be implemented because it did not exist prior to this thesis. The functionality to check if the threshold has been reached checks the number of the current audit record file and compares it to the maximum number of audit records files. If the threshold has been reached, an event will be logged into the audit records.

The resulting log message in the audit records can be seen in figure 42 below.

```
29-06-2018 06:22:48:7027 I/auditlogger: Audit records reaching 75% of the full capacity
```

FIGURE 42. Audit records capacity log message

6.1.7 Initiation of application installation or update – FMT_SMF_EXT.1

The security requirement requires the logging of the initiation of application installation and update events. (7, p. 26)

The requirement is a little vague however, on whether the auditable events are expected to be logged prior to the installation or update of the application or afterward. Because of this vagueness the auditable events are included for both initiation and the completion of the actual installation or update.

Based on the previous experience on similar auditable events as for this thesis, it was known that the system already logs installation, update and uninstallation events of the applications.

These events are one of the specifically defined auditable events that require additional information to be gathered and logged into the audit records as defined by the *FAU_GEN.1.2* security requirement. The additional information for the events was the name and version of the application that is being installed or updated.

The resulting events that get logged into the audit records can be seen in figure 43 below.

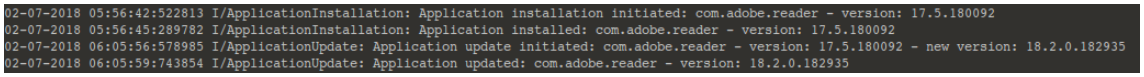

FIGURE 43. Application installation / update log messages

The additional information required by the *FAU_GEN1.2* requirement can also be seen in the figure.

6.1.8 Initiation or failure of self-test – FPT_TST_EXT.1

The security requirement requires the logging of the initiation and failure events of the cryptographic self-tests. (7, p. 23)

The self-testing of the cryptographic features is not yet implemented in the asset which this thesis was based on and therefore cannot be audited. The self-testing will be implemented in the future along with the logging functionality.

6.1.9 Modifications to audit configuration – FAU_SEL.1

The security requirement requires that all events, where modifications are made to the audit functionalities while they are operational, are logged in the audit records. (7, p. 23)

The existing implementation of the audit functions did not provide the administrator with any way to modify the configuration of the audit functions.

But as one of the security requirements required the functionality to alter the audit configuration while operational, the functionality to log the events when modifications are made to the audit configuration also needed to be implemented into the audit functions.

The required the functionality to alter the configuration of the audit functions is part of this security requirement but is covered on its own later in section 6.6.

The implementation for the logging of the events where the audit configurations are modified use the same logging system that was introduced in section 6.1.1 and was also used to fulfill the security requirement presented in section 6.1.6.

When modifications are made to the audit configurations, the audit functions will log a message to the audit records, where the administrator can see that modifications to the audit configurations have been made.

The log message contains the information that the event has occurred and what was modified. At the time of writing, the only configurable option is the configuration of the auditable events as will be presented in section 6.6.

The resulting log messages in the audit records can be seen in figure 44 below.

```
03-07-2018 12:26:31:774790 I/auditlogger: AuditLoggerd configuration modified - event types: I/
03-07-2018 12:26:33:977634 I/auditlogger: AuditLoggerd configuration modified - event types: I/W/
```

FIGURE 44. Audit configuration modification audit log messages

6.1.10 Import or destruction of key – FCS_STG_EXT.1

The security requirement requires the events, where a cryptographic key is imported or destructed, to be logged, along with the identity of the key and the identity of the requestor of the action. (7, p. 22)

Android uses Keystore to handle storing of cryptographic keys. Keystore provides features, such as key generation, importing of symmetric and asymmetric keys among other things. (26.)

Keystore is an API for an underlying Keymaster Hardware Abstraction Layer (HAL). The Keymaster HAL is in turn used to communicate with Secure Element hardware located on the device. Secure Element is a hardware based secure environment which is used to carry out cryptographic operations.

If the Secure Element does not exist on the device or it is not supported, Keymaster can fallback to a software-based mode and carry out the cryptographic operations using OpenSSL cryptographic library.

How Keymaster is utilized in the system and how it is used to interact with a Secure Element is too extensive to be covered in this thesis.

A good research paper on different secure storage solutions and how they work on Android platforms can be found from the references. (27.)

The functionality of the Keystore provided logging of the event when the keys are deleted but did not provide logging for the keys being imported.

As such, the functionality of the Keystore had to be modified to include the logging of the event when keys are being imported.

The resulting log messages from the events can be seen in figure 45 below.

```
05-07-2018 10:01:06:589190 I/keystore: ImportKey - userId: 0, appId: 10112, key: USRSKEY_AES-secretKey
05-07-2018 10:01:06:892062 I/keystore: ImportKey - userId: 0, appId: 10112, key: USRPKEY_RSAPrivateKey
05-07-2018 10:01:22:142156 I/keystore: Delete - userId: 0, appId: 10112, item: USRSKEY_AES-secretKey
05-07-2018 10:01:22:145991 I/keystore: Delete - userId: 0, appId: 10112, item: USRPKEY_RSAPrivateKey
```

FIGURE 45. Keystore import and delete log messages

6.1.11 Failure to verify integrity of stored key – FCS_STG_EXT.3

The security requirement requires the event when an integrity check fails for a stored key to be logged. The security requirement required the information on the identity of the key to be gathered from the event. (7, p. 22)

The key identities are specified by the security requirement to be Data Encryption Keys (DEK) and Key Encryption Keys (KEK).

The difference between a *DEK* and *KEK* is that the *DEK* is used to encrypt and decrypt actual data. *KEK* is instead used to encrypt and decrypt other cryptographic keys.

KEK is usually derived from the user's password. In most cases the *DEK* is encrypted using the *KEK*. This provides the possibility for the user to change passwords without having to encrypt all of the data over again. This is because the *DEK* can be re-encrypted again with a new derived *KEK* instead of creating a new *DEK* and re-encrypting all of the data with the new *DEK*.

After inspecting and studying the functionality of the Keymaster HAL and the underlying Secure Element implementation, it became clear that the system does not provide complete assurance that the integrity failure is due to just the keyblob integrity verification failure and not due to other related errors that the Secure Element reports as an integrity failure.

Keymaster will return an error code in case an operation fails in the Secure Element. The integrity failure error code is used for various reasons related to the integrity failure in addition to the actual integrity verification failure of the keyblob. Despite this, the integrity failure error code is used to determine the event in which the integrity verification fails.

In addition, the system does not determine whether a keyblob that is being verified contains a *DEK* or a *KEK* and as such their identities cannot be known based solely on the keyblob itself. As the identity of the key completely depends on the use case of the key it is sufficient to audit the integrity check of the encrypted keyblob itself and disregarding the identity of the key.

In *VOLD*'s case it is known that the key that is being used for encrypting and decrypting of data on the */data/* partition is a *DEK*, as the key is used for encrypting and decrypting data instead of other cryptographic keys.

VOLD handles the creation of the *KEK*, which is derived from the user's password. After the *KEK* has been created *VOLD* passes the created key forward to TrustZone. TrustZone is responsible for handling and decrypting the *DEK* using the created *KEK*. The decrypted *DEK* is then used to encrypt and decrypt data on the */data/* partition.

TrustZone is a proprietary Trusted Execution Environment by ARM and as it is a proprietary component, the logging of the integrity verification of the *DEK* is not possible.

As for the Android Keystore, the keys that are being stored can either be *DEK's or KEK's* depending on their use. As mentioned earlier, in this case it is sufficient to record the verification of the keyblob integrity without identifying the stored key as either a *DEK* or a *KEK*.

Android Keystore provides functionality to double encrypt the cryptographic keys that are stored. In these cases, Keystore encrypts the cryptographic keys already encrypted by Keymaster.

The resulting double encrypted keyblob is illustrated in figure 46 below. This double encryption feature provided by Keystore is completely optional. Whether the cryptographic keys are double encrypted or not, the actual cryptographic key within the keyblob will never be exposed outside of Keymaster.

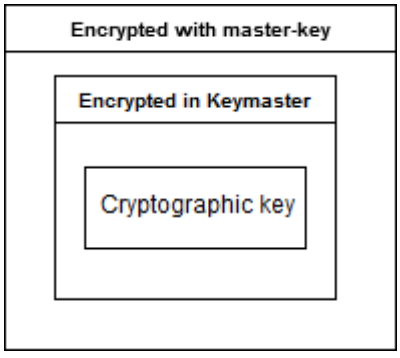

FIGURE 46. Double encrypted keyblob

Keystore uses a *master-key* for encrypting and decrypting the stored cryptographic keys. The *master-key* is a randomly generated key, which is stored in an encrypted keyblob on the device. The key is encrypted using a *KEK*, which is derived from the user's password. The encryption of the *master-key* is illustrated in figure 47 below.

The *master-key* is decrypted and loaded into memory when the user unlocks the device with a password. When the user locks the device the decrypted master-key is cleared from the memory.

If the cryptographic key is only encrypted by Keymaster, the steps to encrypt the cryptographic key in the Keystore using the *master-key* are not taken.

The removal of the encryption placed on the keyblob by the Keystore already contains logging of the integrity failure that might occur during the decryption. The integrity failure of the keyblob during the Keymaster operation however did not exist and had to be implemented.

Keymaster will return an error code if the operation performed with the keyblob fails due to the integrity of the keyblob. This error code is checked in Keystore and the event is logged in the case that the failure was due to the integrity failure of the keyblob.

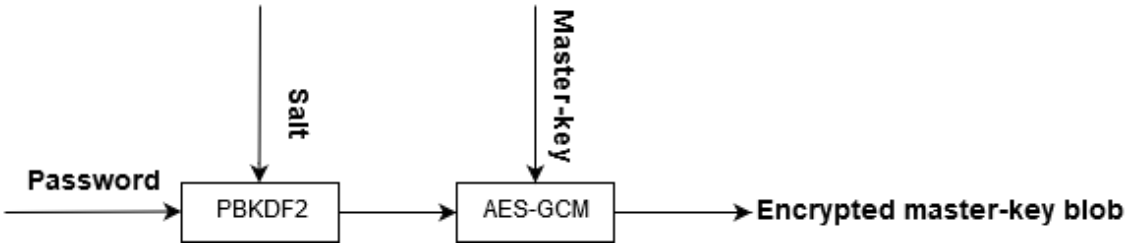

FIGURE 47. Creation of the encrypted master-key blob

PBKDF2, as seen in the above figure, is an algorithm that is used for deriving cryptographic keys out of passwords. The algorithm uses key stretching to derive stronger and longer cryptographic keys out of weak and short passwords.

AES-GCM is a symmetric key encryption algorithm that is used to encrypt the *master-key* with a key derived using the above-mentioned method.

6.1.12 Failure to encrypt/decrypt data – FDP_DAR_EXT.2

The security requirement requires an audit event to be logged in the case that encryption or decryption of sensitive data fails. (7, p. 22)

The encryption of sensitive data is separate from the normal full disk encryption and adds an additional layer of security for the data that is stored as sensitive data. The sensitive data is stored encrypted while the device is locked. When the device is unlocked by the user, the sensitive data is made available for use by providing the possibility to decrypt the data.

The encryption and decryption of the sensitive data is done using Elliptic curve cryptography. Elliptic curve cryptography is asymmetric, similar to the more familiar RSA cryptography, where the cryptography is done using a public-private key pair. This is different from symmetric cryptography, where the encryption and decryption are done using the same secret key. One of the most known symmetric cryptography algorithms is AES, which was briefly mentioned before in the previous subsection.

The sensitive data uses asymmetric encryption instead of symmetric encryption because the requirement states that the device should be able to encrypt data while the device is locked. The device is able to encrypt the data while the device is locked by using asymmetric cryptography. (7, p. 72-74)

The sensitive data itself is stored in a specific directory on the device. When an application or a user writes data to this directory, the written sensitive data gets automatically encrypted. In addition to the encryption of the sensitive data itself, the filenames and the file paths of the sensitive data are also encrypted to provide more security and confidentiality for the sensitive data.

The logging of the events when the sensitive data encryption or decryption fails is already implemented in the sensitive data functionality and as such, did not require any modifications to be made in order to fulfill the security requirement.

6.1.13 Addition or removal of certificate from Trust Anchor Database – FDP_STG_EXT.1

The security requirement requires logging of addition and removal events for certificates in the Trust Anchor Database. The subject name of the certificate needs to be recorded in the log message. (7, p. 22)

The Trust Anchor Database on Android is currently provided by a Java Security Provider called Conscrypt.

Conscrypt offers cryptographical primitives and Transport Layer Security (TLS) for Java applications on Android platforms. (28.)

More information on Conscrypt and the features it provides can be found from the Conscrypt GitHub page, which is listed in the references (28.)

Trusted certificates that come with the Android system are stored in the following location: */etc/security/*.

The amount of certificates that are currently shipped with the Android system is a little under 200 certificates. The amount of the certificates can vary between different Android versions and device manufacturers.

The certificates that have been added by the user or an application are stored in */data/misc/user/n/cacerts-added/*, where n indicates the id number of the user.

The Trust Anchor Database provided by Conscrypt did not contain logging of the addition or removal events for the certificates. This required the logging of the required events to be added to the implementation.

Below in figure 48 is an example of the resulting log messages for the addition and removal events of certificates in the Trust Anchor Database.

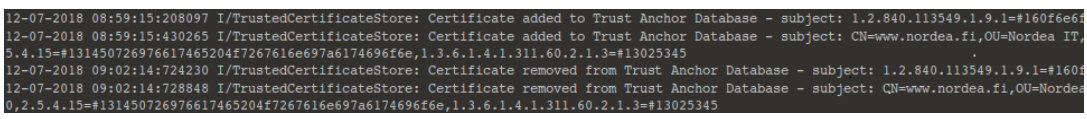

FIGURE 48. Trust Anchor Database addition and removal log events

6.1.14 Failure to validate X.509v3 certificate – FIA_X509_EXT.1

The security requirement requires logging of the event, in which the validation of an X.509v3 certificate fails. (7, p. 23)

Android provides centralized trusted certificate storage for applications to be utilized. The usage of this certificate storage, however, is not enforced, and applications can choose to implement their own certificate storages and own certificate validation implementations.

Android provides standard TLS libraries that provide the functionality to validate the server's certificates against the trusted certificate storage on the device. The application, however, might not use the standard TLS libraries and choose to implement its own certificate validation implementations or use other third-party TLS libraries. In addition to the standard TLS Android libraries, some of the well-known and more used third-party TLS libraries are libraries, such as OpenSSL and GnuTLS.

Research has been made on the usage of various TLS libraries on Android applications. It shows that 84% of the tested applications utilize the standard Android TLS libraries. The rest are spread between different OpenSSL versions and other unclassified third-party TLS libraries. (29.)

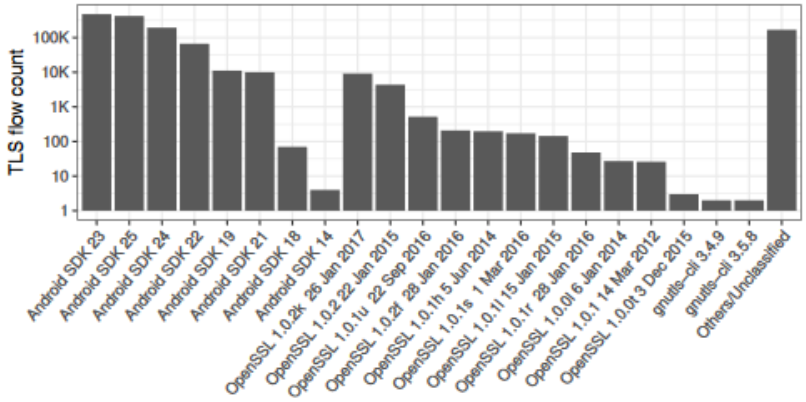

FIGURE 49. TLS library TLS flows (29, Figure 2.)

As this is the case, it is impossible to provide audit logging of the events, in which the certificate validation fails, and the verification is done in an application specific implementation or in a third-party library.

6.2 Audit Records Format - FAU_GEN.1.2

The security requirement defines strict guidelines on the format in which the log messages are to be stored in the audit records, including what information is required to be recorded in the log messages.

The information that needs to be recorded in order to fulfill the requirement:

- Date and time of the event
- Type of the event
- Subject identity
- Outcome of the event
- Additional information in specific events

Overview of the format of the log record can be seen in figure 50.

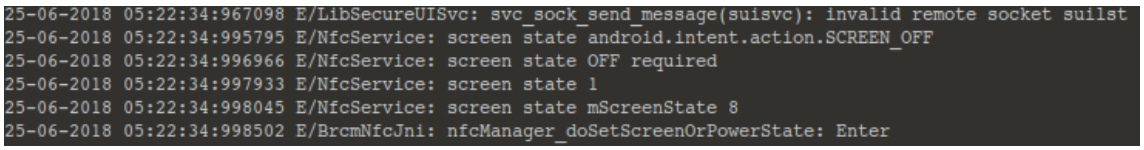

FIGURE 50. Overview of the log record format

A more detailed explanation of an individual log message can be seen in figure 51.

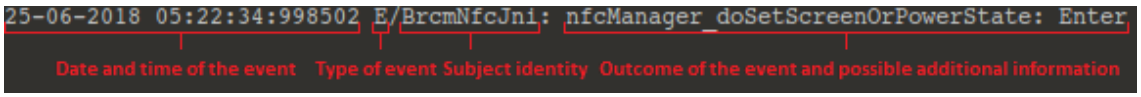

FIGURE 51. Explanation of the log message information

The date and time of the event information is first in the log message. The date is recorded in a *dd-mm-yyyy* format and the time is recorded in a 24-hour time system.

The type of the event is expressed with the severity level of the log message. In the case in the above figure, the severity level has been set to Error. The severity levels that are recorded into the audit records: *Info, Warning and Error.*

The subject identity is expressed by an identifier, which is recorded along with the log message. This is sufficient to identify the source of the log message.

The last piece of the log message is the actual outcome of the event itself and possible additional information, depending on the event requirements.

The outcome of the event is not specified as to what it should be by the requirement. Depending on the event, the outcome is expected to be either a success or a failure, or if the event cannot be evaluated to be a success or a failure, just a recording of the event that has occurred.

The expression of the success or failure of an event is not defined by the requirement either. Depending on the event and the source of the event, the outcome can sometimes be a written explanation of the success or failure or simply a number representing a success or failure state.

The additional information is required by the requirement in certain events but it is not limited to those specific events. Other events can also record additional information if they wish to do so. This is useful as the more information that is gathered from the event, the more complete picture the administrator gets of the whole event and what has transpired.

The additional information has specific requirements on what needs to be recorded in the case that the event is one of the specifically defined events. If the event is not one of those specific events, the additional information to be recorded can be selected at will.

The requirement however does not state the actual format the additional information needs to be presented in. As there are no requirements set for the format, the format can be chosen based on the event and the information that needs to be recorded.

Below in figure 52 is an example format for the additional information.

```
userId: 0, appId: 10112, key: USRSKEY_AES-secretKey
userId: 0, appId: 10112, key: USRPKEY_RSAPrivateKey
```

FIGURE 52. Example format for the additional information

6.3 Audit Storage Protection – FAU_STG.1

The security requirement requires the protection of the stored audit records from unauthorized deletion and modification. (7, p. 27)

These requirements are met as the Linux operating system provides an extensive access control system for individual files and directories.

When the audit records are created, the permissions for the files are set as such that only the program as the owner of the file and the *root* user are allowed to modify the contents of the audit records.

The permissions are indicated in three pairs of three letters: *RWX*, where they indicate the read, write and execute permissions respectively.

The first letter of the permissions string can be used to indicate the type of the file. There are multiple of different letters indicating different types of files. If the file is a regular file, the letter is replaced with a dash.

The first pair of three letters is used to indicate the permissions that the owner of the file has over the file. As can be seen in figure 53 below, only the owner of the file has been granted read and write permissions for the file. No execution permissions have been granted since the file is a normal text file and cannot be executed.

The next pair after the owner's permissions are the permissions that a user group has over the file. The last pair are the permissions that other users have over the file.

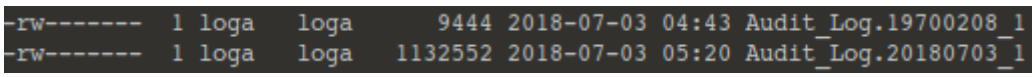

FIGURE 53. Audit records access permissions

The audit records are also protected by strict SELinux policies.

By properly utilizing the access control systems of Linux and SELinux, the access control systems take care of the protection of the audit records and prevent unauthorized modifications of the audit records.

6.4 Prevention of Audit Data Loss – FAU_STG.4

The security requirement requires that the audit functionalities overwrite the oldest audit record in the case that the maximum capacity is reached. (7, p. 27)

As was previously mentioned in subsection 6.1.6, the audit records have a set maximum capacity to prevent the audit records from filling up the entire internal memory of the device.

As was also briefly mentioned in subsection 6.1.6, the current implementation of the audit functions provides functionality to remove the oldest audit record when the maximum capacity is reached. After the oldest record has been deleted, a new audit record will be created in its place.

This kind of functionality enables the audit functions to keep operating practically infinitely, only replacing the oldest audit record when necessary.

As the implementation is already in place, fulfilling the security requirement, no changes needed to be made to the functionality.

6.5 Audit Review – FAU_SAR.1

The security requirement requires that the audit functions must be able to provide the capability to read the record contents and provide them in a manner suitable for the administrator to interpret. (7, p. 145)

The requirement is first of the requirements that were decided to be included, even though these are completely optional at the time.

The current implementation of the audit functions provide functionality to transfer the audit records via an interface through which the audit records can be sent onward to, for example, an MDM server, where the audit records can be viewed and interpreted by the administrator.

The audit events in the audit records are also recorded in a simple format to help the administrator interpret the contents of the audit records. This can be seen in the figures shown in the previously analyzed auditable events subsections.

6.6 Selective Audit – FAU_SEL.1

The security requirement requires that the audit functions must be able to provide the administrator a possibility to select a set of events to be audited from all of the auditable events. (7, p. 145)

This is the second and final of the optional security requirements that were decided to be included.

The requirement provides a selection of parameters based on which the auditable events can be filtered. The list of the parameters that could be selected was shown earlier in section 3.5.

From the list of possible parameters, the event type was chosen as the only configurable parameter. This was because the event type was thought of being the most descriptive of the parameters and to simplify the implementation of the selection functionality.

This kind of selection was not originally implemented in the audit functions and it needed to be implemented in order to fulfill the security requirement.

The implemented functionality to select the events to be audited based on the type of the event is provided via the same interface as the one that was mentioned in the previous subsection, which presented the analysis of the audit review security requirement.

The administrator can configure the event types that the audit functionalities record through the aforementioned interface. The audit functions will then compare the audit messages it receives and stores them in the audit records depending if the event type matches the ones defined by the administrator.

The event types were defined by the *FAU_GEN.1.2* security requirement, which was presented earlier in subsection 3.1.2.

As the administrator is provided the functionality to select the events to be audited based on the event type, the security requirement is fulfilled.

7 CONCLUSION

The main aim of this thesis was to familiarize myself with the requirements set for Security Audit by the Common Criteria Mobile Device Fundamentals Protection Profile and to analyze the current situation of the system and how it in its current state fulfilled these requirements. In addition to this, the secondary aim was to add the functionality to fulfill the security requirements where the system did not meet the requirements.

To achieve these aims, I had to familiarize myself with the logging system of Android, which has been implemented for logging messages. The whole logging system was presented in detail, simplifying certain aspects of it when necessary to help the reader get a better overall understanding of the system.

The final phase of this thesis was the actual analysis of the system itself. To my surprise quite a few modifications had to be made to the system to be able to fulfill the security requirements. A majority of these modifications were made to Android components and only a small portion to the proprietary components of the company.

It is surprising that Android implementations have not been made more compatible with some of these more common certification requirements, such as the ones presented in this thesis. Instead, Android leaves the modification up to the device manufacturers. This is, of course, not always the case as Android also provides many features to comply with certain, usually larger scale requirements.

Nevertheless, the aims for this thesis were met, the system was analyzed and modifications to comply with the security requirements were made where it was convenient and possible without larger modifications of the components.

In the cases where the system did not meet the requirements and the modifications could not be made at the current time in order to conform to the requirements, they were highlighted and will be resolved as necessary at some time in the future outside of the scope of this thesis.

REFERENCES

1. Bittium Corporation. About Bittium. Date of retrieval 13.7.2018 https://www.bittium.com/about_bittium

2. Common Criteria. Common Criteria. Date of retrieval 2.6.2018 https://www.commoncriteriaportal.org/

3. Common Criteria. About Common Criteria. Date of retrieval 4.6.2018 https://www.commoncriteriaportal.org/ccra/

4. Common Criteria 2014. Arrangement on the Recognition of Common Criteria Certificates In the field of Information Technology Security. Date of retrieval 4.6.2018 https://www.commoncriteriaportal.org/files/CCRA%20-%20July%202,%202014%20-%20Ratified%20September%208%202014.pdf

5. Common Criteria. Members of the CCRA. Date of retrieval 4.6.2018 https://www.commoncriteriaportal.org/ccra/members/

6. Common Criteria. Protection Profiles. Date of retrieval 4.6.2018 https://www.commoncriteriaportal.org/pps/

7. Common Criteria 2017. Protection Profile for Mobile Device Fundamentals, version 3.1, National Information Assurance Partnership. Date of retrieval 4.6.2018. https://www.commoncriteriaportal.org/files/ppfiles/pp_md_v3.1.pdf

8. Common Criteria. Certified Products. Date of retrieval 5.6.2018 https://www.commoncriteriaportal.org/products/

9. Common Criteria 2017. Common Criteria for Information Technology Security Evaluation, Part 3: Security assurance components, Version 3.1, Revision 5. Date of retrieval 5.6.2018
https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R5.pdf

10. Android Source. ART and Dalvik. Date of retrieval 12.6.2018
https://source.android.com/devices/tech/dalvik/

11. Oracle. Java Platform Standard Ed. 7, Class Logger. Date of retrieval 17.7.2018
https://docs.oracle.com/javase/7/docs/api/java/util/logging/Logger.html

12. Android Developers. Reference, EventLog. Date of retrieval 8.6.2018
https://developer.android.com/reference/android/util/EventLog

13. GoogleGit, Android Source Code, Log.java. Date of retrieval 8.6.2018
https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/util/Log.java

14. Embedded Linux Wiki, Android Logging System. Date of retrieval 8.6.2018
https://elinux.org/Android_Logging_System

15. Android Developers. Reference, Log. Date of retrieval 8.6.2018
https://developer.android.com/reference/android/util/Log

16. Android Developers. Android NDK Native APIs. Date of retrieval 11.6.2018
https://developer.android.com/ndk/guides/stable_apis

17. GoogleGit. Android Source Code, log_main.h. Date of retrieval 11.6.2018
https://android.googlesource.com/platform/system/core/+/master/liblog/include/log/log_main.h

18. Android Source. Security, Security-Enhanced Linux in Android. Date of retrieval 14.6.2018
https://source.android.com/security/selinux/concepts

19. Linux man-pages. Netlink. Date of retrieval 14.6.2018
http://man7.org/linux/man-pages/man7/netlink.7.html

20. Android Developers. Reference, Logcat command-line tool. Date of retrieval 8.6.2018
https://developer.android.com/studio/command-line/logcat

21. Android Developers. Reference, Android Debug Bridge (adb). Date of retrieval 8.6.2018
https://developer.android.com/studio/command-line/adb

22. Kobayashi, Tetsuyuki 2012. ADB (Android Debug Bridge): How it works? Date of retrieval 15.6.2018
https://www.slideshare.net/tetsu.koba/adbandroid-debug-bridge-how-it-works

23. GoogleGit. Android Source Code, init.rc. Date of retrieval 26.6.2018
https://android.googlesource.com/platform/system/core/+/master/rootdir/init.rc

24. Android Developers. DevicePolicyManager. Date of retrieval 27.6.2018
https://developer.android.com/reference/android/app/admin/DevicePolicyManager

25. GoogleGit. Android Source Code, DevicePolicyManager.java. Date of retrieval 27.6.2018
https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/app/admin/DevicePolicyManager.java

26. Android Source. Security, Keystore features. Date of retrieval 4.7.2018 https://source.android.com/security/keystore/features

27. Coojimans, Tim – de Ruiter, Joeri – Poll, Erik 2014. Analysis of Secure Key Storage Solutions on Android. Date of retrieval 5.7.2018 http://www.cs.ru.nl/~joeri/papers/spsm14.pdf

28. Github. Conscrypt. Date of retrieval 12.7.2018 https://github.com/google/conscrypt

29. Razaghpanah, Abbas – Akhavan Niaki, Arian - Vallina-Rodriguez, Narseo - Sundaresan, Srikanth – Amann, Johanna – Gill, Phillipa 2017. Studying TLS Usage in Android Apps. Date of retrieval 16.7.2018 http://www.icir.org/johanna/papers/conext17android.pdf