Secure Storage Guideline for Android Devices

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Cyber Security

at the

University of Waikato

by

Sjoerd de Feijter

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

University of Waikato

2016

Abstract

The sudden surge in the mobile market has given developers the ability to create applications that reach millions of potential users. Many applications store sensitive data, which makes them potential targets for malicious individuals. A large majority of developers do not have a background in security making it difficult for them to securely store this data. This research addresses the problem by acting as a guide for developers, allowing them to create attack resistant applications. This is achieved by analysing techniques used by attackers to compromise applications, and illustrating countermeasures, which will give a greater insight into the concept of secure storage. As opposed to many other resources, the research presented is a consolidated resource of security techniques and gives numerous implementation examples, making it straightforward for developers to use.

Acknowledgements

I would like to thank my supervisors Dr Ryan Ko, Dr Sivadon Chaisiri and the other members of the Cyber Security Researchers of Waikato lab. As well as my supervisor Mark Dunning from the Gallagher Group, for their continued support throughout the research.

I would like to express my gratitude towards Gallagher Group for the Sir William Gallagher Cyber Security Scholarship and The University of Waikato for the Masters Research Scholarship. The scholarships provided essential financial assistance to help support the research.

Lastly, I would like to thank my family and friends for their continued support throughout the year.

Contents

1 Introduction **1**

1.1 Scope . . . 2

1.2 Thesis layout . . . 3

2 Background **4**

2.1 Data Storage . . . 4

2.1.1 Hardware based architecture . . . 5

2.1.1.1 Secure Element . . . 6

2.1.1.2 Trusted Execution Environment . . . 9

2.1.2 Cloud based architecture . . . 11

2.1.3 Software based architecture . . . 13

2.1.4 Summary . . . 13

2.2 Android . . . 13

2.2.1 Android applications . . . 15

2.2.2 Android keystore system . . . 15

2.3 Security Guidelines and Standards . . . 16

3 Attack Phases **19**

3.1 Phase I: Extraction . . . 20

3.1.1 Rooting Devices . . . 20

3.1.2 Using Emulators . . . 21

3.1.3 Android Debug Bridge . . . 22

3.2 Phase II: Analysis . . . 22

3.2.1 Static Analysis . . . 22

3.2.2 Dynamic Analysis . . . 24

3.2.2.1 Debugging . . . 24

3.2.2.2 Memory Dump . . . 27

3.3 Phase III: Exploitation . . . 30

4 Security Techniques **32**

4.1 Obfuscation . . . 32

4.1.1 Description . . . 32

4.1.1.1 Lexical Transformations 33
4.1.1.2 Control Transformations 35
4.1.1.3 Data Transformations 35
4.1.1.4 Anti-Decompilation 37
4.1.2 Implementation 37
4.1.3 Analysis 38
4.2 Tamper Detection 40
4.2.1 Certificate Checking 40
4.2.1.1 Description 40
4.2.1.2 Implementation 41
4.2.1.3 Analysis 41
4.2.2 Installer Verification 42
4.2.2.1 Description 42
4.2.2.2 Implementation 42
4.2.2.3 Analysis 43
4.2.3 Checksumming 43
4.2.3.1 Description 43
4.2.3.2 Implementation 44
4.2.3.3 Analysis 45
4.3 Root Detection 47
4.3.1 Kernel modification 47
4.3.1.1 Description 47
4.3.1.2 Implementation 47
4.3.1.3 Analysis 48
4.3.2 Su binary 48
4.3.2.1 Description 48
4.3.2.2 Implementation 48
4.3.2.3 Analysis 49
4.3.3 Command execution 49
4.3.3.1 Description 49
4.3.3.2 Implementation 49
4.3.3.3 Analysis 49
4.3.4 Rooting Applications 50
4.3.4.1 Description 50
4.3.4.2 Implementation 50
4.3.4.3 Analysis 50
4.4 Debugging Detection 51
4.4.1 Execution Timing 51
4.4.1.1 Description 51
4.4.1.2 Implementation 51

4.4.1.3 Analysis 52
4.4.2 Debugger Connection 52
4.4.2.1 Description 52
4.4.2.2 Implementation 52
4.4.2.3 Analysis 53
4.4.3 Debugging Flag 53
4.4.3.1 Description 53
4.4.3.2 Implementation 53
4.4.3.3 Analysis 53
4.5 Emulator Detection 54
4.5.1 Description 54
4.5.2 Implementation 54
4.5.3 Analysis 54
4.6 Sanitization 55
4.6.1 Description 55
4.6.2 Implementation 55
4.6.3 Analysis 56

5 Recommendations **57**

6 Conclusion **61**
6.1 Summary 61
6.2 Contributions 61
6.3 Future Work 62

References **63**

Appendices **69**

A Code Listings **70**

List of Figures

2.1 Hardware based architecture 6

2.2 Embedded SEs 7

2.3 Trusted Execution Environment 9

2.4 Comparison of security offered by software, TEEs and SEs . . 10

2.5 Running TEE Driver 11

2.6 Cloud based architecture. 11

2.7 Android Software Stack 14

3.1 Attack Life Cycle 19

3.2 Decompiled java code 23

3.3 Modified APK using APK Studio 25

3.4 Debuggable android application 26

3.5 Attaching to a debuggable application. 26

3.6 Application inspection 27

3.7 Memory dump 29

3.8 Successful tampering 31

4.1 Obfuscation lexical transformation 34

4.2 Obfuscation control transformation 36

4.3 Enabling ProGuard obfuscation 38

4.4 Code obfuscated with ProGuard 39

4.5 Specify installer package name with ADB 43

4.6 Storing checksum signature problem 45

4.7 Graph showing performance impact of Tamper Detection . . . 46

Listings

A.1 Class with hardcoded values 70
A.2 Class without a hardcoded value 71
A.3 Java code used for memory dump 72
A.4 C code used for memory dump 73
A.5 Java code tamper demonstration 76
A.6 Smali code tamper demonstration 77
A.7 Opaque predicates 79
A.8 String encoding 80
A.9 Retrieving public key 81
A.10 Verifying public key 81
A.11 Retrieving installer name 81
A.12 Verifying installer name 81
A.13 Extracting .dex file from APK 82
A.14 Calculating checksum 83
A.15 Root detection via Kernel modification 84
A.16 Root detection via su binary 84
A.17 Root detection via su execution 85
A.18 Root detection via applications 86
A.19 Debug detection via execution time 87
A.20 Debug detection via debugger connection 88
A.21 Debug detection via debugger attach 88
A.22 Debug detection via debugging flag 89
A.23 Emulator detection 90
A.24 Sanatization 91

List of Tables

2.1 OWASP Mobile Top 10 17

4.1 Android obfuscation tools. 38

5.1 Security technique recommendations 60

Chapter 1

Introduction

Mobile device usage has increased immensely over the last few years. Applications running on these devices have become part of everyday life for many users, and often rely on them. There are applications for almost any purpose, for instance, banking, email, access control, payments and social media. Many of these applications store sensitive information, such as user names, passwords and tokens in order to authenticate with infrastructure. This means that developers need to have the expertise to securely store this information inside applications, which is often not the case.

The amount of resources and development that goes into an application can vary. Some applications are created by a team of developers with access to huge amounts of resources, other applications are created by a single individual with little to no resources. Whether a whole team, or a single developer is used, sensitive data should be stored in a secure way. This can prove difficult if no security expertise are available, and leads to easy exposure of sensitive data. Not many resources are available that facilitate developers in the creation of secure applications. Resources that are available, are often incomplete and do not provide enough implementation details. This is not ideal and needs to be addressed.

Applications need to be created with security in mind. There are many techniques that can be used to aid in the protection of applications, however they are often not incorporated by developers, most likely due to a lack of available resources. This research will provide developers a resource to refer to, in order to create more secure applications, whose sensitive data is less likely to be compromised. Topics covered to achieve this are:

- The capabilities attackers have when attacking applications.
- The security techniques that can be put in place to combat attackers.

Covering these topics gives developers an insight into what is required to create attack resistant applications, and gives implementation details to make it easier for developers to incorporate the presented security techniques.

1.1 Scope

The concepts covered by this research will apply to most modern mobile Operating Systems (OS). However it is not feasible to target all of them, as specific implementation are provided and platforms differ. Therefore the most popular mobile OS will be the target of this investigation, which is the Android platform with the majority of the market share [1]. As a result, other platforms such as Windows phone and IOS will not be specifically covered by this research.

Mobile development companies often hire security experts in order to add security features into their applications, this requires a large amount of resources. Many Android developers do not have the resources for this, which means that security is often overlooked. Developers that do not have a large amount of resources available, with little to no security background will benefit most from this research. Therefore these individuals are the target audience.

1.2 Thesis layout

Chapter 2 will examine the Android platform in relation to data storage and code execution, along with related work.
Chapter 3 explores how attackers go about compromising applications and extracting data.
Chapter 4 examines security techniques that developers can use to protect their applications.
Chapter 5 gives recommendations on what security techniques developers should use.
Chapter 6 provides a quick summary of the thesis, covers possible future work in the area and highlights the contributions made by this research.

Chapter 2

Background

2.1 Data Storage

Sensitive data is used and stored by many applications for authentication purposes. Storing this data on a device can be problematic as the information can be stolen by malicious individuals. This could lead to user accounts or infrastructure being compromised. A solution to this problem would be to not store the data on the device at all. Each time authentication occurs, the user themselves supplies the authentication data. This would ensure that no sensitive data could be leaked. However, this is often not feasible. For example, it will be very difficult for users to remember access tokens[1]. Therefore, sensitive data often requires to be stored on the device itself.

There are many ways to store sensitive data on an Android device [2]. The most common approach is to store it inside of the application data directory. This directory is protected by the application sandbox, as mentioned in Section 2.2. On top of that, encryption can be used to further protect the data at rest. The Android Keystore, outlined in Section 2.2.2 can be used for this purpose.

[1]Access tokens are objects that can be used for authentication and are often generated using user credentials

If the data is encrypted at rest, it becomes difficult for attackers to extract or use the sensitive information without possession of the decryption key. There is a fundamental problem, this approach only protects the data at rest. When the application is running and requires the use of the data, it is decrypted and used by the application, exposing it to attackers.

The concept of secure data storage is not only about the data itself, it is about securing the whole application. If data is exposed in the application during any part of execution, attackers have the ability to obtain the information. Cryptography is sufficient at protecting data at rest [3], however protecting the data during execution is not so simple. In this section three different architectures are discussed that can be used to execute code on mobile devices. Exploring these environments gives developers a better understanding of available options when it comes to creating secure applications.

2.1.1 Hardware based architecture

In a hardware based architecture, data storage or processing can be performed through hardware present on the mobile device. The hardware provides a secure environment where applications and data cannot be inspected from the outside world. Trusted applications[2] are installed on the hardware, and provide certain services [4] [5] [6]. For example, cryptographic operations, data storage and hashing applications often come pre-installed. These applications can perform security sensitive operations without exposing them to the underlying OS. Figure 2.1 illustrates a hardware based architecture. The OS provides the means for applications to communicate with the trusted applications. Allowing developers to use the services provided by the trusted applications.

[2]Trusted applications are applications that have been designed and authorised to run on the target hardware environment, ensuring that applications running on the hardware do not compromise one another

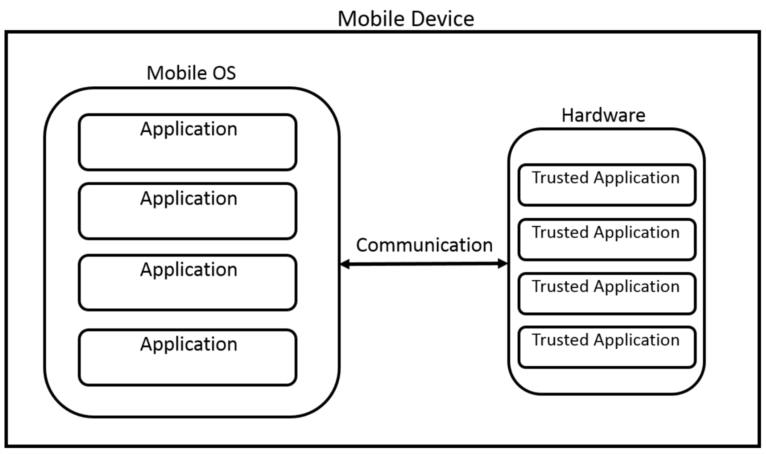


Figure 2.1: Hardware based architecture

When it comes to hardware based architectures there are two main options available, the Secure Element (SE) and the Trusted Execution Environment (TEE).

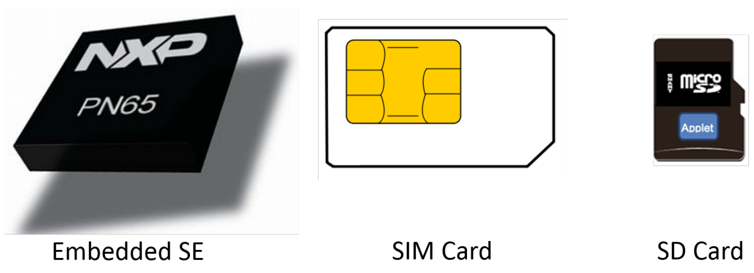
2.1.1.1 Secure Element

A Secure Element (SE) consists of a tamper-proof chip, similar to that of chips inside banking cards [4]. The tamper proofing technology present in these chips allows for secure computation in insecure environments. The chip is capable of running trusted applications that are installed on the chip by the manufacturers. These trusted applications provide services that can be communicated with via a serial interface. Application Protocol Data Units (APDUs) [7] are used to communicate with the trusted applications.

In relation to the mobile platform, a Subscriber Identity Module or SIM card is a type of removable SE [6]. Applications installed on modern day SIMs come in the form of Java Card [8] applets[3]. Common applets installed on SIMs are responsible for storing identity and network information along with performing the authenticating to mobile networks. SIM cards are not the only

[3]Java Card application, also known as applets, are Java applications designed for embedded environments.

SEs that can be present on mobile devices. Other hardware modules such as Secure Digital (SD) cards or embedded SEs are also used. As shown in Figure 2.2.



source:
http://cdn.iphonehacks.com/wp-content/uploads/2014/08/nxp_nfc_chip.jpg,
https://nesmobile.com/images/Israel_sim_card.png,
https://www.sdcard.org/developers/overview/ASSD/smartsd/img/smartSD-mobile.png

Figure 2.2: Embedded SE, SIM Card and SD Card [9] used as SEs.

Developers wanting to create their own trusted applications to be used on SEs face many problems, making it not feasible to use under normal circumstances [6]. Problems and challenges are listed below:

- **SE communication.** In order to communicate with SIMs or other SE there is often an extra library required on the mobile device, such as the Open Mobile API [10] by SIMalliance [11]. The stock Android release does not come with this library, which makes it difficult to use with devices that come with this version of Android. There are two ways for the library to be present on the device, device manufacturer can bundle it with the OS, which is the case for many Samsung devices, or it can be flashed to the device. SEEK for Android [12] provides the means to flash the API to the device, but it is not practical to do this for every device that an application needs to run on.

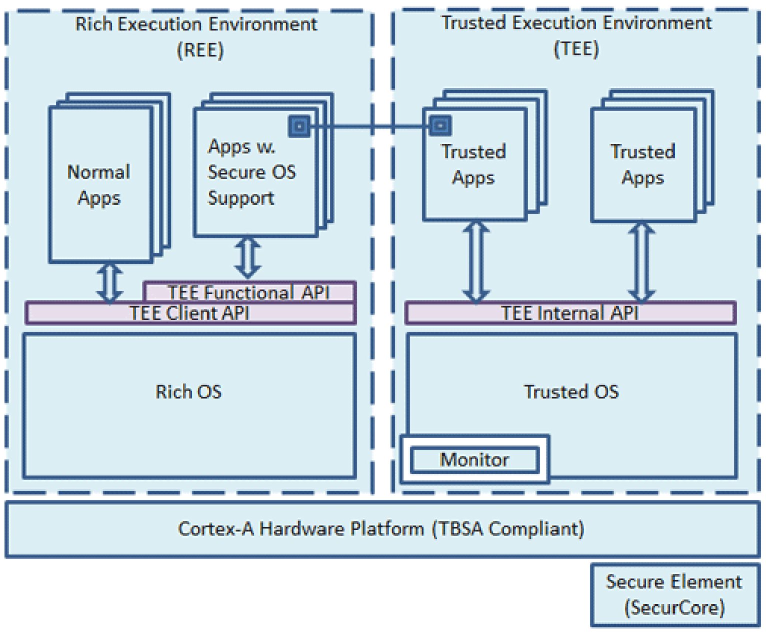
- **Installing SE applications.** SEs, including SIMs come pre-loaded with trusted applications, such as applications that can perform cryptographic operations. In order to achieve secure data processing, developers need to be able to create a trusted application that can then be installed on the SE. The main problem is installing these applications on the hardware. Production SIMs require specific management keys to authenticate with the SIM card manager, through which applications are installed [6]. Developers are not given access to these keys since they would be able to install any application on the SIMs, including malicious ones. This means developers have to cooperate with telecommunications companies that govern the SIM cards, in order to install the applications. If wanting to support customers from different providers, developers would have to cooperate with each of the telecommunications companies involved. This is not realistic approach for many developers, as large amounts of resources are required.

Embedded SEs are similar to standard SEs but are installed during device manufacturing and cannot be removed from the device [13]. They are often bundled with a Near Field Communication (NFC) chips and provide similar functionality to that of SIM cards. As with SIM cards it is difficult to communicate with an embedded SE [14]. Only whitelisted applications are able to communicate with the SE, this is problematic since only a few core OS level applications are whitelisted. In order to communicate with the SE, it would be necessary to modify the OS which is not applicable for commercial applications. The application whitelist is located in */etc/nfcee_access.xml*.

Resource limited developers will have great difficulties when it comes to using SEs. Cooperation with many different entities is required in order to achieve this, which requires access to many resources. However, it is not an impossible feat. An example of a product that has successfully achieved this, is Semble [15]. Semble is an application used for mobile payments. This has been achieved through cooperation with telecommunication and banking entities. However, users are required to insert specific SIM cards into their devices in order for this to work.

2.1.1.2 Trusted Execution Environment

The concept of a Trusted Execution Environment (TEE) [5] is very similar to that of a SE. It provides an isolated execution platform for running trusted applications. A TEE is authorised, isolated and loaded during the secure boot process provided by certain processors. This process isolates it from the rest of the OS, providing secure application execution. Figure 2.3 shows the concept of a TEE.


Rich Execution Environment (REE)
Trusted Execution Environment (TEE)
Normal Apps
Apps w. Secure OS Support
Trusted Apps
Trusted Apps
TEE Functional API
TEE Client API
TEE Internal API
Rich OS
Trusted OS
Monitor
Cortex-A Hardware Platform (TBSA Compliant)
Secure Element (SecurCore)

source:http://www.arm.com/assets/images/TEE.gif

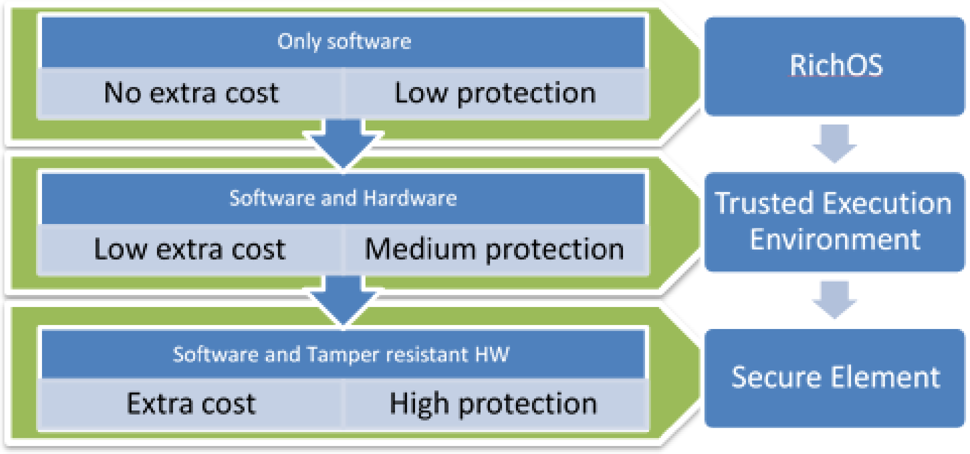
Figure 2.3: TEE in a mobile environment [16].

A TEE can be thought of as a secure environment running on the device that is separate from the OS. As with SEs there are issues that developers face when wanting to use this technology:

- **TEE providers.** As TEE technology emerged, different providers came up with their own TEE platforms, such as Solacia [17], Trustonic [18] and Qualcomm [19]. The TEE platform is installed on the device during manufacturing, not all manufacturers choose the same TEE platform, which means that different devices come with different TEE platforms. Ideally this should not be an issue, but developer licenses are required in order to develop and install trusted applications. This means it can cost quite a lot of money to support a multitude of devices. Many developers do not have access to this kind of money, which means it is not a viable option as of yet.

- **New Technology.** The TEE platform is still relatively new. There are still a large amount of Android devices that do not come with a TEE platform (estimated 75 percent [20]), however the majority of newer devices come shipped with one. Currently this technology will not be feasible to implement for most developers as there are not enough supported devices. As the technology matures so will the ability for developers to use it.

TEEs appear more accessible to developers, and have greater functionality than SEs [21]. However they are less secure than SEs. SEs have an Evaluation Assurance Level[4] (EAL) rating of 4+ [22] whereas TEEs have a EAL rating of 2+ [23]. Figure 2.4 gives a quick comparison of the security provided by software, TEEs and SEs. It shows that TEEs have a lower implementation cost than SEs and also provide less protection, but offer higher protection and requires a small extra cost compared to software only implementations.

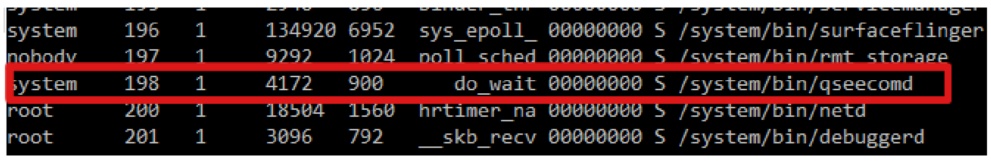


source:http://www.globalplatform.org/images/tee-simple_clip_image004.png

Figure 2.4: Comparison of security offered by software, TEEs and SEs [5].

[4]A higher EAL indicates a higher level of confidence in the security of a product

An example of TEE applications running on a device is the Android keystore system [24]. On certain devices the keystore system can be hardware backed, meaning that trusted applications running on the TEE are used by the system to provide secure cryptographic operations. The way communication occurs between the OS and TEE applications is through a driver running on the OS. Figure 2.5 shows the Qualcomm Secure Execution Environment Communicator (QSEECOM) driver running on a Nexus 5 device.



```
system    196   1     134920 6952  sys_epoll_ 00000000 S /system/bin/surfaceflinger
nobody    197   1     9292   1024  poll_sched 00000000 S /system/bin/rmt_storage
system    198   1     4172   900      do_wait 00000000 S /system/bin/qseecomd
root      200   1     18504  1560  hrtimer_na 00000000 S /system/bin/netd
root      201   1     3096   792   __skb_recv 00000000 S /system/bin/debuggerd
```

Figure 2.5: TEE driver running on a Nexus 5 device.

2.1.2 Cloud based architecture

In a cloud based architecture, the processing and storage of the sensitive information is performed in the cloud [25]. The cloud infrastructure can be thought of as an isolated execution platform similar to that of a SE or TEE. The application running on the device can be thought of as relay device, relaying data between the cloud infrastructure and authentication infrastructure. Figure 2.6 illustrates the concept.

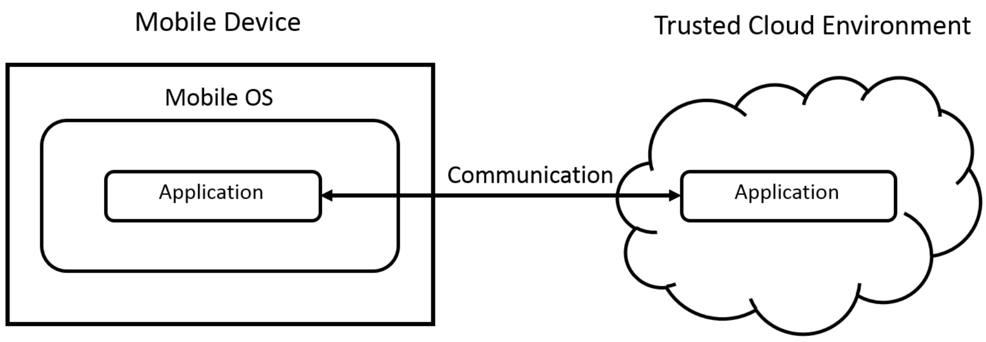


Figure 2.6: Cloud based architecture.

Even though security sensitive operations are not performed on the mobile device itself, sensitive information will pass through the phone at some point, meaning that the application using the data still needs to be protected. Storing sensitive data in the cloud and retrieving it when needed can problematic:

- Cloud infrastructure needs to be acquired and maintained by the developers. This reason alone will be enough for most developers to not implement this type of architecture, as it requires constant upkeep and resources.

- Whenever the data needs to be retrieved, a connection to the cloud infrastructure needs to be made. This means users require an Internet connection every time they want to use the application, which is not ideal.

A cloud based infrastructure requires a large amount of resources on behalf of the application developers and is not feasible in many circumstances. However it can be used in production environments. An example of cloud based security used in a commercial application is Google Wallet[5]. Where user credit card details are stored on secure servers. These servers create tokens[6] that allow users to perform mobile payments. Even if the one-time token is exposed in the application, attackers will not be able to reuse the token since it is not valid anymore. A cloud based architecture works in this case due to the payment infrastructure that has been set up. It is often the case that developers do not have control over the infrastructure they need to authenticate with. Therefore this type of architecture cannot always make use of tokens and will require persistent storage of sensitive data, meaning that when the data is retrieved it will be exposed in the application.

[5]`http://www.google.co.nz/wallet/`

[6]A type of one-time password

2.1.3 Software based architecture

In a software based architecture the processing or storage of sensitive information is performed inside the application itself. This is the most common approach for developers as it does not require any extra resources. However it is the least secure way to handle and process sensitive information as there are many attack vectors that can be used to compromise the data, as shown in Chapter 3. Code running inside Android applications can easily be exposed to attackers who will be able to extract sensitive data. Therefore, if sensitive information is processed in any part of the code, attackers will be able to get a hold of it.

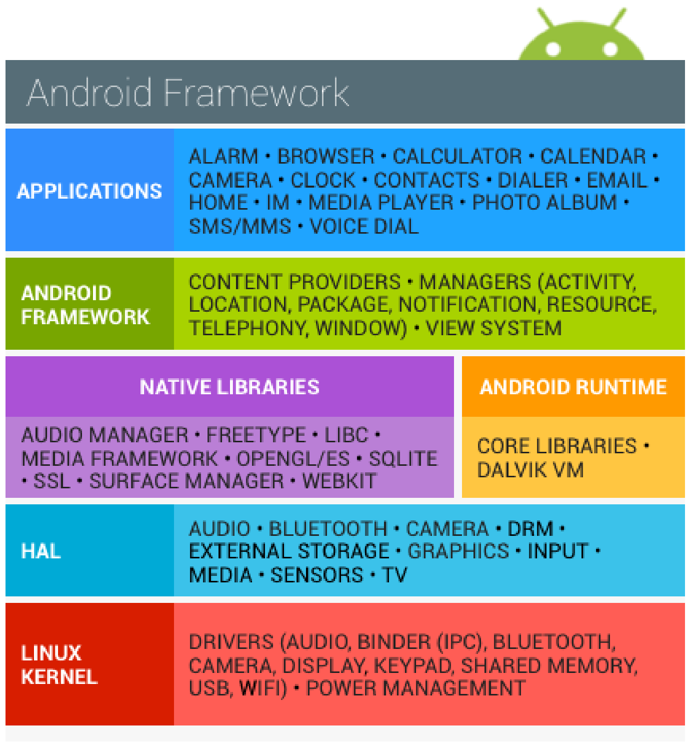
2.1.4 Summary

Hardware and software based architectures have the ability to provide isolated execution and secure data storage, however when used in a mobile environment, sensitive data is eventually used and exposed in the software implementation. This means that even if hardware or cloud based architectures are used, the software based side of the application will still need to be secured as much as possible. Therefore the main focus of the research is to protect the software side of applications.

2.2 Android

Android is an open-source mobile OS based on the Linux kernel and is the most popular mobile OS [1] with over 80% of the mobile market share. A few Android security features worth mentioning are:

- The application sandbox. The application sandbox ensures that applications are isolated from one another in terms of execution and application data. Each application is given a unique ID and is run as a separate user, this is done at a kernel level which means that the sandbox applies to all applications running above the kernel. Figure 2.7 illustrates this, the application sandbox governs all layers above the Linux Kernel layer.


Android Framework
APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

source:https://source.android.com/security/images/android_software_stack.png

Figure 2.7: Android software stack [26].

- Application permissions. In order for Android applications to access other Android resources they have to explicitly ask for permission, this is due to the application sandbox [27]. When an application is first installed, the user will have to grant permissions to the application, in order to use the specified resources.

The attacks covered in Chapter 3 are used by attackers to bypass and break these security features which means that the protection provided by the Android OS is not sufficient. Which means that other protection methods need to be considered when developing applications, which are covered in Chapter 4.

2.2.1 Android applications

Android applications come in the form of Android Package (APK) files [28]. An APK file contains all the code and data for an application, which is used to install the application on the device. A few noteworthy locations on the Android file system that are used for applications:

- The */data/app/* directory contains user applications. This is the default location that user applications get installed to.
- The */system/app/* directory contains system applications. These are the applications that come preinstalled on the device and cannot be removed.
- The */data/data/* directory is the default location where applications store their data. This directory is private storage that other applications cannot access.

These locations on the file system are normally protected by the application sandbox, however as mentioned before the sandbox can be bypassed. As a result these file system locations can be accessed by attackers and cannot be assumed to be secure.

2.2.2 Android keystore system

The Android keystore system provides functionality to generate, store, protect and use cryptographic keys [29]. The *KeyStore* API was introduced in API level 1, but is just an interface to an underlying keystore implementation. The underlying keystore implementations expose interfaces that adhere to the *KeyStoreSpi* class. One of the keystore types is the *AndroidKeystore* implementation and can be software or hardware based. A hardware based keystore implies that a SE or TEE is available that has trusted applications installed on it, that provide cryptographic operations. The software based implementation uses cryptographic operations executed on the OS. Developers can use the keystore system to:

- Generate keys.
- Encrypt/decrypt data.
- Sign/verify data.

When a key is generated, it is placed on the file system in the */data/misc/keystore* directory. The stored keys are protected using encryption, through a Key-Encryption Key (KEK). The KEK is generated from the device's masterkey along with other data relating to the device [30]. If the keystore is hardware backed, the master key is stored inside secure hardware, in a software based implementation, the master key is generated using the the device unlock password/PIN [31]. The keystore is very important when it comes to storing data and should be used by developers when performing cryptographic operations on sensitive data. However, when sensitive data is decrypted during runtime, it can be extracted from the application. Therefore protection techniques are required to make it more difficult for attacker to do this.

2.3 Security Guidelines and Standards

The Open Web Application Security Project (OWASP) is an organization that focuses on information security. The goal of this organization is to create awareness when it comes to software security. In terms of mobile security they have released a list of top 10 risks to mobile applications [32]. Table 2.1 illustrates this along with the vulnerability distributions for each of the risks [33].

Results show that Insecure Data Storage and Lack of Binary Protections are the top two most common risks to mobile applications, both of these categories

Category	Vulnerability Distribution
M1:Weak Server Side Controls	6%
M2:Insecure Data Storage	17%
M3:Insufficient Transport Layer Protection	16%
M4:Unintended Data Leakage	13%
M5:Authorization and Authentication	6%
M6:Broken Cryptography	3%
M7:Client Side Injection	4%
M8:Security Decisions Via Untrusted Inputs	1%
M9:Improper Session Handling	2%
M10:Lack of Binary Protections	19%
No Category	13%

Table 2.1: OWASP Mobile top 10 [32] with corresponding vulnerability distribution [33].

directly apply to this research. This indicates that there are not enough resources available for developers to be able to protect applications properly. The information given by OWASP to remedy these issues is very limited. There are two resources provided by OWASP [34] [35] that provide more information about reverse engineering and code modification but only provide limited implementation details. OWASP resources are excellent at providing a general overview of the subject, but do not go into specific implementation details that developers often require.

The National Institute of Standards and Technology[7] (NIST) is an organization that is involved in the creation of standardization documents. NIST have released mobile security standards such as [36] and [37]. These standards briefly mention security concepts but do not sufficiently cover security techniques or provide implementation details that can be used by developers.

[7]`http://www.nist.gov/`

The International Organization for Standardization[8] (ISO) is an organization responsible for creating international standards. The International Electrotechnical Commission[9] (IEC) is a standardization organization for electronic technology. These organization have created the ISO/IEC 27000 series standards, which are standards related to information security. They contain information in regards to application security, but not specifically for mobile security. Certain concepts can be transferred over to mobile security but many of the standards do not apply.

In summary, there are resources available to developers such as standards and guidelines. However, these only briefly cover security concepts and do not provide implementation details. The research performed in this thesis is to be seen as a guidance document for developers. It provides many implementation details, explains how attackers are able to compromise applications and data (Chapter 3), and goes over security techniques that can be used to hinder attackers from doing so (Chapter 4).

[8]`http://www.iso.org/iso/home.html`

[9]`http://www.iec.ch/`

Chapter 3

Attack Phases

The sections outlined in this chapter examine the phases an attacker will go through when attacking applications. During each phase of the attack, different techniques are employed in order to obtain sensitive information from applications. The attack life cycle is shown in Figure 3.1.

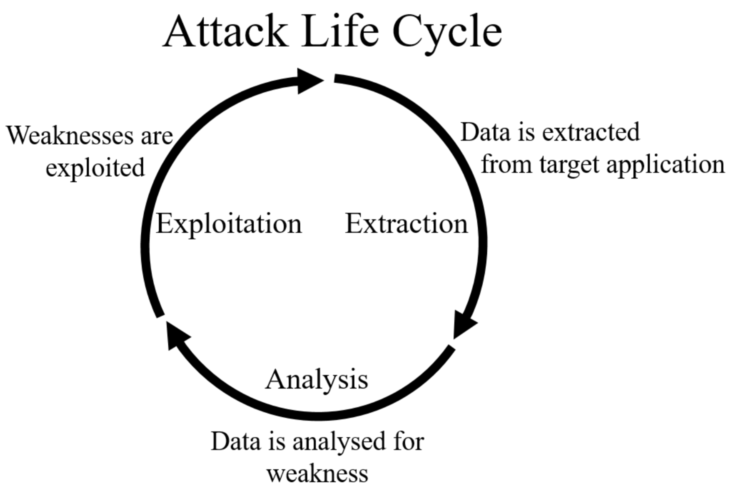


Figure 3.1: Attack life cycle

3.1 Phase I: Extraction

The extraction phase is about gathering information from the device and applications in order to identify valuable data that might be stored [35]. Attackers make use of techniques such as rooting and device emulation to perform this stage of the attack.

3.1.1 Rooting Devices

Rooting gives users root privileges on a device, granting them the ability to make changes to the underlying file system, which would otherwise not be possible. There are legitimate reasons for users rooting their devices as it allows for much greater device customization. However, doing so compromises the device and poses a security risk, as it allows applications to break out of their application sandbox. With root access, applications are able to explore any part of the device file system, including the private file storage areas of other application.

The way users go about rooting, changes from device to device. Some manufacturers actively block users from rooting their device, whilst others do not. Certain devices are easier to root than others, many of the Google Nexus series devices allow for the user to unlock the boot loader, which makes the rooting process much easier. The boot loader [38] is responsible for loading the OS, if unlocked it is possible to change system files, for example adding the su binary. Which is often used to execute privileged commands on Linux based devices.

There are many tools available that provide users the ability to root a device. A few are listed below:

- KingoRoot[1]

[1]`https://www.kingoapp.com/`

- OneClickRoot[2]
- Framaroot[3]
- KingRoot[4]
- RootMaster[5]

These tools make it very easy for attackers of any skill level to root their devices. Once the device is rooted, attackers can access any part of the file system, allowing them to extract APK files, private application data and modify the files system as they see fit.

3.1.2 Using Emulators

Attackers can use Android emulators for the extraction phase of the attack lifecycle, they can be used to achieve the same results as device rooting. Emulators are used to simulate real Android devices, allowing developers to test applications on many different devices without physically owning them. Since emulated devices are run entirely in software, they are much more customisable than hardware devices. Attackers can make use of this to change the environment in which applications are run, for example enabling root access.

Emulators are available from the Android developer tools, these are the official emulators supplied for developers and come with an unlocked boot loader. Meaning that the su binary can be pushed to the device to give applications root.

[2]`https://www.oneclickroot.com/`
[3]`http://framaroot.net/?scn=1`
[4]`http://www.kingroot.net/`
[5]`http://rootmasterapk.org/`

3.1.3 Android Debug Bridge

The Android Debug Bridge (ADB) is a tool included in the Android SDK that allows a user to communicate to Android devices. ADB gives users a shell terminal that can be used to execute commands for example it can be used to push files such as the su binary to the file system. Using ADB it is still possible to pull data from the */data/app/* directory, which contains APK files. Commands such as *pm list packages -f* can be used to see what applications are installed on the device and using the pull command *adb pull /data/app/[.apk file] [location]* it is possible to retrieve any APK file installed on the device. Giving attackers the source code they need to start the next stage in the attack lifecycle.

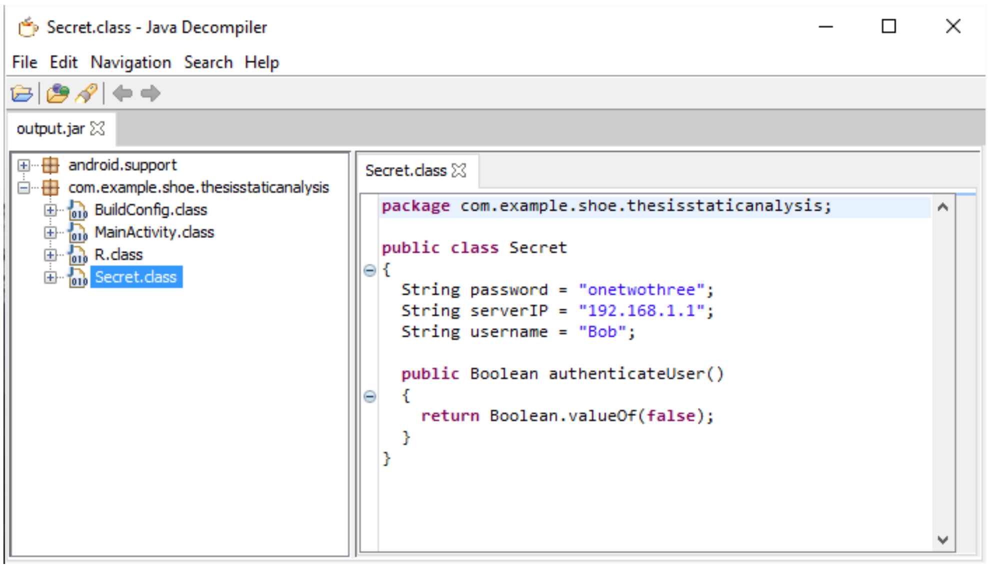
3.2 Phase II: Analysis

The analysis phase of an attack consists of the examination of extracted data [35]. The goal is to understand how the application works to determine where and how the sensitive data is used. This is often referred to as reverse engineering and is performed through static and dynamic analysis.

3.2.1 Static Analysis

Static analysis is the process of looking through static code in order to understand more about an application. Attackers first extract application binaries from the device, which are then analysed to gain information about how the application works. Attackers are able to obtain sensitive information such as proprietary algorithms or hardcoded values from this process. The more information gathered from this process, the easier it is to further exploit the application.

Listing A.1 is an example of a java class that can be used in an android application. It is listed to illustrate standard java source code before it is compiled into an application. Once the application has been installed, an attacker can get a hold of the application binaries as discussed in Section 3.1. After the APK has been obtained, it can be converted into a jar file using a tool such as dex2jar [39]. A standard java decompiler can then be used on the jar file to view the java code contained in the application. The result of this is shown in Figure 3.2.


Secret.class - Java Decompiler
File Edit Navigation Search Help
output.jar
android.support
com.example.shoe.thesisstaticanalysis
BuildConfig.class
MainActivity.class
R.class
Secret.class
Secret.class

```
package com.example.shoe.thesisstaticanalysis;

public class Secret
{
  String password = "onetwothree";
  String serverIP = "192.168.1.1";
  String username = "Bob";

  public Boolean authenticateUser()
  {
    return Boolean.valueOf(false);
  }
}
```

Figure 3.2: Decompiled java code using JD-GUI [40].

Although not identical, the generated code is an accurate representation of the original source code, the attacker can then inspect the code to view to understand more about the application and extract static values from the application.

The reason APK files are so easily decompiled is due to the nature of Java code. Android applications are written in Java which is compiled into java bytecode (.class files), which in turn is translated into the DEX file used by Android. When Java is turned into bytecode, a lot of meta-data is preserved such as class names, method names and variable names. When a decompiler is used to turn bytecode back into source code, this meta-data is used to reconstruct the code, producing readable source code.

3.2.2 Dynamic Analysis

Dynamic analysis is the act of analysing code while it is being executed. It is used to extract information from applications during execution, which allows attackers to obtain data that was not discovered during the static analysis process. Two techniques used to perform dynamic analysis are described.

3.2.2.1 Debugging

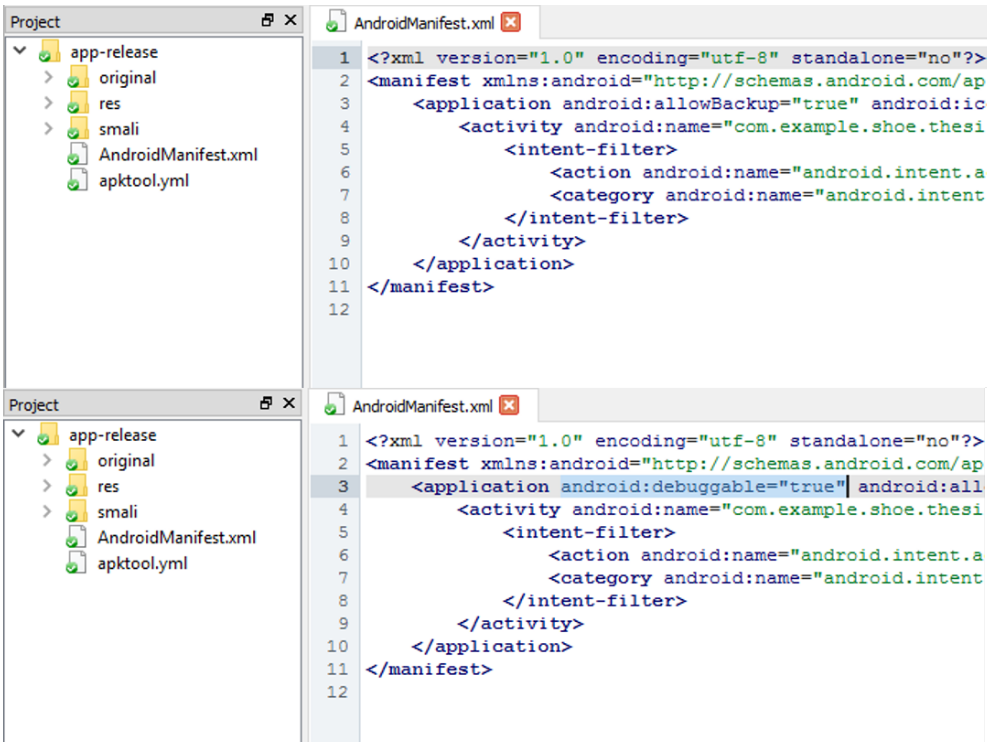
Android applications can be debugged by developers in order to step through application code whilst the application is running, this is done to detect errors and inspect the program at runtime. This can only be done on applications that have the debugging flag set to true in the android manifest. The debugging flag is automatically set to true in a development environment, however when an application is released the flag is set to false by default. Attackers can modify an application and set the debugging flag to true in order to debug the application, using the debugging tools provided by the Android developer environment [41].

The program in Listing A.2 is used to illustrate how this attack can be used to extract data at runtime. The password is programmatically generated which means that an attacker will not be able to determine the password using static analysis. This is a basic example but we can imagine that in a real application the password is stored in an encrypted form and the decrypted version is passed into the method used for authentication on line 11.

The process an attacker goes through to perform this attack is shown. After an attacker has retrieved the APK, a tool such as apktool[6] can be used to

[6] `http://ibotpeaches.github.io/Apktool/`

modify and recompile the APK. Adding 'android:debuggable="true"' to the AndroidManifest.xml file will make the application debuggable once recompiled. Figure 3.3 shows this using APK Studio[7].



Project

- app-release
  - original
  - res
  - smali
  - AndroidManifest.xml
  - apktool.yml

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/ap
    <application android:allowBackup="true" android:ic
        <activity android:name="com.example.shoe.thesi
            <intent-filter>
                <action android:name="android.intent.a
                <category android:name="android.intent
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Project

- app-release
  - original
  - res
  - smali
  - AndroidManifest.xml
  - apktool.yml

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/ap
    <application android:debuggable="true" android:all
        <activity android:name="com.example.shoe.thesi
            <intent-filter>
                <action android:name="android.intent.a
                <category android:name="android.intent
            </intent-filter>
        </activity>
    </application>
</manifest>
```

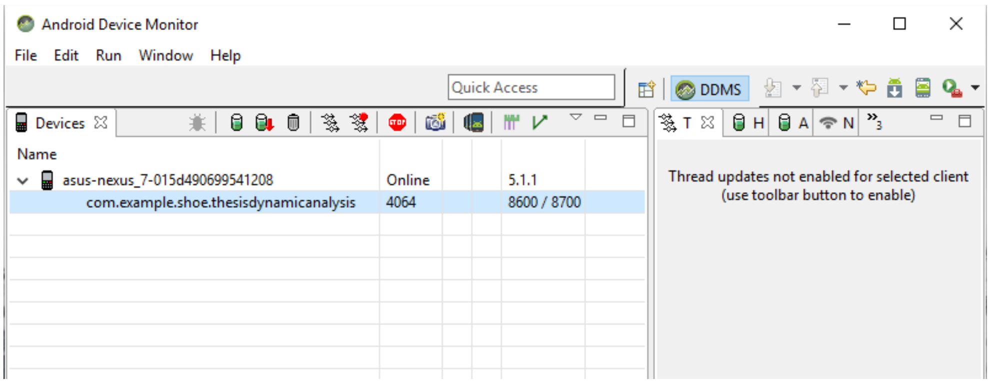
Figure 3.3: Modified APK using APK Studio.

Once an application has been modified, it needs to be resigned as the binary signature has been broken. After resigning the application, the old version needs to be deleted from the device and the modified version can be installed using ADB. If the application is run, it will show up in the Android Device Monitor[8] as shown in Figure 3.4, which is an indication that it is debuggable.

Now that the application is debuggable it is possible to make use of the developer tools provided by Android to examine the application in detail. The ultimate goal is to step through the code and look at specific variables in the application during runtime. The developer tools are able to attach to a debuggable process, running on the target device as shown in Figure 3.5. If source code is available for the target process, it is possible to add breakpoints and

[7]`http://www.vaibhavpandey.com/apkstudio/` (GUI for apktool)

[8]Tool provided for Android developers to monitor a device



Name

asus-nexus_7-015d490699541208	Online	5.1.1
com.example.shoe.thesisdynamicanalysis	4064	8600 / 8700

Thread updates not enabled for selected client (use toolbar button to enable)

Figure 3.4: Modified application is now debuggable.

step through the code.

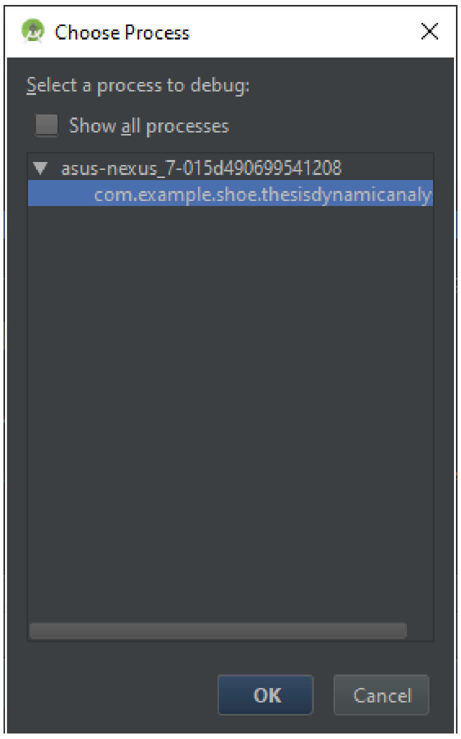
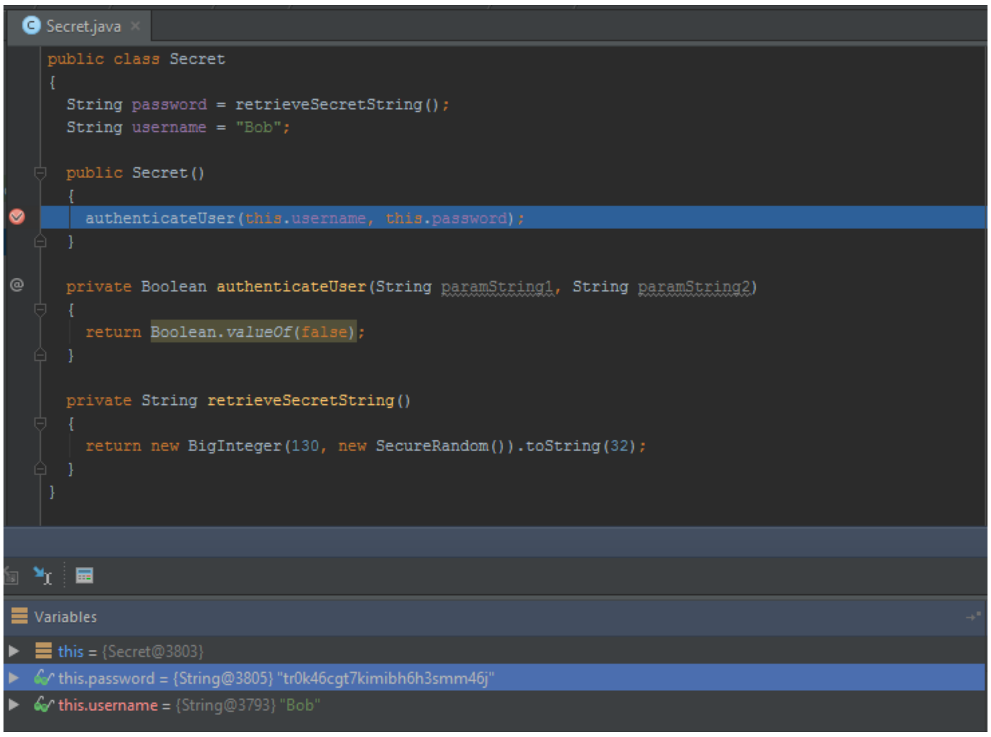

Choose Process
Select a process to debug:
Show all processes
asus-nexus_7-015d490699541208
com.example.shoe.thesisdynamicanaly
OK
Cancel

Figure 3.5: Attaching to a debuggable application.

To add the source code and breakpoints, a new application has to be created with the same application package name as the running process. This is how the developer tools relate the source code to the running process. The application source code can be extracted from the modified APK using techniques discussed in Section 3.2.1, which is then imported into the application. Breakpoints can then be added to the imported code, when the breakpoints are triggered by events in the running process it is possible to inspect the process, as shown in Figure 3.6.



```java
public class Secret
{
  String password = retrieveSecretString();
  String username = "Bob";

  public Secret()
  {
    authenticateUser(this.username, this.password);
  }

  private Boolean authenticateUser(String paramString1, String paramString2)
  {
    return Boolean.valueOf(false);
  }

  private String retrieveSecretString()
  {
    return new BigInteger(130, new SecureRandom()).toString(32);
  }
}
```

```
this = {Secret@3803}
this.password = {String@3805} "tr0k46cgt7kimibh6h3smm46j"
this.username = {String@3793} "Bob"
```

Figure 3.6: Inspecting application after breakpoint trigger.

Although the sample application used is a very simple one, it is effective in showing how detrimental this technique is to applications processing sensitive information. If sensitive data is exposed at any stage of the application lifecycle, it can be inspected at runtime using this technique. This technique can be applied to any application which makes it a big threat to any application developer wanting to keep information secure.

3.2.2.2 Memory Dump

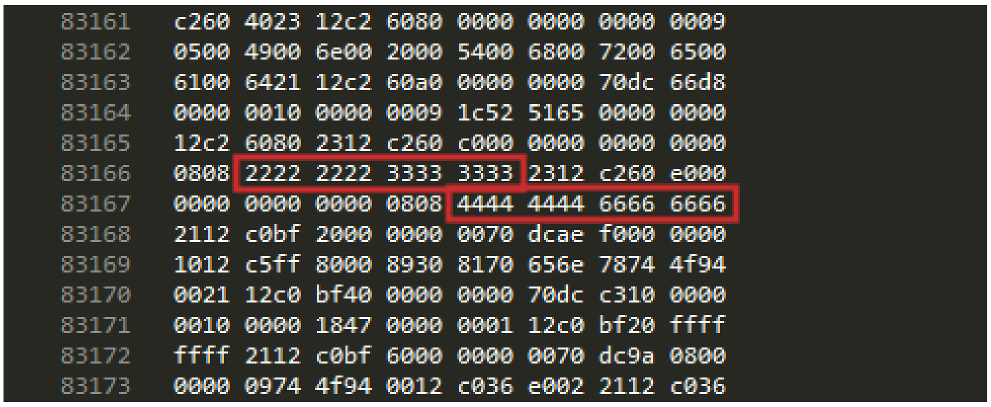
When applications are executed, they are loaded into device memory. During application execution many objects are created, when these objects are no longer needed, the memory they occupy needs to be reclaimed which is the job of the garbage collector. The garbage collector performs Garbage Collection (GC) every so often to free up memory, the more objects are created, the more often GC occurs. The process of GC is performed by marking the target memory as free, meaning that other applications can use the memory. This does not delete the memory from the location, it just allows other applications to write over the existing memory stored there.

If applications are using sensitive data, it will eventually end up somewhere in the application memory. If attackers are able to get a hold of the application memory it could be possible to extract this sensitive data. The difficulty to perform this attack is quite high as it is a time dependant attack. To get a hold of the target sensitive information an attacker would need to know when the data is used in memory, and then perform a memory dump before GC occurs. There are several ways in which an attacker can perform memory dumps as outlined below.

Memory dump through debugging

The memory of debuggable applications can be dumped using the Android Device Monitor (ADM). The application needs to be made debuggable as shown in Section 3.2.2.1, which allows attackers to dump device memory. This produces a *.hprof* file containing the application memory. Skilled attackers can analyse these files and retrieve sensitive information from them. As an example, the code in Listing A.3 is run in debug mode on the device after a button press. The code is blocking to make it easier to capture a successful memory dump, by preventing GC from cleaning up the memory. Once the thread has started, the ADM is used to capture a memory dump. The resulting file can be analysed using memory analysis tools such as jhat[9] or VisualVM[10]. Figure 3.7 shows a section of the memory dump. It illustrates that hardcoded and dynamically generated data can be seen in the application memory. It can be imagined that these values are not just random pieces of data, they can be cryptographic keys, passwords or other sensitive data.

[9]`https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jhat.html`
[10]`https://visualvm.java.net/`



```
c260 4023 12c2 6080 0000 0000 0000 0009
0500 4900 6e00 2000 5400 6800 7200 6500
6100 6421 12c2 60a0 0000 0000 70dc 66d8
0000 0010 0000 0009 1c52 5165 0000 0000
12c2 6080 2312 c260 c000 0000 0000 0000
0808 2222 2222 3333 3333 2312 c260 e000
0000 0000 0000 0808 4444 4444 6666 6666
2112 c0bf 2000 0000 0070 dcae f000 0000
1012 c5ff 8000 8930 8170 656e 7874 4f94
0021 12c0 bf40 0000 0000 70dc c310 0000
0010 0000 1847 0000 0001 12c0 bf20 ffff
ffff 2112 c0bf 6000 0000 0070 dc9a 0800
0000 0974 4f94 0012 c036 e002 2112 c036
```

Figure 3.7: Data found in memory dump

This demonstrates the fact that sensitive information is kept in application memory and can be retrieved through dumping memory. The attack itself is quite difficult to perform, it requires understanding of the application itself, good timing between GC cycles and requires knowledge of memory analysis.

Memory dump on rooted device

The application sandbox is used to run apps in isolation of one another, this includes the memory. When a device is rooted the security provided by the sandbox is nullified and applications are able to access each others memory. Therefore when a device is rooted, applications can be created or installed on the device to read the memory of specific applications. There are many different ways to do this, a few methods are briefly outlined below.

- As shown in [42], it is possible to create a native application in C to dump device memory. The code provided in [42] has been slightly modified (shown in Listing A.4) to print out the stack for the specified process id. The binary can be installed and executed on the device using ADB.
- In [43] it is shown how to dump the memory on a rooted device by installing gdbserver[11]. Gdbserver is a program that can be used to re-

[11]`https://sourceware.org/gdb/onlinedocs/gdb/Server.html`

motely debug applications using the GNU Project Debugger (GDB)[12]. It is then possible to remotely debug a specified process and dump the device memory.

Compared to the debugging technique, the methods shown produce memory dumps that are a bit more difficult to analyse and require more effort to produce. The device requires root and more advanced techniques need to be used in order to dump the memory, however, they seem just as effective in obtaining sensitive information from applications.

3.3 Phase III: Exploitation

If an attacker has not been able to obtain the information they wanted in the previous to stages, they move onto the exploitation phase [35]. Which mainly involves application tampering. Application tampering has been partially explored in Section 3.2.2.1, where the manifest had been altered to provide debugging functionality. Tampering explored in this section will show how attackers can make an application function differently than it was designed to.

The steps to tamper with an application are outlined:

1. Disassemble the bytecode of the application. The code produced is in a format that is more difficult to understand than java source code.
2. Modify the disassembled code. Code can be modified to add in extra functionality, remove functionality or change existing functionality.
3. Assemble the modified code and repackage it back into the application.

The disassembling/assembling part of tampering can be performed using a tool called smali/baksmali [13], which produces a language called smali that can

[12]`https://www.gnu.org/software/gdb/`
[13]`https://github.com/JesusFreke/smali`

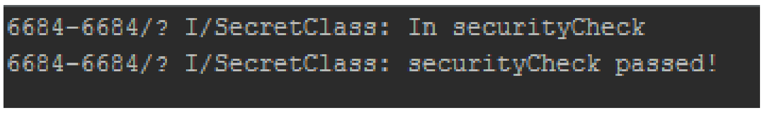
be modified. Smali/baksmali can be used as a standalone tool but since it is integrated with APK Studio, it will be used in the examples shown. The code in Listing A.5 will be modified to disable the security check put in place. The security check in the code represents obstacles that an attacker has to bypass in order to successfully tamper with an application. Listing A.6 shows the smali representation of the Java code which will be modified. Line 50 in the smali code shows:

```
const/4 v0, 0x0
```

If this is changed to:

```
const/4 v0, 0x1
```

The method *securityCheck* will return **true** instead of **false**. The code has to be assembled, the resulting APK needs to be resigned and the application needs to be installed on the device. The result of running the application once more produces the expected output as shown in Figure 3.8.



```
6684-6684/? I/SecretClass: In securityCheck
6684-6684/? I/SecretClass: securityCheck passed!
```

Figure 3.8: Result of running tampered application on device

This is a very simplistic example but it shows the idea behind application tampering. Attackers are able to add entirely new methods and remove security features that developers have implemented. The attack is relatively easy to perform when making small changes and not much knowledge is required. However, as code complexity increases and the more modifications that need to be made, the more difficult it becomes to successfully tamper with an application.

Chapter 4

Security Techniques

In this chapter, security techniques are explored that can be used to hinder attacker from performing attacks outlined in Chapter 3.

4.1 Obfuscation

4.1.1 Description

Higher level languages such as Java and C# contain a lot of meta-data when they are compiled into their respective binaries. This meta-data contains information such as variable, class and method names. When these high level languages get decompiled they still contains this metadata and therefore can be translated back into code that resembled the original source code [44] [45]. Lower level languages such as C and C++ have this meta-data stripped away when compiled, therefore making it much harder to understand when decompiled.

Obfuscation is the process of transforming code or binaries in such a way that it makes it much more difficult to understand once decompiled, even though the code is functionally still the same. It is more effective to use on higher level languages than lower level languages, since the meta-data is one of the main targets of obfuscators. Security through obscurity[1] is the main idea behind obfuscation and is used to prevent attackers from easily reverse engineering applications. However, obfuscation should be used in conjunction with other security practices since determined enough attackers will always be able to bypass this security technique.

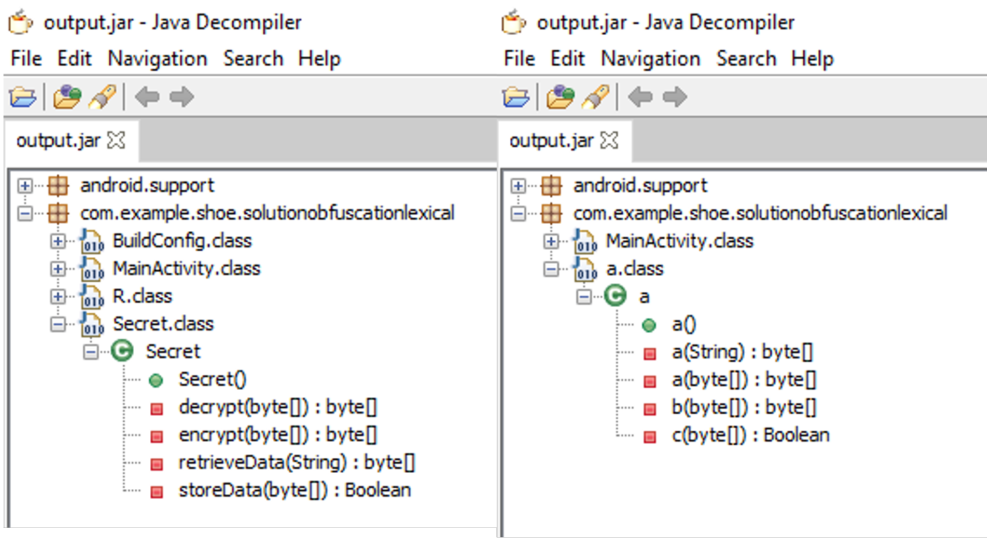
There are many different techniques used to provide code obfuscation [45–49]. The primary techniques will be briefly outlined in the sections below.

4.1.1.1 Lexical Transformations

This technique is employed by virtually all obfuscators, it renames identifiers such as variable, method and class names, to something incoherent. Meaning that when code is decompiled, the names of identifiers will not give away any information as to its function. For example, a function that is used to decrypt data is called *decrypt()* but is renamed to *abc*. When an attacker decompiles the code, they will see a function called *abc* which does not give away any information as to its purpose. If the application is large and this is applied to all identifiers it can be quite time consuming to reverse engineer. Figure 4.1 illustrates this.

The lexical transformations applied make it more difficult to analyse the code. The class has been renamed from *Secret* to *a* giving away no information as to its functionality. The same goes for the methods contained in the class. Attacker will have to examine each method individually in order to understand more about their functionality. This might not be difficult in a application with one class, but as more classes are introduced it becomes challenging. However, there are some issues associated with this transformation [50]:

[1]Security through obscurity is a well known security concept that relies on hiding implementation details to provide security to an application


output.jar - Java Decompiler
File Edit Navigation Search Help
output.jar
output.jar - Java Decompiler
File Edit Navigation Search Help
output.jar

```
android.support
com.example.shoe.solutionobfuscationlexical
    BuildConfig.class
    MainActivity.class
    R.class
    Secret.class
        Secret
            Secret()
            decrypt(byte[]) : byte[]
            encrypt(byte[]) : byte[]
            retrieveData(String) : byte[]
            storeData(byte[]) : Boolean
```

```
android.support
com.example.shoe.solutionobfuscationlexical
    MainActivity.class
    a.class
        a
            a()
            a(String) : byte[]
            a(byte[]) : byte[]
            b(byte[]) : byte[]
            c(byte[]) : Boolean
```

Normal Code   Obfuscated Code

Figure 4.1: Example of a lexical transformation.

- Not all code can be obfuscated. Identifiers that belong to standard java libraries cannot be renamed since other applications have to use those same libraries. The result of this is that it might be relatively easy to understand parts of an application that use many calls to these libraries.

- Code that is designed to be called using reflection cannot be obfuscated, since the identifiers are needed for this.

- Crash logs coming from an obfuscated application will be much more difficult to understand. Since identifiers have been renamed, the crash log will point to obfuscated code which developers will have a hard time understanding. However, a reverse mapping is often provided to developers when code is obfuscated which is used counter this problem.

An added benefit from lexical transformations is its ability to shrink the size of a program. Stripping out a lot of the meta-data results in smaller file sizes.

4.1.1.2 Control Transformations

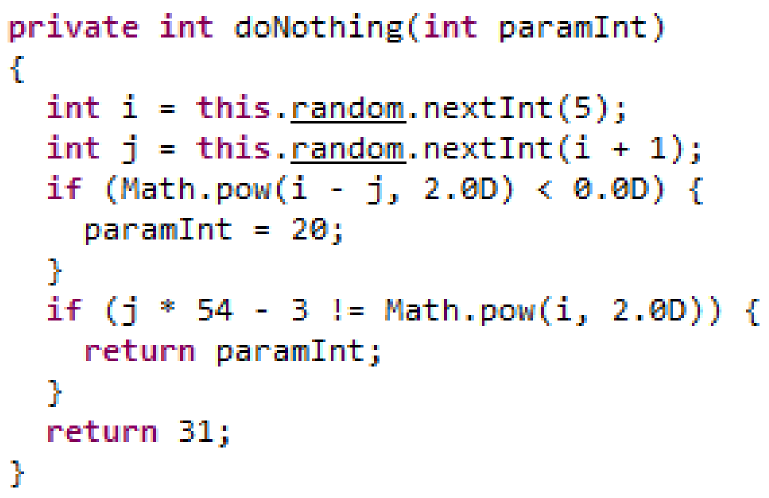
The idea behind control transformations is to change how a program is perceived to flow. Extra code can be inserted into the application which appears to add more flow paths even though the program stays functionally the same. This can done using opaque predicates [48] [49] [51]. Opaque predicates are expressions that always evaluate to a certain value, even though, when decompiled they appear to evaluate to other possible values. As an example, imagine a predicate always evaluates to *false* and is inserted into an if statement as the conditional expression. Any code contained within the statement will not be executed but when analysed by the attacker, it might seem that the code gets executed occasionally. If many of these predicates are included in a piece of code, it can be quite challenging for an attacker to deduce what parts of the code actually get executed.

Examples of opaque predicates are shown in Listing A.7. Line 10 contains a predicate that always returns *false* and the predicate on line 14 always returns *true*. The function *doNothing* does absolutely nothing, the value returned from the method is exactly the same as the value inserted. The values of both predicates are known before execution time by the developer, however it is difficult to determine this while looking at the code when it is decompiled (Figure 4.2). This confuses attackers looking at the code.

4.1.1.3 Data Transformations

Data transformations are used to hinder decompilation by representing data types and structures in uncommon ways [46] [49]. Common transformations are listed below:

- String obfuscation is used to make sure that strings are not represented in plain text once the program has been decompiled. This can be performed



```
private int doNothing(int paramInt)
{
  int i = this.random.nextInt(5);
  int j = this.random.nextInt(i + 1);
  if (Math.pow(i - j, 2.0D) < 0.0D) {
    paramInt = 20;
  }
  if (j * 54 - 3 != Math.pow(i, 2.0D)) {
    return paramInt;
  }
  return 31;
}
```

Figure 4.2: Example of an opaque predicate for a control transformation.

by storing encrypted versions of strings in the code and decrypting them when needed. The same thing can be done with classes, they can be encrypted when stored and decrypted during runtime. Even though it might be trivial for attackers to decrypt the data since the decryption algorithm has to be included in the source code, it does add an extra step for the attackers to go through.

- Variable splitting can be used to represent variables using different variables. For example a boolean value can be represented by two integers. When this code is decompiled the attacker will need to go through extra steps in order to deduce the actual value.

- Splitting classes, is where the functionality provided by one class can be split into many smaller classes. The result of this produces a large amount of obfuscated code that can be difficult to sift through.

A simple example of string obfuscation is shown in Listing A.8. Strings can be encoded before being statically stored in a variable using the *stringEncode* function. The variables *string*, *stringFromEncodedString* and *stringFromBytes* all the same value of *secretPassword123* when decoded. When the string is needed, a call to *stringDecode* transforms the string back its original form.

The encoded value cannot easily be interpreted by an individual performing static analysis.

4.1.1.4 Anti-Decompilation

The three obfuscation transformations mentioned above are targeting the attacker itself, obscuring the code in order to make it more difficult for the attacker to understand it. There are other defences that can be inserted by an obfuscator to confuse the program used in the decompilation process. This is often referred to as anti-decompilation where code is inserted into the application to target a specific weakness in the decompiler, which can cause the decompiler to crash or not perform the decompilation properly. Examples of this can be found in [47]. This technique can be quite effective as it prevents decompilation of entire applications, however the smarter decompilers get, the more advanced these obfuscation techniques need to become.

4.1.2 Implementation

Implementing obfuscation manually is not feasible, the transformations have been automated and there are quite a few tools available. Popular obfuscators are listed in table 4.1

Proguard is a free obfuscator included in the Android build tools and can be used by developers to obfuscate code [52]. It primarily makes use of lexical transformations to provide obfuscation and can easily be deployed in an application. By default ProGuard is disabled, but enabling it is as simple as setting

[2] `https://www.guardsquare.com/proguard`

[3] `https://www.guardsquare.com/dexguard`

[4] `https://dexprotector.com/`

[5] `https://jfxstore.com/stringer/`

[6] `https://www.preemptive.com/solutions/android-obfuscation`

[7] `http://shield4j.com/main/sidecc63dd15b9656e1c877087e024689646fc27b38`

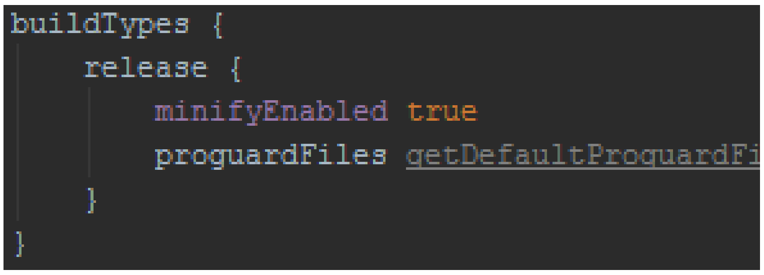
[8] `http://www.allatori.com/features/android-obfuscation.html`

Obfuscation Tool	**Cost**	**Protection Provided***
Proguard[2]	Free	Low
Dexguard[3]	Paid	High
DexProtector[4]	Free & Paid	Medium to High
Stringer[5]	Free & Paid	Medium to High
DashO[6]	Paid	Medium
Shield4J[7]	Paid	Medium
Allatori[8]	Paid	Medium

Table 4.1: Android obfuscation tools.

*The amount of protection provided is based on the obfuscation techniques used, as advertised by the tools.

the *minifyEnabled* flag to *true* in the build script, as shown in Figure 4.3.



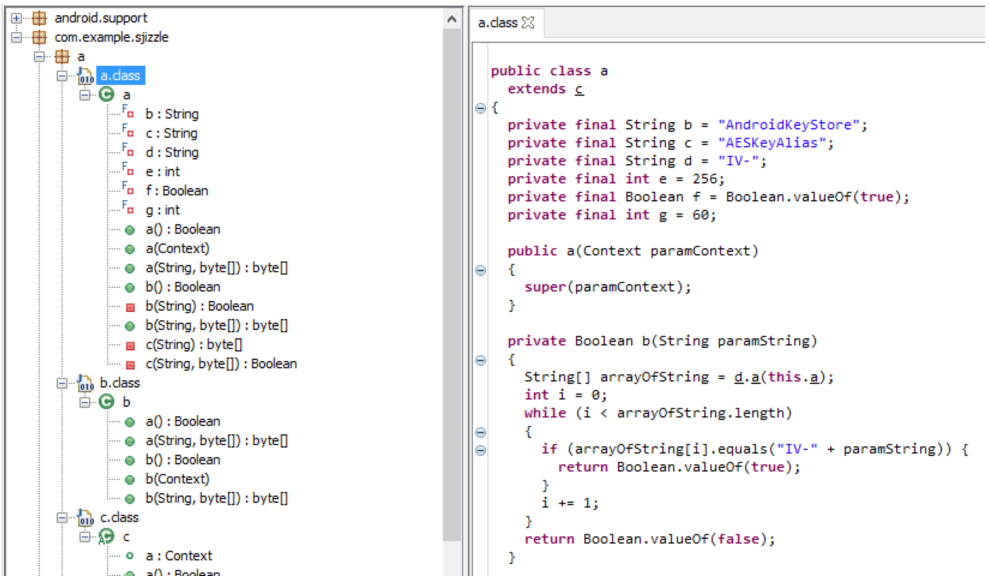
```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFi
    }
}
```

Figure 4.3: Enabling ProGuard obfuscation in Android Studio.

4.1.3 Analysis

Obfuscation is a technique that all developers should use when deploying applications. It is extremely trivial to make use of ProGuard as it is integrated into the build process but it is still not used often enough. This is most likely due to the fact that it is disabled by default and the fact that stack traces come in obfuscated form.

The protection offered provided by ProGuard is quite low as only lexical transformations are applied. An example Proguard obfuscation on a larger application is shown in Figure 4.4, classes, methods and variables have been renamed, but strings remain as plaintext and the code is relatively easy to follow due to many standard library calls. The reverse engineering process will be slowed down, but it will not stop an attacker from understanding the application given enough time.



```
android.support
com.example.sjizzle
  a
    a.class
      a
        b : String
        c : String
        d : String
        e : int
        f : Boolean
        g : int
        a() : Boolean
        a(Context)
        a(String, byte[]) : byte[]
        b() : Boolean
        b(String) : Boolean
        b(String, byte[]) : byte[]
        c(String) : byte[]
        c(String, byte[]) : Boolean
    b.class
      b
        a() : Boolean
        a(String, byte[]) : byte[]
        b() : Boolean
        b(Context)
        b(String, byte[]) : byte[]
    c.class
      c
        a : Context
```

```
public class a
  extends c
{
  private final String b = "AndroidKeyStore";
  private final String c = "AESKeyAlias";
  private final String d = "IV-";
  private final int e = 256;
  private final Boolean f = Boolean.valueOf(true);
  private final int g = 60;

  public a(Context paramContext)
  {
    super(paramContext);
  }

  private Boolean b(String paramString)
  {
    String[] arrayOfString = d.a(this.a);
    int i = 0;
    while (i < arrayOfString.length)
    {
      if (arrayOfString[i].equals("IV-" + paramString)) {
        return Boolean.valueOf(true);
      }
      i += 1;
    }
    return Boolean.valueOf(false);
  }
```

Figure 4.4: Code obfuscated with ProGuard.

An obfuscator that only provides lexical transformations is better than no obfuscator at all, but developers that need to create secure applications have to go an extra step further. A commercial obfuscator can be used to provide more protection. Tools such as DexGuard make use of lexical, data and control transformations to provide a more secure solution, however, licences need to be purchased in order to use these tools.

An alternative option to obfuscation is the use of native code. As mentioned before, high level languages are easier to decompile than low level ones. Developers can implement security sensitive parts of the code in C or C++ and call these functions from within the application. The result of this will be code that is harder to reverse engineer but doing this will add extra complexity to the code.

Obfuscation is a great way to obstruct attackers from reverse engineering applications, it makes it more difficult for attackers to use Static and Dynamic analysis (Sections 3.2.1 and 3.2.2 respectively). However, if an attacker is determined enough, it is inevitable that they will be able to get past obfuscation set in place by obfuscation tools or through native code.

4.2 Tamper Detection

Tamper detection is a technique used to detect whether or not an application has been changed by an attacker. In relation to the attack lifecycle it applies to the exploitation phase. If an application has been modified by an attacker (to bypass security features or extract sensitive data) it could be possible to detect the changes made to the code using tamper detection. If tampering has been detected, the developer can choose the best course of action, such as corrupting or deleting sensitive data. Tampering can be detected using the techniques described in the sections below.

4.2.1 Certificate Checking

4.2.1.1 Description

Android applications that are released to the public have to be signed with a certificate. Developers have to generate a self-signed certificate and sign their application with it. The private key of the certificate is used to determine the owner of the application, therefore anyone with the private key of an application is able to publish updates and make changes to an application that has been released.

When an application is signed, a digital signature is generated for the application using the private key [53, 54]. This signature can be validated using the corresponding public key. If an application has been changed, then the signature validation will fail and the application cannot be installed on the device. This is used to detect application tampering or corruption. In order to tamper with applications, attackers need to resign the application with their own certificate to create a valid digital signature. This results in the application having a different public key associated with it, which is the basis of this tamper detection method.

The modified application has a different public key than the original application. It is possible to get an application's public key using inbuilt libraries, this allows for a comparison between the original public key used and the current key used. If the attacker has resigned the application, it will be possible to detect this change. This allows for the detection of resigning, which indicates application tampering. We refer to this process as Certificate Checking.

4.2.1.2 Implementation

In order to implement Certificate Checking it is required that the public key of the signed application, is stored somewhere in the application. The simplest way to get the public key is to perform a release build of the application and print out the public key in logcat[9]. This can be done using the code in Listing A.9. The resulting string should be stored in the application and compared to the signature retrieved at runtime, as shown in Listing A.10. Implementing this tamper detection technique is very easy and does not require much effort on the developers side.

4.2.1.3 Analysis

The impact of this technique on performance is negligible and is very easy to implement. However, this technique can be easy to bypass. Since the

[9]Logging service for Android applications

public key is hardcoded in the application it can easily be found using static analysis and changed to a different value. Therefore it should be used in conjunction with other techniques such as obfuscation and tamper detection via Checksums. This makes it more difficult for attackers to simply change the value and requires more effort as they have to bypass more than one technique. The technique is used to protect against the exploitation phase of the attack life cycle as shown in Section 3.3.

4.2.2 Installer Verification

4.2.2.1 Description

The majority of Android applications are distributed using the Google Play Store[10]. If a developer releases their application through the Play Store, they expect the application to be distributed from there as well. It is possible for developers to check the name of the installation source. If an attacker tampers with an application they often install it on the device through ADB, which results in a different installer name being registered with the application. The expected installer name can be compared to the actual installer name to determine if the application has been installed from the right source [55]. This technique is referred to as Installer Verification.

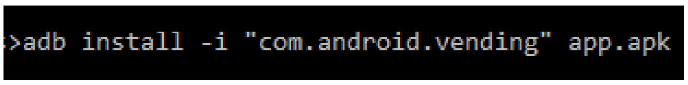
4.2.2.2 Implementation

Listing A.11 shows how to retrieve the name of the installer. Listing A.12 shows the verification process for distributions via the Google Play Store. If an application has been installed via ADB, the returned value is *null*, distributions via the Play Store return *com.android.vending*. The implementation of this technique is quite similar to that of Certificate Checking.

[10]`https://play.google.com/store/apps?hl=en`

4.2.2.3 Analysis

The impact on performance is negligible and is relatively easy to implement. However just like Certificate Checking it is very easy to bypass as an attacker. This technique relies on the fact that installing the APK through ADB produces *null* as the installer package name. However it is possible to specify the installer package name when using ADB to install an application. This is shown in Figure 4.5. Setting the installer package name to the same as the Google Play Store negates this technique. Even though the technique can be easily bypassed it will still be useful in protecting the application against tampering (Section 3.3).



```
>adb install -i "com.android.vending" app.apk
```

Figure 4.5: Specify installer package name with ADB.

4.2.3 Checksumming

4.2.3.1 Description

A checksum is a type of unique signature calculated over data, if the data changes by one bit the signature produced will change. This is often used to check the integrity of data to make sure the data has not been corrupted or changed. There are many algorithms available to calculate these signatures such as MD5, SHA-1, Adler-32, CRC32, etc.

Tamper detection can be added by calculating checksums over application code [56]. Checksums can be calculated when the application is in a known good state, the signature produced can be stored in the application. When the application is running on the user device, the checksum can be calculated and verified against the stored signature. If the signatures match then the code remains unchanged and therefore has not been tampered with. If the signatures do not match, then it is very likely that an attacker has changed the code or that the application has been corrupted.

Incorporating this into an application relies on a few different things:

- Developers need to be able to access the application code at runtime, in order to perform a checksum of the code.
- At least one checksum signature need to be stored somewhere in the application that is not being protected by a checksum.

4.2.3.2 Implementation

There is a lot more involved when using checksums as tamper detection compared to Certificate Checking and Installer Verification. To begin with, it needs to be determined what parts of the code should be protected. This is very hard to do since all of the code in an Android application gets bundled into a single *classes.dex*[11] file. It is quite difficult to understand how classes are structured within a file which means it is quite difficult to target specific pieces of code. Therefore the tamper detection will be applied over the entire DEX file.

Storage of the checksum signature is quite an important aspect of this technique. A checksum cannot be calculated over the data storing the signature. Figure 4.6 shows the problem. Storing the signature generated by the checksum back into the file causes the signature of the file to change, which creates a loop. Therefore, the signature will be stored in a different file.

To create the checksum for the *classes.dex* file, a release build of the application will need to be installed on the device. The code running on the application needs to extract the *classes.dex* file from the *.apk* located in

[11]A Dalvik Executable (DEX) file contains all the code for an application

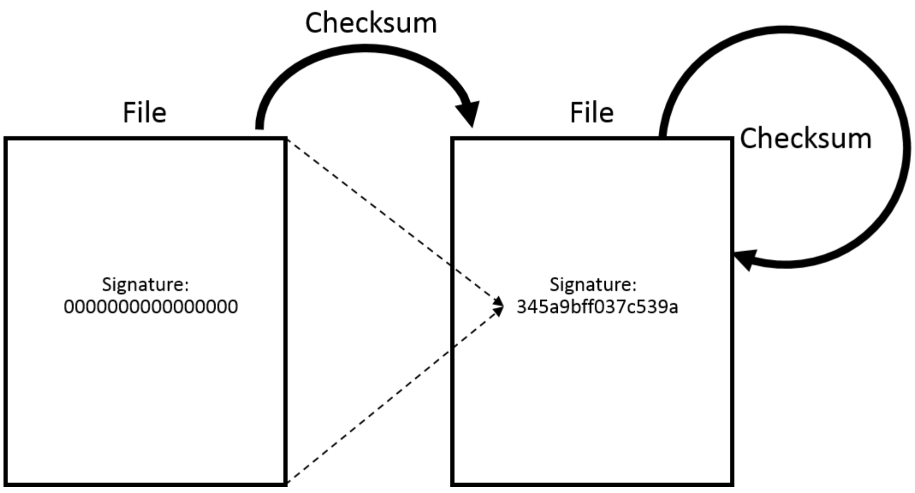


Figure 4.6: Storing checksum signature problem.

/data/app/"package name"/, as shown in Listing A.13. The checksum signature can then be calculated using the code shown in Listing A.14. The resulting signature is included in a file stored in the assets or resources folder of the application. To check the integrity of the application, developers need to compare the stored signature, with one calculated at runtime. If the signatures match it indicates that application code has not been modified, else it is highly likely that the application has been modified.

4.2.3.3 Analysis

Performance will be slightly affected through using this technique. As shown in Figure 4.7, the bigger the size of the *.dex* file the longer it will take to extract the file. The time it takes to perform the checksum function using different algorithms is also shown, however the performance between the two algorithms are very similar. In the scheme of things, performance is not such a big issue as the check only has to run once when the application starts.

It is relatively easy to implement for a developer, but also easy to circumvent for an attacker. The attacker has to modify the application, calculate the new signature and replace the stored signature. Developers can encrypt the file

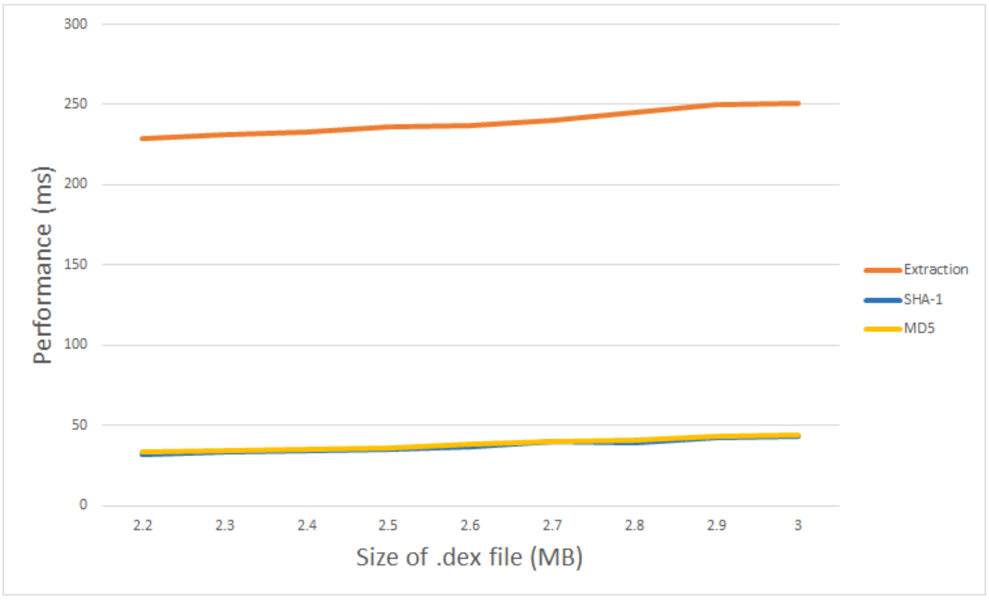


Figure 4.7: Performance impact of tamper detection functions in relation to the size of .dex file (device: Nexus 7 2012 WiFi tablet).

to make it more difficult to replace the old signature, however this will only slow down attackers, as the decryption code will be located somewhere in the application.

Developers can use multiple checksums over the same file to create more secure protection [57]. However, this requires much more effort on the part of the developer. The more effort a developer puts into this technique the more difficult it will be for attackers to disable the checksums, however attacker that are sufficient at tampering with applications will always be able to get past this method. Using this technique in conjunction with other security mechanisms will prove much more useful than spending a long time implementing tamper detection using multiple checksums. This technique will guard against application tampering, which is part of the exploitation phase of the attack life cycle (Section 3.3).

4.3 Root Detection

The first step an attacker often takes is to install a target application on a rooted device, this makes it much easier for them to inspect any data saved by the application. In order to protect user data it is beneficial that the application does not work properly or does not run at all on a rooted device, therefore Rooting Detection is needed.

4.3.1 Kernel modification

4.3.1.1 Description

If a device is rooted it is an indication that the device has been compromised. The method mentioned here does not directly check if a device has been rooted, but if it the kernel has been compromised. The kernel is signed with a key belonging to android to verify the integrity of the kernel. The build tag associated with the officially signed kernel is *release-keys*. If the kernel is modified by a user it needs to be signed with their own key and the associated build tag is *test-keys*. It is possible to to check for the kernel build tag which is outlined below.

4.3.1.2 Implementation

Listing A.15 shows how to get the kernel build tag and compare them to the expected values. If the kernel is compromised it is just as bad as having the device rooted, therefore we indicate that rooting has occurred.

4.3.1.3 Analysis

This method does not detect rooting, but does detect if the device has been compromised. Both rooting and kernel modifications are an indication that a device has been compromised, which is the reason for checking the kernel. However, a rooted device can still have a normal kernel build, so this method will not detect rooted devices. Even though rooting is not directly detected this technique can be used to protect against the extraction phase of the attack life cycle 3.1.

4.3.2 Su binary

4.3.2.1 Description

Rooted devices often have the su binary installed in the file system to give applications the ability to get root permissions. If the binary is present on a device, it is a good indication that a device is rooted. The binary itself can be stored in many different locations on the file system, so many places have to be checked.

Checking for the su binary. If the su binary is present on a device it is quite likely that it is rooted. Listing A.16 shows how to check for the binary.

4.3.2.2 Implementation

Listing A.16 can be used to check for the su binary. The binary can be stored anywhere on the device which makes it difficult to locate, but common areas are checked along with a few other files that might be present on a rooted device.

4.3.2.3 Analysis

If the su binary is not in the locations checked it is highly likely that the device is not rooted, but this might not always be the case. An attacker has the ability to place the binary anywhere on the file system which means that this method can easily be avoided by storing the binary in an unexpected location. This technique is used to protect against the extraction phase of the attack life cycle 3.1.

4.3.3 Command execution

4.3.3.1 Description

Root can be detected by trying to run a root level command on the device and seeing if it is successful.

4.3.3.2 Implementation

Listing A.17 tries to execute a command using root privileges. The command being executed is trying to list files in a protected section of the file system, that normal applications should not have access to. On a non-rooted device, an exception will be thrown with regards to denied permissions. If this occurs it is assumed that the device is not rooted. If an exception is not thrown, we assume that the device does have root privileges.

4.3.3.3 Analysis

This method of detecting root is quite good. It does not check for specific files so it does not suffer from the same flaw as the previous method mentioned and protects against the extraction phase of the attack life cycle 3.1. However, attackers can use tampering to remove the check from the application. It is also possible to move the su binary to a place not specified in the path environment variables, which means that the su command cannot be found and therefore root detection can be bypassed.

4.3.4 Rooting Applications

4.3.4.1 Description

Root detection is implemented in this method by looking for applications that are commonly installed on rooted devices. These applications are installed on the device by the program used to gain root.

4.3.4.2 Implementation

Listing A.18 checks for the package names of applications associated with rooting. If an application is detected, it is assumed that the device is rooted.

4.3.4.3 Analysis

Even if applications associated with rooting are installed on a device, it does not necessarily mean that the device is rooted. This technique may suffer from false positives, which could prevent legitimate users from using the application. However, this technique can help protect against the extraction phase of the attack life cycle 3.1.

4.4 Debugging Detection

In the analysis phase of the attack life cycle (Section 3.2.2), dynamic analysis is performed to gather information during application execution, attackers often use debugging techniques to perform this. Debugging is a very powerful method as it allows attackers to inspect specific values in the code that cannot always be retrieved through static analysis. Therefore debugging detection can be used as a countermeasure to detect dynamic analysis.

4.4.1 Execution Timing

4.4.1.1 Description

When an application is being debugged, it is often the case that certain parts of the code are being executed at a much slower rate than normal. Attacker often add breakpoints to the code in order to step through execution, which results in slower execution. Functions can be put in place to measure the execution time of sensitive parts of the code, allowing for the detection of debugging.

4.4.1.2 Implementation

Listing A.19 shows an example. The time before and after a sensitive function is recorded. If the execution of this function takes a lot longer than it normally would then it is possible that the code is being looked at by an attacker. To implement this properly, timing tests would have to be performed by the developer to get a good indication of the time it takes to execute the function under normal circumstances.

4.4.1.3 Analysis

The method itself is not very difficult to implement and gives no significant performance overhead. The more checks that are implemented the more time consuming for developers and the more it starts to affect performance. Also, the more checks in the code, the more time consuming it will be for attackers to remove them all.

Devices have different performance capabilities, which means it can become difficult for developers to distinguish between applications that are just running on a slow device and applications that are being debugged. This can lead to false positives which is not ideal.

4.4.2 Debugger Connection

4.4.2.1 Description

When an application is being debugged, a connection with a debugging service is created. Developers have the ability to query this connection to see if it is available or has been established. This can be used to determine if an application is currently being debugged.

4.4.2.2 Implementation

Listing A.20 is used to query the *Debug* class in order to determine if the debugger has already been connected. If it is, then the application is being debugged. Listing A.21 waits one second for a debugger to attach, if a debugger attaches during that time then the application is being debugged. If a debugger does not attach within a second then it is assumed the application is not being debugged.

4.4.2.3 Analysis

Both methods are quite reliable and easy to implement. The method that uses the blocking operation takes little over a second to complete which is quite a long time. It is not necessary to have both of these methods implemented, therefore the non blocking method seems like the better option to use.

4.4.3 Debugging Flag

4.4.3.1 Description

To make an application debuggable, attackers often tamper with the manifest and set the debuggable flag to *true* as shown in Section 3.2.2.1. Developers have the ability to check manifest settings, which means it is possible to check if the application debuggable flag has been changed.

4.4.3.2 Implementation

Listing A.22 shows how the value of the debugging flag can be determined. If the flag is set to 0 (false) the application is not debuggable, if the flag is set to something other than 0, such as 1 (true) it means that debugging is enabled.

4.4.3.3 Analysis

This method is quite reliable and is easy to implement, however it would be quite easy for attacker to use tampering to circumvent this technique. If used in conjunction with other security techniques it can prove very useful.

4.5 Emulator Detection

4.5.1 Description

As mentioned in Section 3.1, emulators are used to extract data from applications. Detecting if an application is running inside an emulator is a valuable tool to prevent sensitive data from being leaked.

There are many ways to detect if an emulator is running, the virtual environment will have different hardware properties compared to real devices as described in [58]. The performance capabilities of the emulator also differ along with network configurations. The method listed below only looks at the environment variables to detect emulation as it suffices to detect most emulators.

4.5.2 Implementation

In [58] many environmental variables are listed along with values commonly associated with emulators. This has been adapted into the code in Listing A.23. There are many emulators, each with slight differences in environmental variables. The detection method will therefore not not always work on all emulators, but most standard emulators will be detected.

4.5.3 Analysis

There are many legitimate use cases for running an application on an emulator, this means that it is not always ideal to use this method. However, it might be a good idea to check for emulation just before handling sensitive information. The detection itself is quite consistent but it might suffer from false positives as it is not feasible to take into account all devices.

4.6 Sanitization

4.6.1 Description

As applications execute, objects are constantly being created and destroyed. These objects are loaded into memory when used and freed from memory when they are not needed anymore. As shown in Section 3.2.2.2 memory dumps can be taken, which results in application memory being saved. Objects that were in memory at that time can now be analysed by attackers.

This cannot be fully avoided, if attackers take a memory dump at the right time, sensitive information can be extracted. However, we can make it more difficult for attackers to collect sensitive data from memory through the use of sanitization. Sanitization is a process used to clean up objects, this can be done by writing garbage data to the memory locations of the objects that we want to sanitize. This makes sure that the data is erased properly and that developers do not have to rely on garbage collection. Even if garbage collection occurs the data can still hang around in memory until it is overwritten by another process. Making sure that any sensitive information is sanitized properly will make it much more difficult for attacker to get a hold of the data, as the time window they have to capture the data decreases immensely.

4.6.2 Implementation

In practice sanitization is quite simple, an example is shown in Listing A.24. Since Java uses pass-by-value for method arguments, some primitive values are passed in as arrays with one element. Sanitization should be used on objects containing sensitive data immediately after they are not needed anymore, this ensures that the data contained in those objects is disposed of as quickly as possible.

4.6.3 Analysis

Sanitization should be used by all developers handling sensitive data. Performance is not affected, it is easy to implement and protects against memory inspection. Attackers that rely on memory dumps will have a much more difficult time extracting sensitive data if sanitization is employed.

Chapter 5

Recommendations

In this chapter of the thesis recommendations are given as to which security techniques should be used depending on the required security level of the application.

Obfuscation.

All applications should use obfuscation. For low security applications it is sufficient to use a free tool such as ProGuard to add lexical transformations (Section 4.1.1.1) to the code. Applications that require moderate security should add data transformations as well as lexical transformations. High security applications should use lexical, data (Section 4.1.1.3) and control (Section 4.1.1.2) transformations as well as anti-decompilation (Section 4.1.1.4) methods. For the high security applications it is recommended to use a paid obfuscator such as DexGuard, which will apply all of the mentioned obfuscation techniques. If it is not possible to use an obfuscator, use native code to implement the security sensitive parts of the applications.

Tamper Detection.

Some form of tamper detection should be applied to applications. For low security applications, it is sufficient to use certificate checking (Section 4.2.1) and installer verification (Section 4.2.2), both of these techniques are easy to implement and quite effective at detecting tampering. Medium and high security applications should use the low security requirements with the addition of the checksumming (Section 4.2.3) technique. As more checksums are added into the application it becomes more difficult for attackers to bypass, therefore the more effort developers put into this technique the better.

Root Detection.

Rooting detection should be used by all applications. Low, medium and high security applications should use all of the mentioned rooting detection techniques (Sections 4.3.1, 4.3.2, 4.3.3 and 4.3.4). The techniques are very easy to implement and can just be copied from the provided implementation examples. However, developers need to be aware that including root detection can reduce the amount of users for their application, as there are legitimate uses for device rooting. Therefore, this technique should only be included if the target application needs to be protected.

Debugging Detection

Debugging detection should be used by all applications. For low security applications it will be sufficient to check for the state of the debugging flag (Section 4.4.3) in the manifest. Medium and high security applications should additionally check for an active connection to a debugger (Section 4.4.2). The execution timing debugging (Section 4.4.1) technique can additionally be used in medium and high security applications but developers need to take care when implementing it since false positives could occur, which could prevent legitimate users from using the application.

Emulator Detection.

Emulator detection (Section 4.5) should be used at the discretion of the developer, there are many legitimate scenarios in which an application is run on an emulator. Therefore it might not always be a good idea to use emulator detection, it depends on the nature of the application. If the developer has decided that the application should never be run on an emulator, then low, medium and high security applications should use emulator detection. It is very easy to implement, the supplied implementation examples can be used for this.

Sanitization.

Sanitization (Section 4.6) should be used in applications that require medium to high security. Low security applications do not need to include it, since the attack it is used to counter, is relatively high level.

Conclusion.

When used individually, the techniques presented do not provide much protection and can be circumvented relatively easily by attackers. Using many of these techniques in combination will boost the security of an application and discourage attackers. Ideally developers should include all the techniques mentioned into their applications, however this is not always possible. Table 5.1 shows which of the covered techniques should be used in relation to the desired security level of applications.

Security Techniques	**Security Level**		
	Low	**Medium**	**High**
Obfuscation			
Lexical Transformations	Yes	Yes	Yes
Data Transformations		Yes	Yes
Control Transformations			Yes
Anti-Decompilation			Yes
Tamper Detection			
Certificate Checking	Yes	Yes	Yes
Installer Verification	Yes	Yes	Yes
Checksumming		Yes	Yes
Root Detection			
Kernel Modification	Yes*	Yes*	Yes*
Su Binary Detection	Yes*	Yes*	Yes*
Command Execution	Yes*	Yes*	Yes*
Rooting Applications	Yes*	Yes*	Yes*
Debugging Detection			
Debugging Flag	Yes	Yes	Yes
Debugger Connection		Yes	Yes
Execution Timing		Yes*	Yes*
Emulator Detection	Yes*	Yes*	Yes*
Sanitization		Yes	Yes

Table 5.1: The recommended security techniques that should be implemented based on the level of security required.

* Should only be included if the developer has determined that the techniques do not negatively impact the application.

Chapter 6

Conclusion

6.1 Summary

Protecting sensitive application data involves a lot more than just cryptography. All parts of the application that interact with the sensitive data need to be protected. It is possible to protect application code and sensitive data through special hardware on devices, however it is not feasible to use the hardware in many circumstances. Therefore applications need to employ software protection techniques such as obfuscation, tamper detection, root detection, debug detection, emulator detection and sanitization. The challenge developers face is the apparent lack of resources to facilitate the implementation of these techniques. The research provided explores the capabilities of attackers and gives implementation details on how to guard against these attacks.

6.2 Contributions

The research presented in this thesis provides developers with a guide that can be referred to when implementing application security. There are other resources out there that aim to provide similar services, however information about security techniques and implementation details are scarce. This guideline gives developers without a security background a resource they can refer to, in order to create more security aware applications on the Android platform.

This research was done in part for the Gallagher Group[1]. The organization has had an avid interest in mobile security, which was the inspiration behind this research topic. The result of this research will help aid in the development of secure mobile applications within the organization.

6.3 Future Work

The initial scope of the research intended to cover many different mobile platforms, not just Android. As more research was done in the area, the more it became apparent it was not feasible to cover more than one mobile platform. Possible future work could include performing similar research but for other mobile platforms such as iOS or Windows Mobile.

More research needs to be performed in the area of hardware based solutions. Hardware based solutions seem to be the most secure way of storing and processing application data, however few applications can make use of this. Research in this area can include how developers can leverage the use of these hardware modules to create more secure applications.

Another interesting area of research includes a more in depth look at tamper detection. Providing reliable tamper detection capabilities to applications can be of great use to developers wanting to create secure applications. However this is difficult in software only implementations, therefore more research can be performed in software based tamper detection.

[1]`https://www.gallagher.com/`

References

[1] IDC, "Smartphone OS market share." http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2016. [Accessed 2016-2-9].

[2] Android. http://developer.android.com/guide/topics/data/data-storage.html, 2016. [Accessed 2016-2-9].

[3] S. N. I. Association, "Solutions guide for data-at-rest," 2001.

[4] GlobalPlatform, "Globalplatform made simple guide: Secure element." https://www.globalplatform.org/mediaguideSE.asp. [Accessed 2016-2-10].

[5] Globalplatform, "Globalplatform made simple guide: Trusted execution environment (TEE) guide." http://www.globalplatform.org/mediaguidetee.asp. [Accessed 2016-2-10].

[6] N. Elenkov, "Using the SIM card as a secure element in Android." http://nelenkov.blogspot.co.nz/2013/09/using-sim-card-as-secure-element.html, 2013. [Accessed 2016-2-9].

[7] ISO/IEC, "ISO/IEC 7816-4 identification cards - integrated circuit cards - part 4: Organization, security and commands for interchange," 2013.

[8] Oracle, "Java card overview." http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html, 2016. [Accessed

2016-2-9].

[9] SDcard, "smartSD." https://www.sdcard.org/developers/overview/ASSD/smartsd/. [Accessed 2016-2-10].

[10] SIMalliance, "Open mobile API specification v3.0." http://simalliance.org/wp-content/uploads/2015/03/SIMalliance_OpenMobileAPI3_0_release1_FINAL3.pdf, 2014. [Accessed 2016-2-9].

[11] SIMalliance, "Simalliance — security / identity / mobility." http://simalliance.org/, 2016. [Accessed 2016-2-9].

[12] SEEK, "SEEK for Android." http://seek-for-android.github.io/, 2016. [Accessed 2016-2-9].

[13] Infineon, "Infineons embedded secure element brings security to NFC enabled smart phones." http://www.infineon.com/cms/en/about-infineon/press/press-releases/2011/INFCCS201105-048.html, 2011. [Accessed 2016-3-15].

[14] N. Elenkov, "Accessing the embedded secure element in Android 4.x." http://nelenkov.blogspot.co.nz/2012/08/accessing-embedded-secure-element-in.html, 2012. [Accessed 2016-2-9].

[15] Semble, "Semble." http://www.semble.co.nz/, 2016. [Accessed 2016-2-9].

[16] ARM, "Development of TEE and secure monitor code." http://www.arm.com/products/processors/technologies/trustzone/tee-smc.php. [Accessed 2016-2-10].

[17] Solacia, "SecuriTEE." http://www.sola-cia.com/en/securiTee/product.asp, 2016. [Accessed 2016-2-9].

[18] Trustonic, “Trusted Executed Environment (TEE) — Trustonic.” https://www.trustonic.com/technology/trusted-execution-environment, 2016. [Accessed 2016-2-9].

[19] Qualcomm, “Snapdragon mobile security for embedded devices, biometric authentication and safeswitch technology.” https://www.qualcomm.com/products/snapdragon/security, 2014. [Accessed 2016-2-9].

[20] A. C. Community, “Trustzone.” https://community.arm.com/thread/7845, 2016. [Accessed 2016-2-9].

[21] GlobalPlatform, “The trusted execution environment:delivering enhanced security at a lower cost to the mobile market.” http://www.globalplatform.org/documents/whitepapers/GlobalPlatform_TEE_Whitepaper_2015.pdf, 2015. [Accessed 2016-2-10].

[22] G. Association, “Embedded UICC protection profile.” https://www.commoncriteriaportal.org/files/ppfiles/pp0089b_pdf.pdf, 2015. [Accessed 2016-2-9].

[23] GlobalPlatform, “TEE protection profile.” https://www.commoncriteriaportal.org/files/ppfiles/anssi-profil_PP-2014_01.pdf, 2014. [Accessed 2016-2-9].

[24] Android, “Android keystore system.” http://developer.android.com/training/articles/keystore.html#ExtractionPrevention, 2016. [Accessed 2016-2-9].

[25] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, “Cloud computing: Distributed internet computing for IT and scientific research,” *Internet Computing, IEEE*, vol. 13, no. 5, pp. 10–13, 2009.

[26] Android, “Security — Android open source project.” https://source.android.com/devices/tech/security/, 2016. [Accessed 2016-2-9].

[27] Android, “Working with system permissions — Android developers.” http://developer.android.com/training/permissions/index.html, 2016. [Accessed 2016-2-9].

[28] Android, “Application fundamentals — Android developers.” http://developer.android.com/guide/components/fundamentals.html, 2016. [Accessed 2016-2-9].

[29] Android, “Android keystore system — Android developers.” http://developer.android.com/training/articles/keystore.html, 2016. [Accessed 2016-2-9].

[30] N. Elenkov, “Keystore redesign in Android m.” http://nelenkov.blogspot.co.nz/2015/06/keystore-redesign-in-android-m.html, 2015. [Accessed 2016-2-9].

[31] N. Elenkov, “Credential storage enhancements in Android 4.3.” http://nelenkov.blogspot.co.nz/2013/08/credential-storage-enhancements-android-43.html, 2013. [Accessed 2016-2-9].

[32] OWASP, “Top ten mobile risks.” https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks. [Accessed 2016-2-9].

[33] OWASP, “OWASP mobile top ten 2015 data synthesis and key trends.” https://www.owasp.org/images/9/9e/2015_Data_Synthesis_Results.pptx, 2015. [Accessed 2016-2-9].

[34] OWASP, “Technical risks of reverse engineering and unauthorized code modification.” https://www.owasp.org/index.php/Technical_Risks_

of_Reverse_Engineering_and_Unauthorized_Code_Modification, 2014. [Accessed 2016-2-9].

[35] OWASP, "Architectural principles that prevent code modification or reverse engineering." https://www.owasp.org/index.php/Architectural_Principles_That_Prevent_Code_Modification_or_Reverse_Engineering, 2014. [Accessed 2016-2-9].

[36] M. Souppaya and K. Scarfone, "Guidelines for managing the security of mobile devices in the enterprise," *NIST special publication*, vol. 800, p. 124, 2013.

[37] S. Quirolgico, J. Voas, T. Karygiannis, C. Michael, and K. Scarfone, "Vetting the security of mobile applications," *NIST special publication*, vol. 800, p. 163, 2015.

[38] XDA, "Bootloader." http://forum.xda-developers.com/wiki/Bootloader. [Accessed 2016-2-9].

[39] pxb1988, "dex2jar." https://github.com/pxb1988/dex2jar. [Accessed 2016-2-14].

[40] J. Decompiler, "JD-GUI." http://jd.benow.ca/. [Accessed 2016-2-14].

[41] E. Gruber, "Attacking Android applications with debuggers." https://blog.netspi.com/attacking-android-applications-with-debuggers/, 2015. [Accessed 2016-2-16].

[42] P. Teoh, "How to dump memory of any running processes in Android (rooted)." https://tthtlc.wordpress.com/2011/12/10/how-to-dump-memory-of-any-running-processes-in-android-2/, 2011. [Accessed 2016-2-16].

[43] D. Lodge, “How to extract sensitive plaintext data from Android memory.” `https://www.pentestpartners.com/blog/how-to-extract-sensitive-plaintext-data-from-android-memory/`, 2015. [Accessed 2016-2-16].

[44] N. A. Naeem, M. Batchelder, and L. Hendren, “Metrics for measuring the effectiveness of decompilers and obfuscators,” in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pp. 253–258, IEEE, 2007.

[45] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[46] C. S. Collberg and C. Thomborson, “Watermarking, tamper-proofing, and obfuscation-tools for software protection,” *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 735–746, 2002.

[47] M. Batchelder and L. Hendren, “Obfuscating java: the most pain for the least gain,” in *Compiler Construction*, pp. 96–110, Springer, 2007.

[48] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly,” in *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 290–299, ACM, 2003.

[49] H. Lai, “A comparative survey of java obfuscators available on the internet,” *Project Report, University of Auckland*, 2001.

[50] D. Leskov, “Protect your java code through obfuscators and beyond.” `http://www.excelsior-usa.com/articles/java-obfuscators.html`. [Accessed 2016-2-17].

[51] C. Collberg, C. Thomborson, and D. Low, “Manufacturing cheap, resilient, and stealthy opaque constructs,” in *Proceedings of the 25th ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 184–196, ACM, 1998.

[52] Android, "ProGuard." `http://developer.android.com/tools/help/proguard.html`. [Accessed 2016-2-18].

[53] Oracle, "jarsigner - JAR signing and verification tool." `https://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html`. [Accessed 2016-2-19].

[54] N. Elenkov, "Android code signing." `http://nelenkov.blogspot.co.nz/2013/04/android-code-signing.html`, 2013. [Accessed 2016-2-19].

[55] S. Alexander-Bown, "Android security: Adding tampering detection to your app." `https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app`. [Accessed 2016-2-19].

[56] T. Johns, "Securing Android LVL applications." `http://android-developers.blogspot.co.nz/2010/09/securing-android-lvl-applications.html`, 2010. [Accessed 2016-2-19].

[57] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Security and privacy in digital rights management*, pp. 160–175, Springer, 2001.

[58] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 447–458, ACM, 2014.

Appendix A

Code Listings

Listing A.1: Class with hardcoded values

```java
public class Secret {
    String password = "onetwothree";
    String username = "Bob";
    public Secret(){
        authenticateUser(username, password);
    }
    private Boolean authenticateUser(String username, String password){
        Boolean authenticated = false;
        //Perform authentication using the username and password
        return authenticated;
    }
}
```

Listing A.2: Class without a hardcoded value

```java
public class Secret {
    String password;
    String username = "Bob";
    public Secret(){
        password = retrieveSecretString();
        authenticateUser(username, password);
    }
    private String retrieveSecretString(){
        return new BigInteger(130, new SecureRandom()).toString(32);
    }
    private Boolean authenticateUser(String username, String password){
        Boolean authenticated = false;
        //Perform authentication using the username and password
        return authenticated;
    }
}
```

Listing A.3: Java code used for memory dump

```java
public class Secret
{
    public Secret()
    {
        authenticateUser();
    }
    private void authenticateUser()
    {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    //Generate a unique byte signature that we can find
                    byte[] dataToRetrieve = doubleData(new byte[]{0x22, 0x22,
                         0x22, 0x22, 0x33, 0x33, 0x33, 0x33});
                    //Block thread execution for easy memory dump capture
                    Thread.sleep(10000);
                }catch (InterruptedException ex){
                    ex.printStackTrace();
                }
            }
        });
        thread.start();
    }
    private byte[] doubleData(byte[] data)
    {
        byte[] dataDouble = new byte[data.length];
        for(int i = 0; i < data.length; i++)
            dataDouble[i] = (byte)(data[i] + data[i]);
        return dataDouble;
    }
}
```

Listing A.4: C code used for memory dump

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>
#include <limits.h>
#define MAX_FILENAME_CHARS 255

int main(int argc, char **argv) {
    if (argc == 2) {
        int pid = atoi(argv[1]);
        int c;
        int counter;
        char filename[MAX_FILENAME_CHARS];
        char line[MAX_FILENAME_CHARS];
        char mem_start[9];
        char mem_start_string[10] = {"0x"};
        char mem_end[9];
        FILE *file_pointer;
        unsigned long mem_start_long;
        unsigned long mem_end_long;
        unsigned long mem_length_long;
        unsigned long starting_address;
        int mem_length_int;
        char *ptr;

        memset(&filename[0], 0, sizeof(filename));
        memset(&line[0], 0, sizeof(line));
        strcat(filename, "/proc/");
        strcat(filename, argv[1]);
        strcat(filename, "/maps\0");
        file_pointer = fopen(filename, "r");
        if(file_pointer == NULL){
            printf("Could not open the file:'%s'\n", filename);
            return 0;
        }
        //Use ptrace to attach to the process
        ptrace(PTRACE_ATTACH, pid, NULL, NULL);
        wait(NULL);
        counter = 0;
        do{
            c = fgetc(file_pointer);
            if(feof(file_pointer)){
                break;
            }
            line[counter] = c;
            if(counter <= 17){
                if(counter < 8){
                    mem_start[counter] = c;
                }
                if(counter == 8){
                     mem_start[counter] = '\0';
                }
                if(counter > 8 && counter < 17){
                    mem_end[counter - 9] = c;
                }
                if(counter == 17){
                    mem_end[counter - 9] = '\0';
                }
            }
            counter++;
            if(c == '\n'){
                counter = 0;
                if(!(strstr(line, "[stack]") != NULL)){
                    strcat(mem_start_string, mem_start);
                    mem_start_long = strtoul(mem_start, &ptr, 16);
                    mem_end_long = strtoul(mem_end, &ptr, 16);
                    mem_length_long = mem_end_long - mem_start_long;
                    sscanf(mem_start_string, "0x%x", (unsigned int *)&
                        starting_address);
                    mem_length_long = mem_length_long/4;
                    //Print out the memory
                    dump_memory(pid, starting_address, mem_length_long);
                    mem_start_string[2] = '\0';
                }
                memset(&line[0], 0, sizeof(line));
            }
        }while(1);
        fclose(file_pointer);
        //Detach from the process
        ptrace(PTRACE_CONT, pid, NULL, NULL);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
      }
      else {
        printf("%s <pid> \n", argv[0]);
        exit(0);
      }
    }
    dump_memory(int pid, unsigned int start_address, unsigned long total_words)
         {
        unsigned int address;
        unsigned int number = 0;
        for (address=start_address;address<start_address+total_words*4;
            address+=4) {
                if (address%16==0) printf("\n");
                number=ptrace(PTRACE_PEEKDATA, pid, (void *)address, (void *)
                    number);
                printf("%x ", number);
        }
        if (total_words==0) {
                number=ptrace(PTRACE_PEEKDATA, pid, (void *)start_address, (
                    void *)number);
                printf("Peek at 0x%x: %x\n", (unsigned int)start_address,
                    number);
        }
    }
```

Listing A.5: Java code tamper demonstration

```
import android.util.Log;
public class Secret {
    private static String TAG="SecretClass";
    public Secret(){
        if(!securityCheck()){
            //Security check failed
            Log.i(TAG, "securityCheck failed!");
            return;
        }
        //Security check passed, continue normal execution
        Log.i(TAG, "securityCheck passed!");
    }
    private Boolean securityCheck(){
        Log.i(TAG, "In securityCheck");
        //Perform a security check
        //True is returned if the application is secure
        //False is returned if the application is not secure
        return false;
    }
}
```

Listing A.6: Smali code tamper demonstration

```
.class public Lcom/example/shoe/attacktampering/Secret;
.super Ljava/lang/Object;
.source "Secret.java"
# static fields
.field private static TAG:Ljava/lang/String;
# direct methods
.method static constructor <clinit>()V
    .locals 1
    .prologue
    .line 7
    const-string v0, "SecretClass"
    sput-object v0, Lcom/example/shoe/attacktampering/Secret;->TAG:Ljava/
        lang/String;
    return-void
.end method

.method public constructor <init>()V
    .locals 2
    .prologue
    .line 9
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    .line 10
    invoke-direct {p0}, Lcom/example/shoe/attacktampering/Secret;->
        securityCheck()Ljava/lang/Boolean;
    move-result-object v0
    invoke-virtual {v0}, Ljava/lang/Boolean;->booleanValue()Z
    move-result v0
    if-nez v0, :cond_0
    .line 12
    sget-object v0, Lcom/example/shoe/attacktampering/Secret;->TAG:Ljava/
        lang/String;
    const-string v1, "securityCheck failed!"
    invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/
        lang/String;)I
    .line 18
    :goto_0
    return-void
    .line 17
    :cond_0
    sget-object v0, Lcom/example/shoe/attacktampering/Secret;->TAG:Ljava/
        lang/String;
    const-string v1, "securityCheck passed!"
    invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/
        lang/String;)I
    goto :goto_0
.end method

.method private securityCheck()Ljava/lang/Boolean;
    .locals 2
    .prologue
    .line 21
    sget-object v0, Lcom/example/shoe/attacktampering/Secret;->TAG:Ljava/
        lang/String;
    const-string v1, "In securityCheck"
    invoke-static {v0, v1}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/
        lang/String;)I
    .line 25
    const/4 v0, 0x0
    invoke-static {v0}, Ljava/lang/Boolean;->valueOf(Z)Ljava/lang/Boolean;
    move-result-object v0
    return-object v0
.end method
```

Listing A.7: Opaque predicates

```
public class Secret {
    Random random = new Random();
    public Secret(){
        doNothing(1);
    }
    private int doNothing(int number){
        int x = random.nextInt(185/34);
        int y = random.nextInt(x + 1);
        //Opaque predicate that always return false
        if(!(Math.pow(x - y, 2) >= 0)){
            number = 20;
        }
        //Opaque predicate that always return true
        if(!(((54*y) - 3) == Math.pow(x, 2))){
            return number;
        }else{
            number = 31;
        }
        return number;
    }
}
```

Listing A.8: String encoding

```java
public class Secret {
    String string = "secretPassword123";
    String stringFromEncodedString = "sfeuiyVh{|" + Character.toString((char)0x81) + "z~q?AC";
    byte[] stringFromBytes = new byte[]{
            (byte)0x73, (byte)0x65, (byte)0x63, (byte)0x72,
            (byte)0x65, (byte)0x74, (byte)0x50, (byte)0x61,
            (byte)0x73, (byte)0x73, (byte)0x77, (byte)0x6f,
            (byte)0x72, (byte)0x64, (byte)0x31, (byte)0x32,
            (byte)0x33};
    public Secret(){
        Log.i("Secret:", string);
        Log.i("Secret:", stringDecode(stringFromEncodedString));
        Log.i("Secret:", byteToString(stringFromBytes));
    }
    private String stringEncode(String word){
        char[] characters = word.toCharArray();
        for(int i = 0; i < characters.length; i++)
            characters[i] = (char) ((int) characters[i] + i);
        return new String(characters);
    }
    private String stringDecode(String word){
        char[] characters = word.toCharArray();
        for(int i = 0; i < characters.length; i++)
            characters[i] = (char)((int)characters[i] - i);
        return new String(characters);
    }
    private String byteToString(byte[] word){
        return new String(word);
    }
}
```

Listing A.9: Retrieving public key

```
Signature[] signatures = context.getPackageManager().getPackageInfo(context
    .getPackageName(), PackageManager.GET_SIGNATURES).signatures;
for(int i = 0; i < signatures.length; i++){
   Log.i("Secret", "Certificate:" + signatures[i].toCharsString());
}
```

Listing A.10: Verifying public key

```
private Boolean verifyCertificate(String current, String stored){
   if(current.compareTo(stored) == 0)
      return true;
   else
      return false;
}
```

Listing A.11: Retrieving installer name

```
   private String retrieveInstallerName(){
      return context.getPackageManager().getInstallerPackageName(context.
          getPackageName());
   }
```

Listing A.12: Verifying installer name

```
private Boolean verifyInstaller(String installer){
   String playStoreName="com.android.vending";
   if(installer == null)
      return false;
   if(installer.compareTo(playStoreName) == 0)
      return true;
   return false;
}
```

Listing A.13: Extracting .dex file from APK

```java
private void extractDexFileIntoDirectory(File zipFile, File targetDirectory
    ){
    ZipInputStream zipInputStream;
    FileOutputStream fileOutputStream;
    try {
        zipInputStream = new ZipInputStream(new BufferedInputStream(new
            FileInputStream(zipFile)));
        ZipEntry zipEntry;
        int count;
        byte[] buffer = new byte[8192];
        while ((zipEntry = zipInputStream.getNextEntry()) != null) {
            if (zipEntry.getName().compareTo("classes.dex") == 0) {
                File file = new File(targetDirectory, zipEntry.getName());
                fileOutputStream = new FileOutputStream(file);
                while ((count = zipInputStream.read(buffer)) != -1)
                    fileOutputStream.write(buffer, 0, count);
                fileOutputStream.close();
            }
        }
        zipInputStream.close();
    }catch (FileNotFoundException ex){
        ex.printStackTrace();
    }catch (IOException ex){
        ex.printStackTrace();
    }
}
```

Listing A.14: Calculating checksum

```java
private byte[] calculateChecksum(String filePath, String algorithm){
    int byteCounter;
    byte[] buffer = new byte[8192];
    try{
        InputStream inputStream = new FileInputStream(filePath);
        //Algorithms: "MD5", "SHA-1", "SHA-256", etc.
        MessageDigest messageDigest = MessageDigest.getInstance(algorithm);
        while((byteCounter = inputStream.read(buffer)) != -1){
            messageDigest.update(buffer, 0 , byteCounter);
        }
        inputStream.close();
        return messageDigest.digest();
    }catch (Exception ex) {
        //Implement proper exception handling
        ex.printStackTrace();
    }
    return null;
}
```

Listing A.15: Root detection via Kernel modification

```
private boolean rootDetectionKernel() {
    String buildTag = android.os.Build.TAGS;
    if("release-keys".equals(buildTag))
        return false;
    if("test-keys".equals(buildTag))
        return true;
    return false;
}
```

Listing A.16: Root detection via su binary

```
private boolean rootDetectionSuBinary() {
    String[] paths = {
            "/sbin/su",
            "/system/su",
            "/system/bin/su",
            "/system/xbin/su",
            "/system/sd/xbin/su",
            "/system/bin/failsafe/su",
            "/data/local/su",
            "/data/local/bin/su",
            "/data/local/xbin/su",
            "/system/app/SuperSU",
            "/system/app/Superuser.apk"};
    for (String path : paths) {
        if (new File(path).exists()) return true;
    }
    return false;
}
```

Listing A.17: Root detection via su execution

```java
private boolean rootDetectionSuExecution() {
    Process process = null;
    try {
        process = Runtime.getRuntime().exec(new String[] { "su", "-c", "ls /
            data/app/" });
        return true;
    } catch (Exception ex) {
       //Expect to get here if device is not rooted
        return false;
    } finally {
        if (process != null)
            process.destroy();
    }
}
```

Listing A.18: Root detection via applications

```java
private boolean rootDetectionSUApplications(PackageManager packageManager){
    String[] applications = {
            "com.noshufou.android.su",
            "com.thirdparty.superuser",
            "eu.chainfire.superuser",
            "eu.chainfire.supersu",
            "com.koushikdutta.superuser",
            "com.zachspong.temprootremovejb",
            "com.ramdroid.appquarantine",
    };
    for(String app : applications){
        try{
            packageManager.getPackageInfo(app, PackageManager.GET_ACTIVITIES)
                ;
            return true;
        }catch (Exception ex){
            //Do nothing if we cannot find the package
        }
    }
    return false;
}
```

Listing A.19: Debug detection via execution time

```
private Boolean debugDetectionExecution(){
    long start = Debug.threadCpuTimeNanos();
    //Function that handles sensitive information
    decrypt();
    long stop = Debug.threadCpuTimeNanos();
    if(stop - start < 10000000)
        return false;
    else
        return true;
}
```

Listing A.20: Debug detection via debugger connection

```java
private Boolean debugDetectionDebuggerConnected(){
    if(Debug.isDebuggerConnected())
        return true;
    else
        return false;
}
```

Listing A.21: Debug detection via debugger attach

```java
private Boolean debugDetectionDebuggerAttach(){
    DebugDetectionThread thread = new DebugDetectionThread();
    thread.start();
    long start = Calendar.getInstance().getTimeInMillis() / 1000;
    long end;
    do {
        end = Calendar.getInstance().getTimeInMillis() / 1000;
        long duration = end - start;
        if(duration > 1)
            return false;
    }while (!thread.done);
    return true;
}
private class DebugDetectionThread extends Thread{
    public Boolean done = false;
    @Override
    public void run() {
        Debug.waitForDebugger();
        done = true;
    }
}
```

Listing A.22: Debug detection via debugging flag

```
private Boolean debugDetectionDebugFlag(Context context){
    if(0 == (context.getApplicationInfo().flags & ApplicationInfo.
        FLAG_DEBUGGABLE))
        return false;
    else
        return true;
}
```

Listing A.23: Emulator detection

```
private Boolean detectEmulation(Activity context){
    TelephonyManager telephonyManager = (TelephonyManager) context.
        getSystemService(Context.TELEPHONY_SERVICE);
    if("x86".equals(Build.CPU_ABI))
        return true;
    if("unknown".equals(Build.CPU_ABI) || "".equals(Build.CPU_ABI))
        return true;
    if("unknown".equals(Build.BOARD) || "Android".equals(Build.BOARD))
        return true;
    if("Android".equals(Build.BRAND) || "generic".equals(Build.BRAND))
        return true;
    if("vbox86p".equals(Build.DEVICE) || Build.DEVICE.contains("generic"))
        return true;
    if("vbox86".equals(Build.HARDWARE) || "goldfish".equals(Build.HARDWARE))
        return true;
    if("Genymotion".equals(Build.MANUFACTURER) || "unknown".equals(Build.
        MANUFACTURER))
        return true;
    if("test-keys".equals(Build.TAGS))
        return true;
    if("000000000000000".equals(telephonyManager.getDeviceId()))
        return true;
    if("15555215554".equals(telephonyManager.getLine1Number()))
        return true;
    if("us".equals(telephonyManager.getNetworkCountryIso()))
        return true;
    if("3".equals(Integer.toString(telephonyManager.getNetworkType())))
        return true;
    if("310260000000000".equals(telephonyManager.getSubscriberId()))
        return true;
    if("+15552175049".equals(telephonyManager.getVoiceMailNumber()))
        return true;
    return false;
}
```

Listing A.24: Sanatization

```
private void sanatizeBytes(byte[] data){
    for(int i = 0; i < data.length; i++)
        data[i] = 0x00;
}
private void sanatizeString(String[] data){
    char[] characters = data[0].toCharArray();
    for(int i = 0; i < characters.length; i++)
        characters[i] = 0x00;
    data[0] = String.valueOf(characters);
}
private void sanatizeInteger(int[] data){
    data[0] = 0;
}
```