2021

A Platform to analyze android application components for vulnerabilities.

Muchoki, Macharia Benson

School of Computing and Engineering Sciences
Strathmore University

A Platform to Analyze Android Application Components for Vulnerabilities

Macharia Benson Muchoki

Submitted in partial fulfilment of the requirements for the Degree of Master of Science in Information Systems Security

Faculty of Information and Technology

Strathmore University

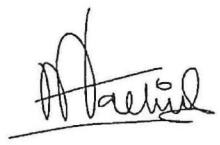
Nairobi, Kenya

December, 2021

Declaration

I declare that this work has not been previously submitted and approved for the award of a degree by this or any other University. To the best of my knowledge and belief, the dissertation contains no material previously published or written by another person except where due reference is made in the thesis itself.

Macharia Benson Muchoki



20th October 2021

Approval

The dissertation of Macharia Benson Muchoki was reviewed and approved for examination by the following:

Dr. Humphrey Njogu,
Senior Lecturer, School of Computing and Engineering Sciences
Strathmore University

Dr. Julius Butime,
Dean, School of Computing and Engineering Sciences
Strathmore University

Dr. Bernard Shibwabo
Director of Graduate Studies
Strathmore University

Abstract

Past research has shown that developers make mistakes in writing Android application configuration files, resulting to multiple vulnerabilities in these applications. More often, these vulnerabilities go unnoticed and the affected applications are installed in many end user devices. One typical security vulnerability is related to misconfiguration of Android application components. Source code obfuscation is increasingly becoming popular and was found in this study to be limiting the accuracy of most Android applications auditing tools. This study was aimed at designing, developing, and testing a platform called MobiSec. MobiSec employs a hybrid analysis technique that examines both static and dynamic features to aid Android application developers and security analysts in identifying vulnerable Android application components. The MobiSec platform was designed, developed, and tested employing the agile methodology for fast delivery. Functional, compatibility and performance tests were carried out by analysing popular Android mobile applications from Google Play Store. Performance and validation testing results showed that the MobiSec platform could be used reliably with 95% accuracy to identify vulnerable Android application components.

Keywords: Application components, Inter-component communications, Dynamic Analysis

Table of Contents

List of Figures

List of Tables

List of Abbreviations

3G and 4G	-	Third Generation and 4 Generation
API	-	Application Programming Interface
APK	-	Android Package
ARM	-	Advanced RISC Machines
CART	-	Classification and Regression Tree
CPU	-	Central Processing Unit
CVE	-	Common Vulnerabilities and Exposures
DCL	-	Dynamic Code Loading
DEX	-	Dalvik Executable
DFD	-	Data Flow Diagram
DOM	-	Document Object Model
GB	-	Gigabyte or 1000 MBs
GHz	-	Gigahertz
GID	-	Group ID
HPKP	-	HTTP Public Key Pinning
HTML	-	Hypertext Markup Language
HTTPS	-	Hypertext Transfer Protocol Secure
IDE	-	Integrated Development Environment
IP	-	Internet Protocol
JSON	-	JavaScript Object Notation
JVM	-	Java Virtual Machine
MB	-	Megabyte or 1,048,576 Bytes
MiTM	-	Man-in-The-Middle
MobiSec	-	Mobile Security
MVVM	-	Model-View-ViewModel
OS	-	Operating System
PDF	-	Portable Document Format
RAM	-	Random Access Memory
SDK	-	Software Development Kit
SSL	-	Secure Sockets Layer
TLS	-	Transport Layer Security
TV	-	Television
UI	-	User Interface
UID	-	User ID
URL	-	Uniform Resource Locator
VPS	-	Virtual Private Server
Wi-Fi	-	Wireless Fidelity
XML	-	Extensible Markup Language

Acknowledgements

First and foremost, I would like to give thanks and glory to the Almighty God for granting me the opportunity pursue and complete my Master of Science in Information System Security at Strathmore University amidst the prevailing social and economic challenges. I would also like to appreciate and acknowledge my supervisor Dr. Humphrey Njogu, whose guidance has been very instrumental in conducting this study and preparing this dissertation document.

Finally, I would like to acknowledge my family for their unwavering support and encouragement, as well as my classmates and colleagues at work, who took their time to positively critique my work and give feedback.

Chapter 1: Introduction

1.1 Background of the study

The past decade has seen a tremendous increase in the number of applications developed for smart devices as manufacturers of these devices give users the freedom to customize their devices by installing third-party applications of their choice. Third-party applications have helped improve the capabilities of smart devices, supporting more functionalities such as video streaming, social networking, and e-commerce (Chin et al., 2011).

To support the distribution of third-party applications, smart device manufacturers have provided third-party application developers with development platforms and software stores such as the Google Android Play Store, Apple App Store and Windows Store (Chin et al., 2011). Google Android Play Store remains the most popular store for smart device applications, currently hosting over 2.1 million applications as illustrated in Figure 1.1 (Statista, 2019).

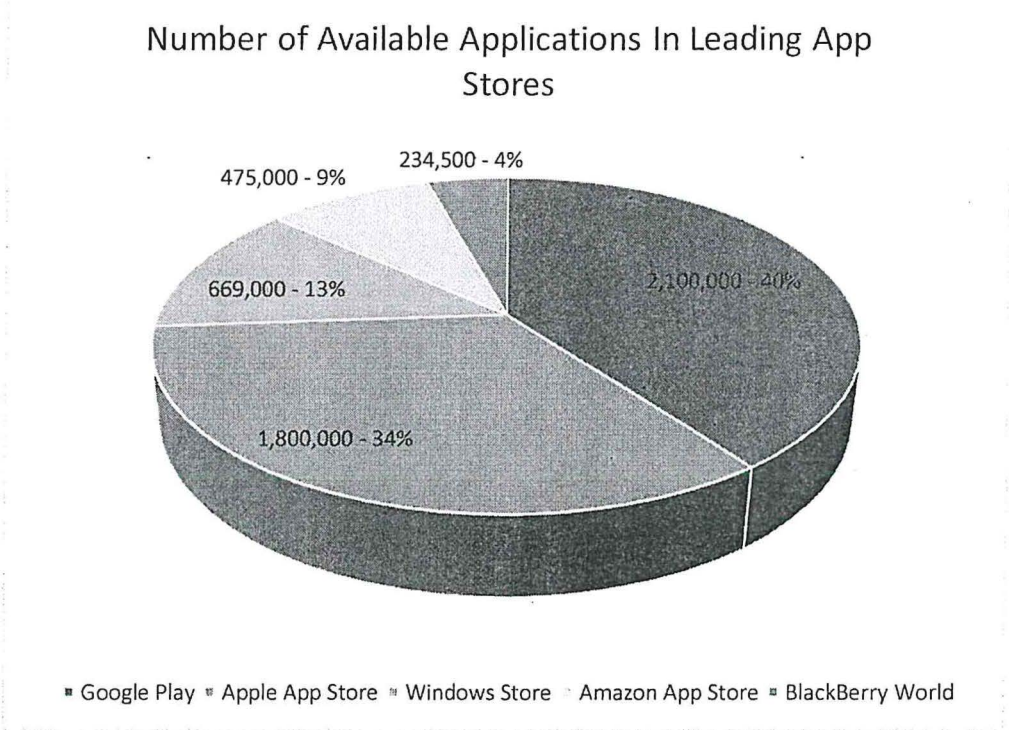


Figure 1.1: Applications available in leading app stores as of 1st quarter 2019.

Source: Statista, (2019).

Mohamed and Patel, (2015) notes that third-party applications are scrutinized for their intended functionality and security before being allowed for distribution in market stores. However, there are security concerns in Android as it has been found that the provenance process for applications in Google Play Store is not as thorough compared to that of other manufactures such as Apple. This has resulted to malicious applications finding their way into end-user devices (Mohamed and Patel, 2015).

Furthermore, the Android platform is more flexible and allows for third-party applications from other sources, other than Google Play Store to be installed on end-user devices. Bypassing Google's evaluation process allows for erroneous and malicious applications to be distributed and installed in end-user devices (Mohamed and Patel, 2015).

This flexibility of the Android platform and large user base has made it a target for attacks through malicious applications aimed at compromising the confidentiality, integrity and availability of information held in the devices. Android is currently positioned at number three among the most vulnerable products based on the total number of disclosed vulnerabilities to date, as illustrated in Figure 1.2 (CVE Details. Top 50 Products By Total Number Of "Distinct" Vulnerabilities, n.d.).

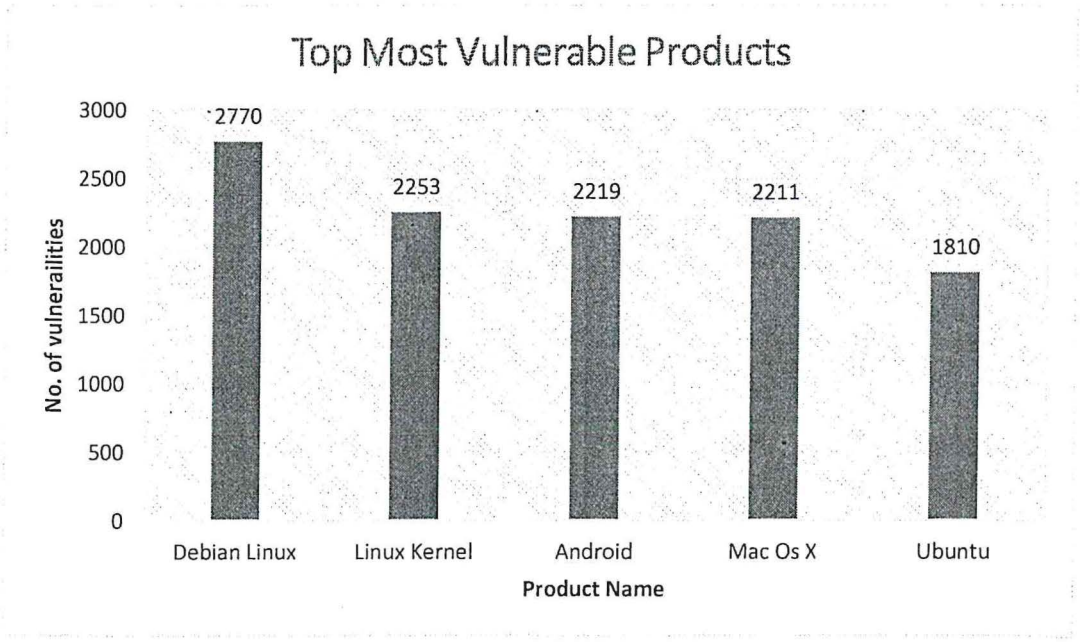


Figure 1.2: Five Topmost Vulnerable Products

Source: CVE Details. Top 50 Products by Total Number of Distinct Vulnerabilities. Retrieved on 20 July 2019.

Jha et al. (2017) notes that developers make mistakes in writing Android applications, leaving them exposed to multiple application-based security threats. One prominent mistake is made in the writing of the Android application's Manifest file that puts together the application's configuration parameters that control the functionalities of the application and the execution behaviour of its components (Jha et al., 2017).

Hur and Shamsi, (2017) explain that Android applications are composed of four categories of components that include: - activities, services, broadcast receivers and content providers. To support inter-component communications, these components are usually in some instances exported or provided for use by other applications. This however poses a security risk when such components are not properly exported making them vulnerable to intent spoofing and unauthorised intent receipt related attacks (Bhiwani and Parekh, 2017).

Intent spoofing vulnerabilities result when an attacker sends an Intent to a component that is not expecting to receive it from the sending component, triggering a malicious action from this component. On the other hand, unauthorised intent receipt vulnerabilities result when an application's Intent is intercepted by another malicious application exposing all its data (Bhiwani and Parekh, 2017).

Bhiwani and Parekh, (2017) recommend that developers of Android applications should exercise caution by protecting sensitive exported components with permissions.

Audit tools such as ComDroid and ManifestInspector should be used to scan Android applications before they are deployed into the Google Play Store to identify exposed components and other vulnerabilities. These tools are mostly developed by employing static and dynamic analysis techniques. Dong et al., (2018) note that static analysis techniques are increasingly becoming ineffective due to code obfuscation that developers employ to their intellectual property. Moreover, dynamic analysis techniques that monitor executables at runtime have also been found not to be comprehensive as they leave out examination of application static features.

This study focuses on designing and developing a platform that can aid Android application developers, security analysts and users in auditing developed and installed applications for vulnerabilities related to exported components making use of the hybrid analysis technique that examines both static and dynamic features. This platform was developed employing the agile methodology and tested by analysing popular Android applications downloaded from the Google Play Store.

1.2 Problem Statement

In the past, most security tools to review Android application components have been developed leveraging on static analysis procedures, that work on disassembled APK files. This approach has recently been proved to be no longer effective as developers are increasingly crafting their applications' code with obfuscation and encryption schemes to protect them against reverse engineering. Other tools employ dynamic analysis techniques that monitors the execution of an application while inspecting its runtime behaviour. This approach has also been found not to be comprehensive as it leaves out examination of application static features. To overcome these shortcomings, a hybrid analysis technique that examines both static and dynamic application component features needs to be explored and leveraged to develop an effective tool for reviewing Android application components.

1.3 Objectives

General Objective

This study was aimed at designing and developing a platform to aid Android developers and security researchers in identifying vulnerable Android application components using a hybrid analysis technique.

Specific Objectives

i. To examine vulnerabilities associated with Android application components.
ii. To review static, dynamic and hybrid analysis techniques, approaches and tools applied in identifying vulnerable Android application components.
iii. To design, develop and test a platform for identifying vulnerable Android application components employing the hybrid analysis technique.
iv. To verify the effectiveness of the developed hybrid analysis based platform in identifying vulnerable Android application components.

1.4 Research Questions

i. What are the vulnerabilities associated with Android application components?
ii. How are the static, dynamic and hybrid analysis techniques, approaches and tools applied to identify vulnerable Android application components?
iii. How can the hybrid analysis technique be applied to design, develop, and test a platform

to identify vulnerable Android application components?

iv. How is the developed hybrid analysis based platform effective in identifying vulnerable Android application components?

1.5 Justification for the study

Jha et al., (2017) note that Android application developers often make mistakes in writing application configuration files, leaving them with multiple vulnerabilities especially those related to exportation of Android application components. When application components are improperly exported, this can lead to any application gaining access to the application's sensitive information and even be able to corrupt its internal state.

These vulnerabilities more often go unnoticed as Android platform has a flexible design that allows installation of third-party applications from other sources without going through the normal security vetting process in Google Play (Mohamed and Patel, 2015). Moreover, with increased popularity of applications' source code obfuscation, the chances of discovering these vulnerabilities in production environments are minimal using the traditional static or dynamic analysis tools.

A hybrid analysis technique that allows for examination of an application's static and dynamic features as borrowed from malware analysis research is more efficient in identifying applications' vulnerable exported components. MobiSec tool was developed in this study employing the hybrid analysis technique and was tested in identifying vulnerable application components from applications whose source code had been obfuscated.

1.6 Scope and Limitation

This study focused on understanding the Android platform architecture and the various components that make up an Android application. Specifically, this study resulted to development of a platform that exposed vulnerabilities related to improperly exported Android application components.

The platform developed in this study however did not focus on auditing for the other security vulnerabilities in Android applications such as physical, system-based, and network-based vulnerabilities. Moreover, the platform developed in this study only generated a report by analysing Android applications' configuration parameters and not by carrying out full

application source code analysis. Finally, the developed platform only gives recommendations on remediating the identified vulnerabilities but did not help its users remediate them.

Chapter 2: Literature Review

2.1 Introduction

This chapter focuses on the reviewing existing literature and theories regarding the Android architecture, Android application components and security vulnerabilities revolving around inter-component communications. This chapter also presents a conceptual framework that illustrates the process that will be undertaken in processing of Android application configuration parameters as inputs to give out a report on inter-component communications vulnerabilities noted.

2.2 Android Platform Architecture

Khan and Shahzad, (2015) note that Android is an open source OS for mobile devices that has become pervasive and ubiquitous, running in smartphones, wrist watches, infotainment systems, tablets and smart TVs. Android can better be described as a middleware running on top of an embedded Linux system as illustrated in Figure 2.1. The middleware itself is written in Java and C/C++ with the underlying Linux internals customised to provide strong isolation and to contain exploitation (Khan and Shahzad, 2015).

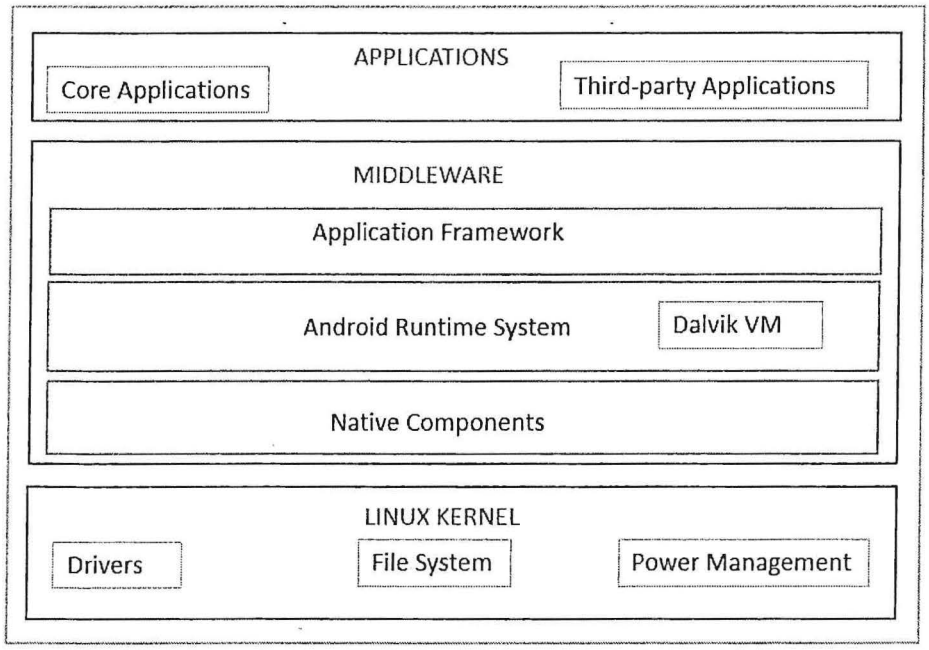


Figure 2.1: Android Platform Architecture

Source: Khan and Shahzad, 2015

Android Applications on the Android platform are written in Java, are compiled to a custom byte-code called Dalvik EXecutable (DEX) byte-code format; and each run as a process with a unique UNIX user identity (UserID) as Khan and Shahzad, (2015) explain.

Each application executes within its own Dalvik VM interpreter instance with all the inter-application communications passing through the Android's middleware. Inter-process communications generally take the form of Intent messages either addressed directly to a component using the application's unique namespace or to an action string (Khan and Shahzad, 2015).

2.3 Android Application Components

The Android middleware supports four types of inter-process communications that correspond to the four categories of components that make up Android applications. These components include: - Activities, Services, Content Providers and Broadcast Receivers.

2.3.1 Activities

Activities in an Android application represent the screens that are presented to the physical user via a touchscreen and keypad. An application contains a main Activity and other child Activities. These Activities are organised in a stack called a back stack, and only one Activity in the application has input and processing focus at a time. The user interaction with the application involves a progression sequence of Activities each calling the other with some data being passed between them. When a new window is started, the previous activity is pushed to the back stack and it is stopped until the new activity is done or the back button is pressed (Six, 2012).

2.3.2 Services

Services in an Android application provide background processing capability that continues even after its attached application loses focus. Services do not provide user interfaces or interact with the user. A service usually takes any of these two forms: -

Started – In this case, a Service starts and runs to perform a single operation after which it terminates itself. No results are returned to the user. An example is a Service started to upload a file (Six, 2012).

Bound – This is where a component is bound to a service to complete a task. This provides a client-server like interface to support inter-process communications between application components (Six, 2012).

2.3.3 Content Providers

Content providers are database-like mechanisms addressable by their application-defined URIs for sharing data with other applications. They are used as a persistent internal data storage and support standard SQL-like queries through which components in other applications can retrieve and store data according to the Content Provider's schema (Chin et al., 2011).

2.3.4 Broadcast Receivers

Broadcast Receivers receive Intents messages sent to multiple applications (broadcasts) and usually relay messages to Activities or Services. Three types of broadcast intents are defined: normal, ordered and sticky broadcasts. Normal broadcasts are sent to all registered receivers at once, then disappear; whereas, ordered broadcasts are delivered serially to one receiver at a time. On the other hand, sticky broadcasts remain accessible to future receivers after they have been delivered (Chin et al., 2011).

2.4 Exported Application Components

To support inter-process communications between Android application components as illustrated in Figure 2.2, Chin et al., (2011) observes that a component needs to be declared in the Android application Manifest.xml file for it to receive Intents. An exported component or public component is an application component that has at least one Intent filter or has the EXPORTED flag set to true in its declaration in the Manifest file (Chin et al., 2011).

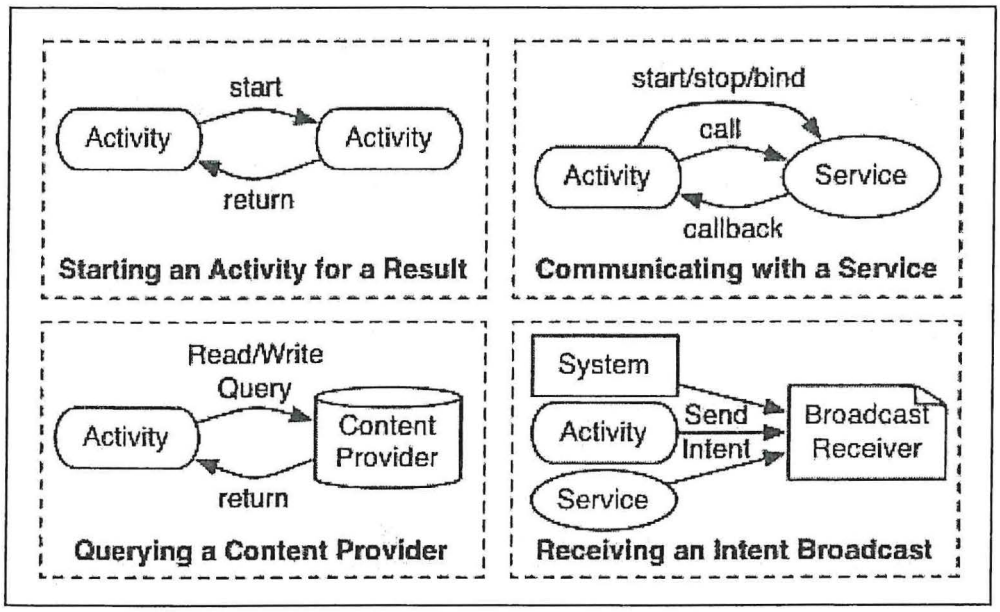


Figure 2.2: Inter-process Communication between Application Components

Source: Chin et al., (2011)

Chin et al., (2011) define Intents as messages or self-contained objects that include data and specify a remote procedure to invoke with associated arguments. Intents are used by Android applications for both inter-application communication and intra-application communication. Inter-application communication refers to communications between components of different applications, while intra-application communication refers to communications between components within an application (Chin et al., 2011).

Intents support both explicit and implicit communication. An explicit Intent explicitly specifies the Android application it should be delivered to whereas an implicit Intent is delivered to any Android application that supports a certain operation. An explicit Intent identifies the intended recipient by name, therefore guaranteeing that the Intent is received as intended. On the other hand, an implicit Intent is received by any application(s) the Android platform chooses, allowing for late runtime binding between applications (Chin et al., 2011).

Exported components can receive Intents from other applications, but these are limited to those Intents of the type specified by their Intent filters. An Intent can include component name, action, data, category and extra data. Android matches each of the Intent's data to the component's declaration to determine which component to receive this Intent (Chin et al., 2011).

2.5 Exported Component Vulnerabilities

Chin et al., (2011) note that if an Intent is sent to the wrong application, it can result to data leakage or transferring of permissions to another application. On the other hand, if a component is accidentally made public and is receiving Intents from other applications, then external applications can invoke its components and inject malicious data into it.

Bhiwani and Parekh, (2017) categorise vulnerabilities related to Android application exported components into Unauthorised Intent Receipt and Intent Spoofing vulnerabilities as illustrated in Figure 2.3.

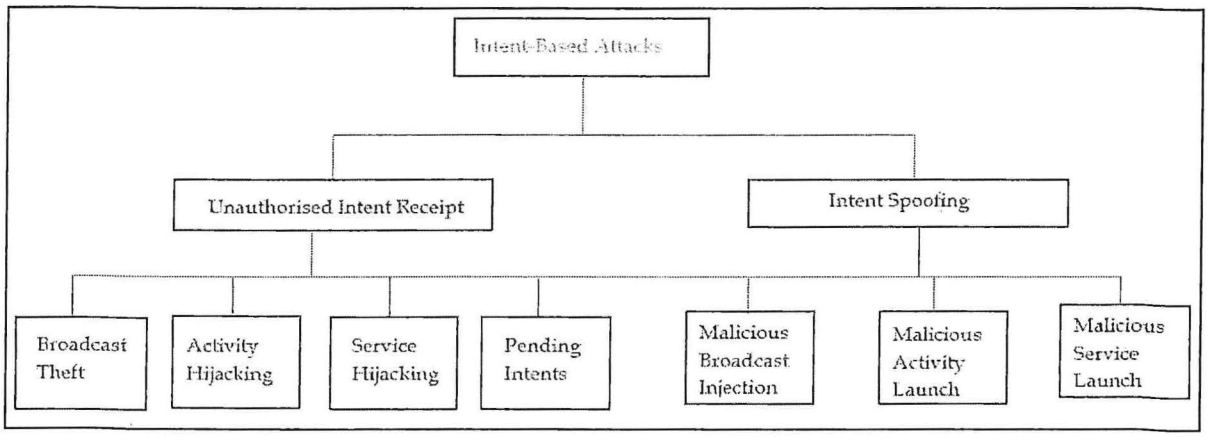


Figure 2.3: Attack Surfaces for Application Exported Components

Source: Bhiwani and Parekh, (2011)

2.5.1 Unauthorised Intent Receipt Vulnerabilities

These vulnerabilities result from the possibility of an implicit Intent being intercepted by a malicious application to which it was not intended. This is possible when a malicious application has declared an Intent filter with all the actions, data, and categories listed in the Intent (Bhiwani and Parekh, 2011). Vulnerabilities in this category include: -

Broadcast Theft

Chin et al., (2011) observe that broadcasts are vulnerable to passive eavesdropping and active denial of service attacks. Passive eavesdropping occurs when an Application sends a public broadcast/unprotected implicit Intent, which is read by a malicious Broadcast Receiver declaring an Intent filter with all the possible actions, data, and categories. On the other hand,

active denial of service attacks can result from a malicious Broadcast Receiver that stops further propagation of an ordered broadcast after its receipt (Chin et al., 2011).

Activity Hijacking

In this case, a malicious Activity is launched in place of the intended Activity reading Intent data meant for the other activity. In sophisticated cases, expected user interface Activity can be spoofed to steal user inputs (Bhiwani and Parekh, 2011).

Service Hijacking

Service hijacking happens when a malicious Service intercepts an implicit Intent meant for another Service. The hijacking Service can then steal data from that Intent of send spoofed responses to the initiating or calling application (Chin et al., 2011).

Pending Intents

Bhiwani and Parekh, (2011) explain that a pending Intent is a type of Intent with a target action that can be performed later by the holder of that Intent. This Intent carries the same privileges and permissions of its creator and if intercepted, then it can lead to permission-escalation attacks (Chin et al., 2011).

2.5.2 Intent Spoofing Vulnerabilities

The Intent spoofing vulnerabilities occur when a malicious application sends an Intent to an exported component that is not expecting Intents from that application. A malicious action can result if the targeted application reacts upon the receipt of that Intent (Chin et al., 2011). Bhiwani and Parekh, (2011) note the following kinds of Intent spoofing vulnerabilities.

Malicious Broadcast Injection

Chin et al., (2011) observe that a malicious broadcast injection can result if an exported Broadcast Receiver blindly trusts incoming broadcast Intents, taking inappropriate action or operating on malicious data from the broadcast Intent. Such malicious Intent data might as well be propagated to other components such as Activities and Services which receivers pass data to.

Malicious Activity Launch

With either implicit or explicit Intents, exported Activities can be launched by other applications. Chin et al., (2011) note three instances in which exported Activities can be exploited. First, an application's data store can be corrupted if it uses Intent data from an Activity without verifying the source. Secondly, a user can be tricked into launching a malicious Activity and making changes in it in place of the legitimate Activity. Thirdly, an exported Activity can leak sensitive data to the calling malicious Activity after completing data processing (Chin et al., 2011).

Malicious Service Launch

A malicious application can start or bind to an exported Service if it's not protected by strong permissions. Depending on the services it provides, such a Service will leak sensitive information to the malicious application or have its data corrupted (Chin et al., 2011).

2.6 Auditing for Exported Component Vulnerabilities

Auditing of exported components refers to security review of Android application components to identify security flaws or configuration gaps that can lead to vulnerabilities in the affected applications. This also includes giving recommendations for remediation of the gaps identified in form of a well organised report.

Chin et al., (2011) acknowledge that previous research has been done to identify mistakes Android application developers make that compromise applications' security. Ahmed and Sallow, (2017) identify two techniques that are used to audit for security gaps in Android applications. These techniques include detection techniques and analysis-based techniques.

2.7 Exported Components Auditing Techniques

Detection-based audit techniques such as the one defined by Malik, Campos & Jaafar, (2019) employ machine learning to detect anomalies in system calls by comparing their type, frequency and sequence. This enables detection and differentiation between benign and malicious Android applications. On the other hand, analysis-based techniques focus on reviewing Android application's static and dynamic features that include disassembled APK files artefacts and runtime behaviour respectively.

This study focuses on examining the available analysis-based techniques to audit for component-based vulnerabilities in Android application components.

2.8 Analysis-based Component Audit Techniques

Analysis-based techniques for auditing security vulnerabilities in Android application components include static analysis, dynamic analysis and hybrid analysis techniques. These techniques are further expounded below with existing tools for each technique.

2.8.1 Static Analysis Technique

Static Analysis involves a review of Android application static source code files to arrive at a finding. These source code files may be obtained from the developer of the application or extracted from disassembled APK files. Some of the common static analysis based tools include ManifestInspector and ComDroid.

2.8.1.1 ManifestInspector Tool

ManifestInspector is an open source static analysis tool written in Java to assist application developers in detecting errors in the Android manifest files employing a rule-based approach. ManifestInspector defines a set of 116 rules and can identify misplaced components in the configuration file, erroneously exported components and unprotected components (Jha et al., 2017).

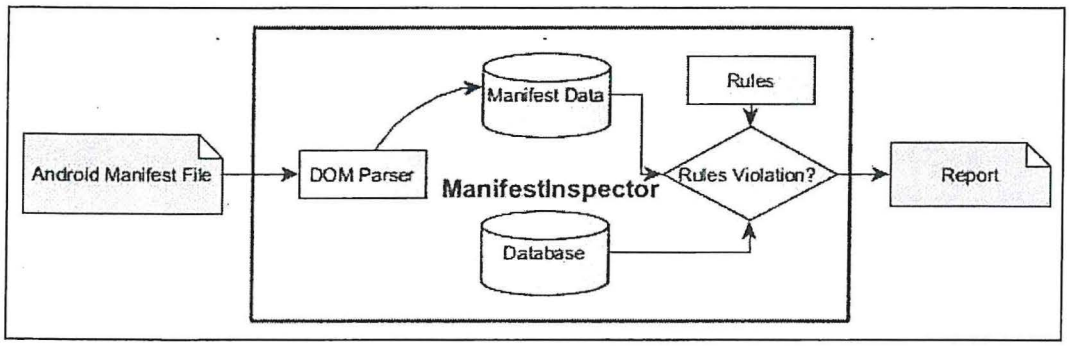


Figure 2.4: ManifestInspector Overview

Source: Jha et al., (2017)

As illustrated in Figure 2.4, the ManifestInspector tool is loaded with Android Manifest files which are then parsed through a DOM parser to extract the relevant information for analysis. The tool makes use of a text-based database to store valid elements and attributes of a manifest file. Using a set of defined rules, the extracted manifest data is validated for compliance with the baseline configurations. The tool then categorises the configuration mistakes observed into

the five categories that included issues related to: user interface, performance, execution, security, and compatibility.

The ManifestInspector tool can identify improperly exported components that have the android: exported attribute set to false but contain intent-filters as well as those that have the android: exported attribute set to true but don't have intent-filters. The tool can also identify those components that are exposed to privilege escalation attacks as they are exported but not protected by relevant permissions.

The ManifestInspector tool was used to review 13,483 free Android applications downloaded from Google Play store. The tool relied on a dataset of permission lists obtained from querying Android 6.0 package manager system permissions as well as from manifest file static attributes. Android system permissions and manifest attributes are however bound to change, and this makes it imperative to have a future-oriented tool that in addition to examining application static features, it also examines the dynamic features to generate a comprehensive security analysis report.

2.8.1.2 ComDroid Tool

ComDroid is a static analysis tool that detects potential vulnerabilities in Android application components by examining DEX files. The DEX files are disassembled using the Dedexer tool and parsed by the ComDroid tool to generate alerts for potential component and intent vulnerabilities. The tool evaluates both the system-defined and application-defined permissions as used to protect components. Components protected with weak permissions which are easy to acquire are exposed (Chin et al., 2011).

ComDroid conducts intents and components analysis to identify unauthorised intent receipt as well as intent spoofing vulnerabilities. The tool tracks intent states, intent-filters, registers, and sinks to identify potential exposure. ComDroid tool was tested by analysing 100 applications from Google Play and the findings were verified by manually testing 20 applications (Chin et al., 2011).

Type of Exposure	Percentage
Broadcast Theft	44 %
Activity Hijacking	97 %
Service Hijacking	19 %
Broadcast Injection	56 %
System Broadcast w/o Action Check	13 %
Activity Launch	57 %
Service Launch	14 %

Figure 2.5: ComDroid Analysis results per Exposure Type

Source: Chin et al., (2011)

Figure 2.5 illustrates results from the ComDroid tool when it was used to examine different exposures from a selected set of Android applications. Activity-related attacks were found to be the most prevalent ones, followed by broadcast-related attacks.

ComDroid focuses on analysing Android application static features. Chin et al., (2011) recommend further research to be done on developing a combined static and dynamic analysis tool that will assist in validating vulnerabilities identified during static analysis through dynamic analysis by trying to exploit them at run-time.

2.8.2 Dynamic Analysis Technique

In this methodology, the Android application is audited while running on an Android device. This is either done when the application is being installed or after installation. The started and running processes for an application are tracked as well as its functionality. Kirin and TainDroid are two of the most popular tools used in examining Android applications' dynamic features to identify component-based vulnerabilities.

2.8.2.1 DATDroid Tool

The DATDroid tool employs dynamic analysis technique for malware detection. The tool consists of three phases that include:- feature extraction, feature selection and classification as illustrated in Figure 2.6. DATDroid makes use of a script to log and extract applications' behaviour that include system calls, CPU usage, memory usage, and network packets. (Thangaveloo et al., 2020).

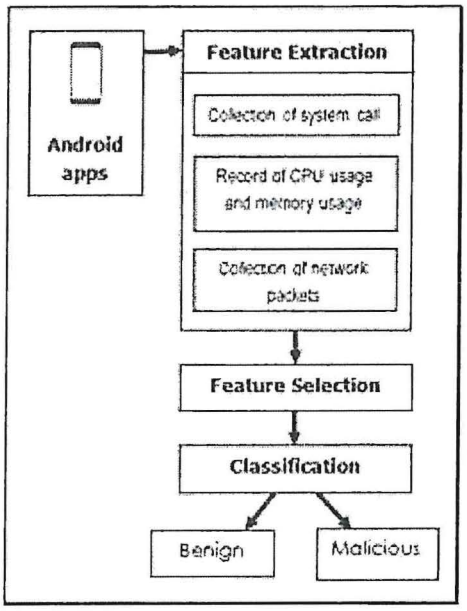


Figure 2.6: DATDroid Framework Architecture

Source: Thangaveloo et al., (2020)

After features extraction, the most relevant features are selected to optimise on performance using the Gain Ratio Attribute Evaluator feature selection methodology. The tool then does classification to discern between malicious and benign application using the Random Forest classifier algorithm (Thangaveloo et al., 2020).

The DATDroid as used to review 200 dataset samples of both malicious and benign applications in both the training and testing phases. Although, Thangaveloo et al., (2020) concluded that the DATDroid could be used reliably to discern between malicious and benign applications with an accuracy rate of 91.7%, they recommend that a hybrid analysis technique be explored that would employ dynamic analysis to reveal runtime information while static analysis would reveal static information not revealed during dynamic analysis.

2.8.2.2 TainDroid Tool

TainDroid is a system-wide information flow tracking tool that dynamically analyses the behaviour of a third-party application in an Android device to identify suspicious behaviour and possibilities of data leakage. The application can specifically locate applications with improperly exported components and in some instances, identify malicious applications trying to exploit this vulnerability (Enck et al., 2010).

TainDroid applies a security label to user data and logs when that data is transmitted through the network or when it leaves the system. This logging highlights the data being transmitted, the application transmitting such data and the destination as illustrated in Figure 2.7.

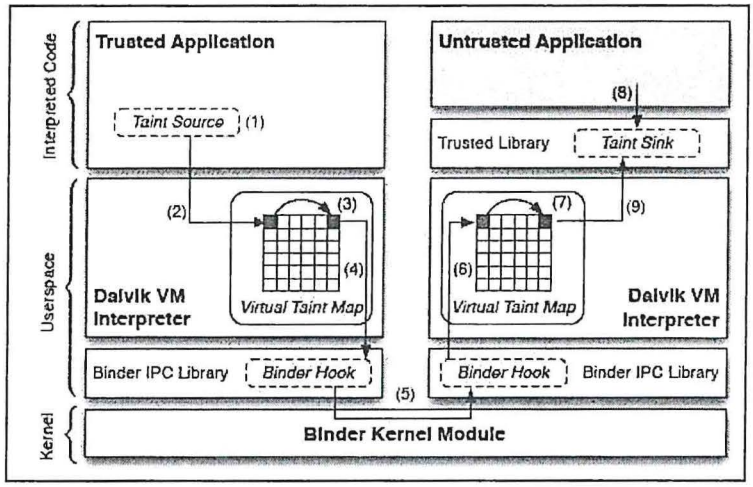


Figure 2.7: TainDroid Architecture within Android

Source: Enck et al., (2010)

From the total number of third-party applications that were reviewed through the TainDroid tool, two-thirds of them were found to be leaking sensitive data through exposed components and excessive permissions (Enck et al., 2010).

Enck et al., (2010) note that TainDroid only tracks data flows and not control flows. Lack of control flows review creates an avenue for exfiltration of data from applications. Therefore, Enck et al., (2010) recommend that static analysis capabilities should be incorporated into future tools to allow for a comprehensive review process that tracks both control and data flows.

2.8.3 Hybrid Analysis Technique

Hybrid analysis focuses on examining both static and dynamic features of an Android application building on both static and dynamic analysis. This reviews Android application source code files, running processes and functionalities to maximise effectiveness in identifying component security gaps.

Dong et al, (2018) observe that to protect intellectual property, application developers frequently obfuscate their source code to prevent competitors from copying the code and building their own products. Dong et al, (2018) note that there exist multiple obfuscation techniques used by Android applications including identifier renaming, string encryption,

excessive overloading and many more. Consequently, many off-the-shelf obfuscators have been developed with the most popular ones including ProGuard, DexGuard, DexProtector and DashO. This presents a challenge to security analysts as having an efficient and effective de-obfuscator becomes almost impossible, hence the increasing preference of hybrid analysis techniques in security review of Android applications.

Previous research has focused on defining procedures, frameworks and tools for malware detection in Android platforms using hybrid analysis such as the mad4a framework. The mad4a framework developed by Kabakus & Dogru, (2018) combines both static and dynamic analysing techniques to detect malware in Android. The mad4a framework employs a two-step process that starts with extracting static information from the APK files. From the static information obtained, the number of API calls mapped to dangerous permissions is calculated. On the second step, the monkeyrunner tool is used to generate pseudo-random streams of user and system events on the emulator, and then logs are corrected and analysed to identify any suspicious activities (Kabakus & Dogru, 2018).

The procedures employed in the mad4a framework can be borrowed in developing a hybrid analysis tool to examine vulnerabilities in Android application components. This tool can examine the static features from an Android application that would include requested permissions and API calls. This tool can also examine dynamic features such as data and control flow within the application.

2.9 Summary of Identified Gaps in Existing Tools

Table 2.1 gives a summary of the gaps that were identified in existing tools used to audit for vulnerabilities in Android application components.

Technique	Tool	Identified Gaps
Static Analysis	ManifestInspector	- Depends on static permissions and manifest attributes leaving out dynamic features such as API calls. - Requires disassembling of application APK files before analysis which is tedious. - Not effective in the review of obfuscated applications.
	ComDroid	- Depends on static permissions and manifest attributes leaving out dynamic features such as API calls. - Requires disassembling of application APK files before analysis which is tedious. - Not effective in the review of obfuscated applications.
Dynamic Analysis	DATDroid	- Not comprehensive and it leaves out the review of application static features. - Only reviews a set of selected features, leaving out some critical features.
	TainDroid	- Not comprehensive and it leaves out the review of

		application static features. - Complicated and tedious to set up

Table 2.1: Summary of Identified Gaps in Existing Tools

2.10 Conceptual Framework

Figure 2.8 illustrates a conceptual framework for the developed platform to audit vulnerable Android application components.

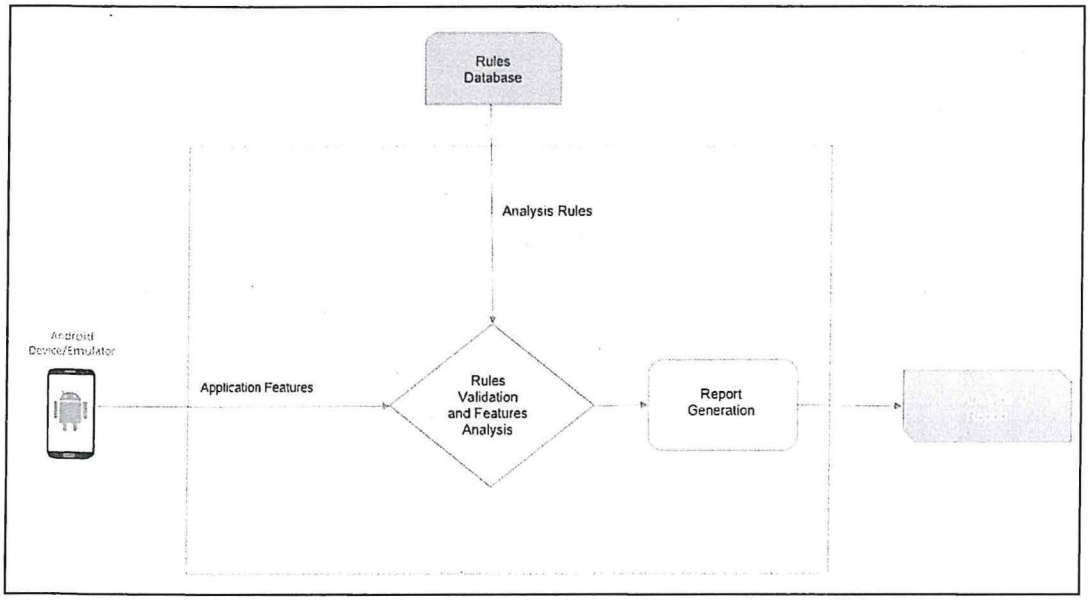


Figure 2.8: Application Components Audit Tool Conceptual Framework

The security review process for Android application components starts by extraction of both application features from a selected installed application. These features are then loaded into the rules validator where they are examined against a predefined set of rules loaded from a rules database to produce results. These results are then formatted by the report generator to give out an analysis report as desired by the user.

2.11 Summary

This section has presented an overview of the related literature to this study, starting by exploring the Android platform, Android applications and various components that make up an Android application. This section has also presented a summarised review of vulnerabilities related to Android applications components. Additionally, existing static analysis and dynamic

analysis tools were also reviewed, examining their architecture, and evaluating their effectiveness in the audit of vulnerable application components.

A hybrid analysis technique has also been introduced in this chapter as applied in other studies for malware analysis. The mad4a hybrid analysis framework has been explored on its applicability to identification of Android application component vulnerabilities.

To close the chapter, a conceptual framework has been presented to illustrate how the developed solution handle and processes inputs to give out the expected results.

Chapter 3: Research Methodology

3.1 Introduction

This chapter describes the methodology used in the study to answer research questions and meet research objectives. Systematic literature review methodology was used to answer research questions 1 and 2 while the Agile methodology was used to answer research questions 3 and 4.

3.2 Systematic Literature Review Methodology

The first and second objectives for examining vulnerabilities associated with Android application components and the techniques, approaches and tools applied to audit for them, were met through literature review conducted following the systematic literature review methodology.

Snyder, (2019) explains that systematic literature review methodology is the process for identifying and critically appraising relevant research, as well as for collecting and analysing data from said research with an aim to identify all empirical evidence that fits the pre-specified inclusion criteria to answer a particular research question or hypothesis.

Systematic literature review synthesizes research findings in a systematic, transparent, and reproducible way. Xiau and Watson, (2019) notes that systematic literature review encompasses the following eight common steps that include:-

i. Formulating the research problem
ii. Developing and validating review protocol
iii. Searching the literature
iv. Screening for inclusion
v. Assessing quality
vi. Extracting data
vii. Analyzing and synthesizing data
viii. Reporting findings

3.2.1 Formulating the Research Problem

Formulation of research problem entitled the definition of research problem and formulation of research questions as discussed in Chapter 1 parts 1.2 and 1.4 respectively. The literature review was geared towards answering research questions 1 and 2.

3.2.2 Developing and Validating Review Protocol

Xiau and Watson, (2019) notes that review protocol is comparable to research design where a plan is developed to specify methods to be utilized in conducting literature review. This study involved describing all review elements including inclusion criteria, quality assessment criteria and screening procedures for relevant literature.

3.2.3 Searching the Literature

This involves identifying the sources for literature to be reviewed. This study took an approach where three sources were identified: electronic databases, backward searching and forward-searching. A keyword search was done on Google and Google Scholar to search for open access publications from electronic databases.

3.2.4 Screening for Inclusion

From the list of references obtained from the previous step, a two-stage procedure was used to sieve through the articles for inclusion based on their relevance. In the first step, articles were sorted for inclusion based on their abstracts while in the second step, this was done based on their full-text review.

3.2.5 Assessing Quality

The quality assessment aimed at understanding the literature before data could be extracted from it. This involved an in-depth critical review of the study based on a checklist to evaluate its qualitative and quantitative properties.

3.2.6 Extracting Data

This step involved synthesizing research data and its extraction. A summary of findings was extracted through a review of the entire paper to obtain relevant information for this study.

3.2.7 Analysing and Synthesizing Data

In this step, data was organized according to the chosen review type. In this study, the research data obtained was organized into a combination of charts, tables and textual descriptions.

3.2.8 Reporting Findings

This involved reporting the steps followed in conducting literature review and the final inclusion of the selected literature into this study. The relevant literature review was included in this study in Chapter 2 and properly cited.

3.3 Agile Methodology for Software Development

Objectives three and four entailed designing, developing, and testing a platform for auditing vulnerabilities associated with Android application components as well as validation of this platform to verify its effectiveness were met through the guidance of the Agile software development methodology.

Agile Software Development Methodologies refers to a set of software development methodologies based on the same principles for providing flexibility and opportunity to assess the direction of a project throughout its development lifecycle. These methodologies are iterative, incremental, adaptive, collaborative and support modularity (Sharma et al., 2012).

Agile Development methodology was selected for this study for the following reasons:-

i. Agile development process supports for rapid delivery of high-quality software products.
ii. Agile development process promotes user-focused testing ensuring frequent inspection of the software product and adaptation.
iii. Agile development process allows for flexibility as new changes can be factored in for each new iteration.

The Agile process is composed of six steps as illustrated in Figure 3.1.

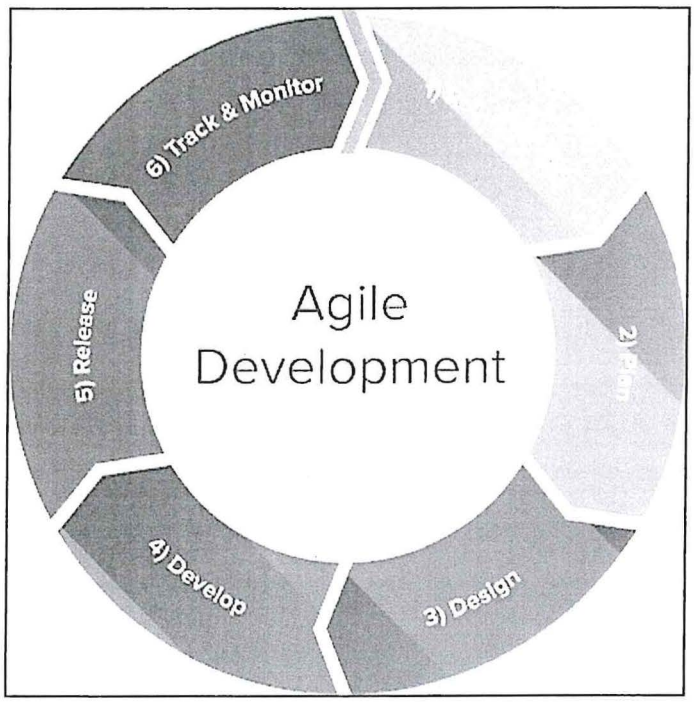


Figure 3.1: Phases of the Agile Development Process.

Source: Smartsheet, (n.d.)

3.3.1 Requirements Gathering

This step entails discovering project requirements by defining a research problem that needs to be resolved. This can be done through techniques such as Brainstorming, Document Analysis, Focus Groups, Interface analysis, Interviews, Observation, Prototyping, Requirement Workshops, Reverse Engineering and Surveys.

In this study, requirements gathering was done employing the document analysis technique. Documentation of existing Android applications audit tools was reviewed for identifying gaps and areas of improvement. Nuggets of requirements information were also obtained from analysing functional and non-functional requirements from previous studies that review vulnerabilities arising from the exportation of Android applications components.

3.3.2 Planning

Planning entails conducting an in-depth analysis of the gathered requirements to obtain a detailed understanding of the problem. The aim is to transform the requirements gathered into unambiguous, complete, consistent and traceable project requirements for resources allocation.

In this study, this involved carrying out a detailed analysis of the requirements gathered to estimate the scope, resources required and formulate the problem statement. This helped in the prioritization of requirements while defining project scope and limitations. Development of planned test activities was also done in this phase.

3.3.3 System Design

In this phase, the system requirements were translated into a physical design. System architecture, modules, interfaces and data were defined in this phase according to the refined requirements from previous stages. Modelling of the proposed system using design diagrams that include use case and sequence diagrams was also done in this phase. The use case diagram was used to demonstrate the relationship between the user who is called an actor and the system. On the other hand, the sequence diagram was used to illustrate object interactions in relation to the time sequence. The following tools were used in this phase:-

- Microsoft Visio 2016 – This is owned by Microsoft Incorporation and was used to design the use case diagram.
- SmartDraw – This is owned by SmartDraw Company and was used to draw the sequence diagram.
- Android Studio IDE – This is an open-source tool owned by Google Incorporation and was used to design user interfaces for the application.

3.3.4 System Development and Testing

3.3.4.1 System Development

Building on the design diagrams, this phase involves the actual development of the software product. In this study, this entailed writing the application's source code using the Java programming language in an object-oriented approach. The following tools were used in this phase:-

- Android Studio IDE – This is an open-source IDE owned by Google Incorporation and was used to write, debug and compile the Java mobile object-oriented source code.
- Linux Operating System – This is an open-source Unix-based operating system on which the development and testing tools were installed.
- Android Device – This is an Android Device running Android version 9 and was used

to test the developed application in a real end-user device.

- Android Emulator – This is an open-source tool developed and owned by Google Incorporation. This will be used to simulate a variety of Android devices and API levels where the physical devices are not present.

3.3.4.2 System Testing

This entails testing the developed program for its reliability and validity. In this study, involved testing of the developed solution for its reliability and validity in answering the study questions. Functional, non-functional, structural, usability and unit tests were also carried out in this phase.

i. Functional Testing

Functional testing is only concerned about the functionality of the developed application and not about its implementation (Sommervile, 2015). The following functional tests were conducted on the developed platform:-

- Installation and deployment on different Android versions.
- Analysis of APK files from different sources and of different sizes.
- Generation of a consistent analysis report for an APK in multiple reviews.

ii. Non-functional Testing

Non-functional tests are usually conducted on the developed platform for those features not related to the functioning of the application. Non-functional testing on the developed platform was done on the following areas:-

- Intuitive user interface
- Resource optimization i.e. storage, power and processor utilization
- Performance

iii. Structural Testing

According to Sommervile (2015), structural testing involves a thorough review of the application's structure and internal implementation from a white-box approach. For this study, this involved source-code review to uncover logical bugs using manual review and the FindBugs open-source tool for static code analysis.

iv. Usability Testing

Usability testing entails conducting tests for the developed product on a group of representative users and observing them complete different tasks on the product. In this study, a total of twenty software developers were selected and issued with the application's user manual to conduct several tests on the application. These users were later issued with a questionnaire to provide feedback as illustrated in Appendix B.

v. Unit Testing

In unit testing, individual software components such as methods and object classes are tested simulating all events that can cause a state change (Sommervile, 2015). In this study, unit tests were integrated into the development phase where local and instrumented tests were carried out.

- Local Tests – These tests are set to run on the local machine as compiled on the JVM to minimize execution time (Android Developers, n.d.).
- Instrumented Tests – These tests are set to run on an Android device or emulator and are necessary for complex Android dependencies that require a more robust environment (Android Developers, n.d.).

3.3.5 Product Release

This entails delivering the product to prospective customers for their testing and feedback. In this study, the developed Android application was uploaded on Google Playstore for open access to users for testing and feedback. A link to the accompanying user manual was also provided for further user guidance. Refer to Appendix A for a copy of the user manual.

3.3.6 Track and Monitor

This step entails collecting feedback from customers after testing the software product. In this study, feedback was collected from developers after testing the solution delivered to them. Feedback was collected through the contacts provided on Google Play as well as from the comments section which assisted in meeting objective 4.

3.4 Research Quality

For quality assurance, this research was conducted in line with the University regulations, guidelines and other set standards. The problem statement was formulated, and research

objectives were set to ensure that research outputs met the set requirements and make contributions to the body of knowledge. Additionally, the literature review sources were restricted to only reputable sources, which were properly cited using a standardised format.

3.5 Ethical Considerations

Ethical considerations were given a priority in this study as per the university guidelines and procedures. Information received from users was processed and used solely for purposes of this research. The confidentiality of responses and feedback received from users was also maintained. The developed platform was tested using applications for companies that have public vulnerability disclosure programs and security gaps were notified to them as and when they were identified. Finally, other researchers' work referenced in this study was properly cited and credited to avoid plagiarism.

Chapter 4: System Analysis and Architectural Design

4.1 Introduction

This chapter covers the system analysis and architectural design of the Android applications components auditing tool. This builds on the requirements gathered from the requirements collection phase. A detailed requirements analysis is conducted in this phase defining both functional and non-functional requirements. This is followed by an architectural system design representation of the application using various system modelling and design tools.

4.2 Requirements Analysis

This section reviews the needs or tasks that require to be met by the developed platform. These are categorised as functional and non-functional requirements.

4.2.2 Functional Requirements Analysis

Sommervile (2015) defines functional requirements as those operations and activities that the system should be able to perform. These represent the software's main features and functions describing the system behaviour under specific conditions. In this study, functional requirements were defined by expounding research objective two, which sought to review existing tools and address gaps identified in them. These requirements were defined as follows:-

i. **The platform should enable listing of all applications installed within the user device.**

   The list should highlight each application's name, package name and launcher icon. The user has an option to select any of the listed applications for review. The list should also be updated every time a new application is installed or deleted from the device.

ii. **The platform should support extraction of a selected Android application's security features automatically within the user device.**

   The designed and developed platform should extract static and dynamic features from a selected Android application within the user device. The analysis technique employed should ensure that these features can be extracted from all applications, whether obfuscated or not. The static features extracted from the Android Manifext.xml file should include requested permissions, signing information, installation and update

dates and components information. On the other hand, dynamic features should be collected through a log collector which include API and System calls, as well as data and function flows.

iii. **The platform should support analysis of Android application component features to identify their security gaps.**

The designed and developed platform should dynamically examine static and dynamic features as the application is executed on end user devices. These form a set of hybrid features that should be analysed to arrive at a decision on whether an application's component is improperly exported or not.

iv. **The platform should generate an analysis report as per user requirements.**

Analysing both static and dynamic features, the platform should be designed and developed to be able to generate a user report following three steps that include: installing and launching the platform, selecting the application to review and finally exporting the report.

v. **The platform should support secure connection to a remote data repository to update analysis rules.**

To support secure network connection. The application designed and developed should make use of HTTPS through SSL/TLS to securely connect back to the backend server through an API. This enables the application to security fetch data with a background service from the backend server. Secure HTTPS should be configured on the remote server. An SSL certificate should be generated and stored on the server, and this contains a public key that the client device will use for encryption when connecting back to the server.

4.2.3 Non-functional Requirement Analysis

Non-functional requirements define quality attributes and non-behavioural requirements of software. These are not mandatory as they mostly focus on how the software fulfils functional requirements. In this study, non-functional requirements were defined by analysing gaps from existing and related tools. These requirements were listed as follows:-

i. **The platform should be readily available for download from popular market stores.**

The Android application developed in this study should be readily available for download from Google Playstore, which is the dominant market store for Android

mobile applications. The application's APK file should be free to download and install from Google Playstore. Users should also be able to leave their comments as well as rate the application's functionalities through the same platform.

ii. **The platform should have an intuitive user interface.**

An intuitive user interface depicts the characteristics of an interface being clear, concise, responsive, familiar and efficient. Google Material Design composes of Material Components that are the interactive building blocks for intuitive user interfaces in Android applications. The application developed in this study should make use of material design components to deliver a user interface that ensures a concise process to deliver intended functionalities to the user. These components include App bars, Buttons, ScrollViews, RecyclerViews, and many more as discussed in system design section 4.4.

iii. **The platform should be efficient in CPU and storage space utilization.**

As defined by quality standards such as ISO/IEC 9126 and ISO/IEC 25010, software performance is measured by how fast it processes a task. For mobile devices, performance is closely related to energy consumption and hence a very paramount requirement. On the other hand, storage space utilization is an important feature especially in less expensive devices where its scarce and a large percentage of it is occupied by the operating system and factory pre-installed applications (Leonardo et al, 2019). Leonardo et al, (2019) observe that in order to reach a larger number of potential users, developers must optimize their application's performance as well as storage space utilization. These two features are determined by the approach the developers take to develop their applications. This study should employ the Android SDK design and development approach as it provides the most efficient approach for better performance and producing the smallest sized applications for optimized storage space utilization.

4.3 System Architecture

The application developed in this study is based on the recommended Model-View-ViewModel (MVVM) architecture for building robust and quality Android applications. The MVVM architecture supports separation of concerns making Activities and Fragments lightweight for easier writing of unit tests (Guide to app architecture, n.d.).

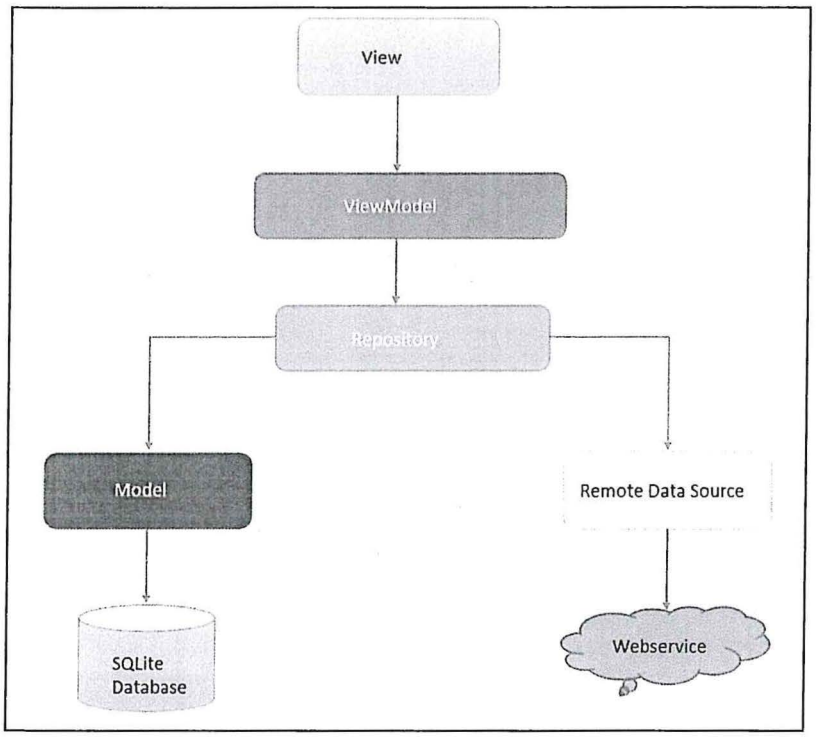


Figure 4.1: Model-View-ViewModel (MVVM) Architecture

The MVVM architecture is composed of three components that include:- a View, ViewModel and Model as further illustrated in Figure 4.1. The components are organised in such a way that each component depends on the component directly. This creates a consistent and pleasant user experience (Guide to app architecture, n.d.).

i. **View**

This is the layout users interact with and that informs the ViewModel about user's actions. This is represented by Activity and Fragment components and through data binding forwards user actions such as button clicks and scrolling to the ViewModel. Each view is usually attached to an .xml file that represents the UI layout definition for the screen. The view is usually the point of data input into the system and data output in form of reports. In this study, the view is the point where users select and customize the kind of reports they want to export from the system.

ii. **ViewModel**

This is the component that retrieves streams of data from the model when requested from the View. The ViewModel contains data-handling business logic for communicating with the model and provides data specific to the View such as an Activity or Fragment. In this study, data analysis and processing take place at the

ViewModel using some predefined rules and algorithm.

iii. **Repository**

The Repository works with the Model to fetch data from either a persistent data model such as an SQLite database or from a remote data source such as an API through webservice. In this study, an SQLite persistent data store is used to store the application's data while the repository module runs a background service to regularly update the data from the remote backend store.

iv. **Model**

The Model abstracts the data source and works with the ViewModel to get data from the date source or save it. In this study, the model works with the repository as well to parse the fetched static and dynamic features from a selected Android application for analysis.

4.4 System Design

This section details the system design process for the application developed in this study to aid in studying how the system components interact and function. Various design tools were used in this section that included data flow diagrams, context diagram, activity diagram, use case diagram, sequence diagram and wireframes.

4.4.1 Use Case Diagram

A use case diagram depicts the interactions between actors and the system. Such interactions are then described according to their specific types and names. Actors within a use case diagram can be human users or other systems (Sommervile, 2015). Figure 4.2 illustrates the use case diagram for platform developed in this study for auditing vulnerable android exported components.

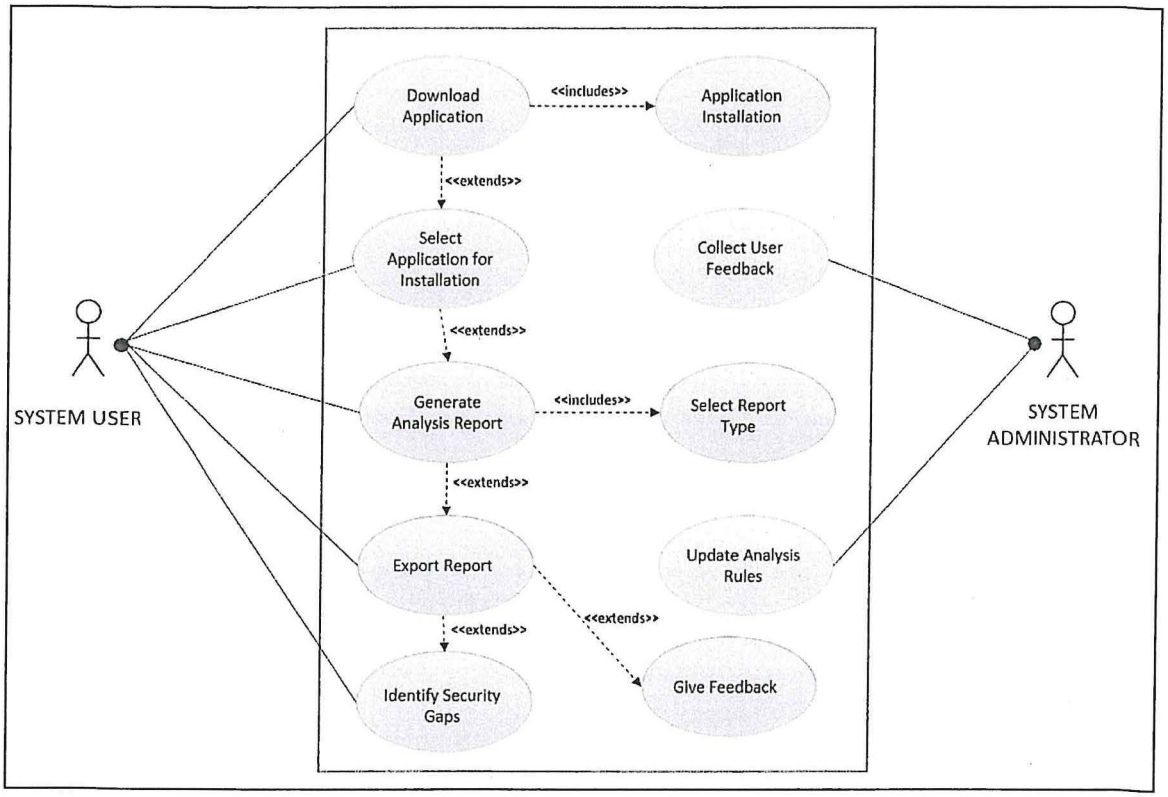


Figure 4.2: Android Components Auditing Platform Use Case Diagram

i. Download MobiSec Application Use Case Description

Table 4.1: Download MobiSec Application Use Case

Use Case Title	Download MobiSec Application
Description	This use case describes the steps taken by the user to download the application to auditing insecure Android components.
Actors	System User
Pre-Conditions	User has logged-in into their Google Playstore
Basic Flow	Search for the MobiSec Application for download. Download and install the MobiSec application Grant the MobiSec application the requisite permissions.
Post-Conditions	User can interact with the MobiSec application
Frequency of Use	Once

ii. Select Application for Analysis Use Case Description

Table 4.2: Select Application for Analysis Use Case

Use Case Title	Select Application for Analysis
Description	This use case describes the steps taken by the user when selecting an Android application for analysis.
Actors	System User
Pre-Conditions	User has installed the MobiSec application on their Android device
Basic Flow	Launch the MobiSec application. On the first page, scroll through the list of installed applications on the device. Press on the row containing the icon and name of the application to be analysed
Post-Conditions	Application to be analysed is selected
Frequency of Use	Many

iii. Generate Analysis Report Use Case Description

Table 4.3: Generate Analysis Report Use Case

Use Case Title	Generate Analysis Report
Description	This use case describes the steps taken by the user while generating an analysis report from the MobiSec application.
Actors	System User
Pre-Conditions	Application to be analysed is selected
Basic Flow	Navigate to the reporting tab on the MobiSec application Select one of the types of reports to generate from the list. Press the Generate Button to generate the report
Post-Conditions	Application analysis report is generated
Frequency of Use	Many

iv. Export Analysis Report Use Case Description

Table 4.4: Export Analysis Report Use Case

Use Case Title	Export Analysis Report
Description	This use case describes the steps taken by the user to export analysis reports from the MobiSec application.
Actors	System User
Pre-Conditions	Application analysis report has been generated
Basic Flow	Click on the export button Select the report export format Select the export location Click on the finish button to finalise the process
Post-Conditions	Application analysis report is exported into the device memory
Frequency of Use	Many

v. Identify Security Gaps Use Case Description

Table 4.5: Identify Security Gaps Use Case

Use Case Title	Identify Security Gaps
Description	This use case describes the steps taken by the user to identify security gaps from the exported analysis report
Actors	System User
Pre-Conditions	Application analysis report is exported
Basic Flow	User opens the exported reports User reviews the noted security gaps and remediation suggestions.
Post-Conditions	Application's exported components security gaps are identified for remediation
Frequency of Use	Many

vi. Give User Feedback

Table 4.6: Give User Feedback Use Case

Use Case Title	Give User Feedback

Description	This use case describes the steps taken by the user to give feedback to the developers and system administrators after using and testing the platform
Actors	System User
Pre-Conditions	Application is installed and used
Basic Flow	User logs in to the Google account User goes to the MobiSec application's download page User writes their feedback User submits their feedback
Post-Conditions	User feedback is saved
Frequency of Use	Many

vii. Collect User Feedback

Table 4.7: Collect User Feedback Use Case

Use Case Title	Collect User Feedback
Description	This use case describes the steps taken by the system administrator after the platform user gives their feedback
Actors	System Administrator
Pre-Conditions	Application uploaded on Google Play Store
Basic Flow	Administrator collects feedback from the Google Play form Administrator reviews feedback Administrator keys in the feedback information on the backend form
Post-Conditions	User feedback is reviewed and entered on the backend form
Frequency of Use	Many

viii. Update Analysis Rules

Table 4.8: Update Analysis Rules

Use Case Title	Update Analysis Rules
Description	This use case describes the steps taken by the system administrator to update analysis rules.

Actors	System Administrator
Pre-Conditions	Backend system already set up
Basic Flow	Administrator logs in into the backend server Administrator updates the list of rules using the backend database form
Post-Conditions	Analysis rules updated
Frequency of Use	Many

4.4.2 Entity Relationship Diagram

An Entity Relationship Diagram (ERD) represents the database design structure and shows relationships between entities. Figure 4.3 depicts the ERD for the development platform in this study.

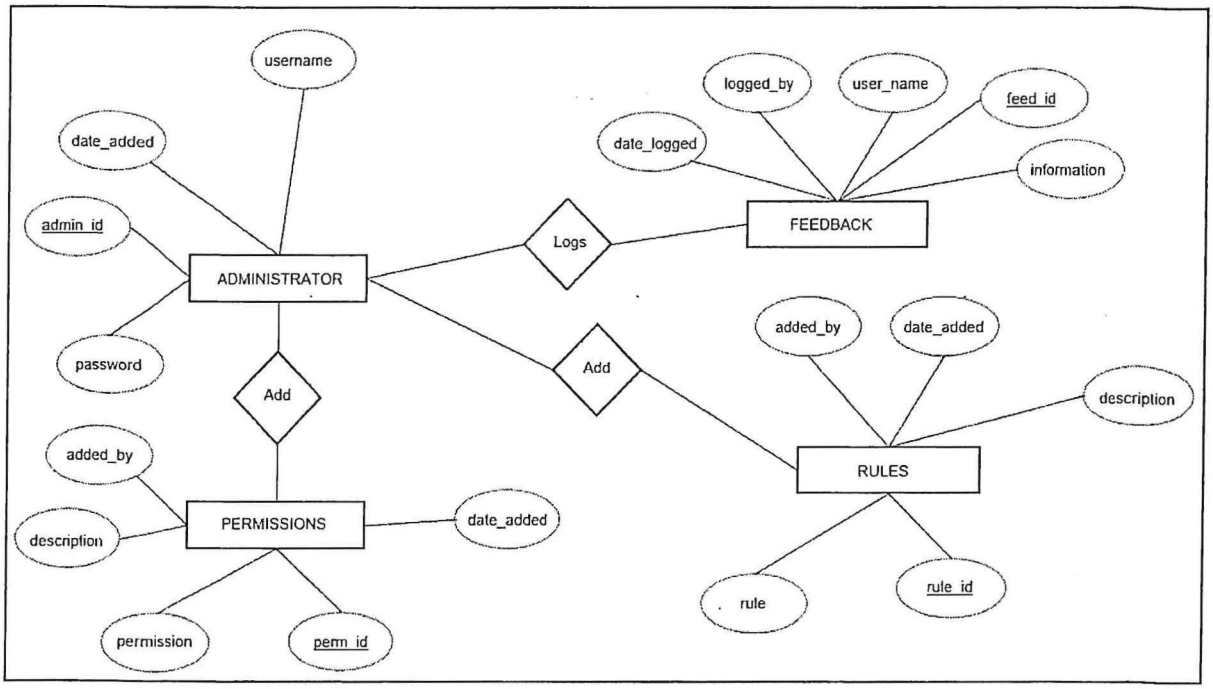


Figure 4.3: Android Components Auditing Platform ERD

The ERD for Android components audit platform consists of four entities that include; the Administrator, Permissions, Rules and Feedback. Each of these entities consists of several attributes including primary keys. These attributes are expounded in the section while describing the database schema.

4.4.3 Database Schema

A database schema is a structure that represents a logical view of an entire database. Figure 4.4 represents a database schema for the developed platform to audit vulnerable Android application components.

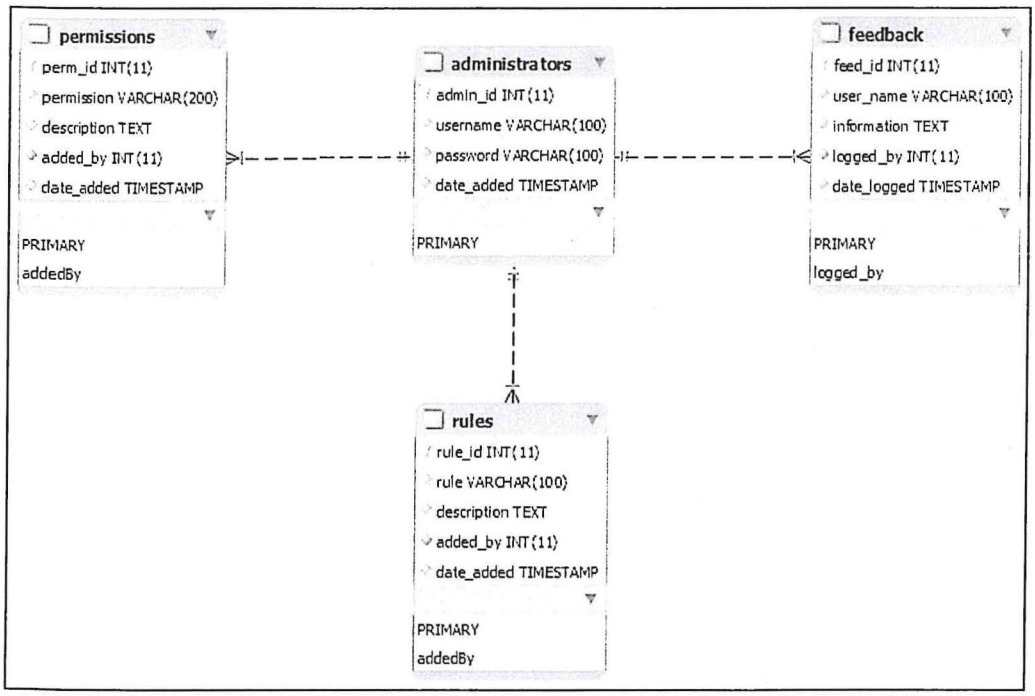


Figure 4.4: Android Components Auditing Platform Database Schema

The database schema tables and fields are described.

The tables describe the fields for each table in database schema. The primary keys are abbreviated as PK, foreign keys as FK, whereas autoincrement fields are abbreviated AI.

Administrators Table

Table 4.9: Administrators Table

Filed	Data Type (Field Length)	Details	Notes
admin_id	Integer (11)	PK, AI	Serves at the Primary key for each administrator record and autoincrements every time a new record is added.
Username	Varchar (100)	UNIQUE	Used during login and is unique to

			identify every administrator.
Password	Varchar (100)		Captures the MD5 hashed password for the specific user
date_added	Timestamp		Holds the date and time record for when an administrative user was added.

Rules Table

Table 4.10: Rules Table

Filed	Data Type (Field Length)	Details	Notes
rule_id	Integer (11)	PK, AI	Serves at the Primary key for each rule record and autoincrements every time a new record is added.
Rule	Varchar (100)	UNIQUE	Captures each rule's unique identifying string
Description	TEXT		Records descriptive details with more information about each rule
added_by	Integer (11)	FK	Serves as a Foreign Key referencing the administrators table to record who added each specific rule
date_added	Timestamp		Holds the date and time record with details for when a rule was added.

Permissions Table

Table 4.11: Permissions Table

Filed	Data Type (Field Length)	Details	Notes
rule_id	Integer (11)	PK, AI	Serves at the Primary key for each Android

			permission record and autoincrements every time a new record is added.
Permission	Varchar (200)	UNIQUE	Captures each permission's unique identifying string
Description	TEXT		Records descriptive details with more information about each Android permission
added_by	Integer (11)	FK	Serves as a Foreign Key referencing the administrators table to record who added each specific permission
date_added	Timestamp		Holds the date and time record with details for when a permission was added.

Feedback Table

Table 4.12: Feedback Table

Filed	**Data Type (Field Length)**	**Details**	**Notes**
feed_id	Integer (11)	PK, AI	Serves at the Primary key for each feedback record and autoincrements every time a new record is added.
user_name	Varchar (100)	UNIQUE	Captures the name for the user who provided feedback
Information	TEXT		Records details about the feedback given by user
logged_by	Integer (11)	FK	Serves as a Foreign Key referencing the administrators table to record who logged each specific feedback item

date_logged	Timestamp		Holds the date and time record with details for when a feedback item was logged.

4.4.4 Sequence Diagram

A sequence diagram models interactions between actors and the system as well as between system components. A sequence diagram shows the sequence of interactions in the order in which they occur for a particular use case instance (Sommervile, 2015).

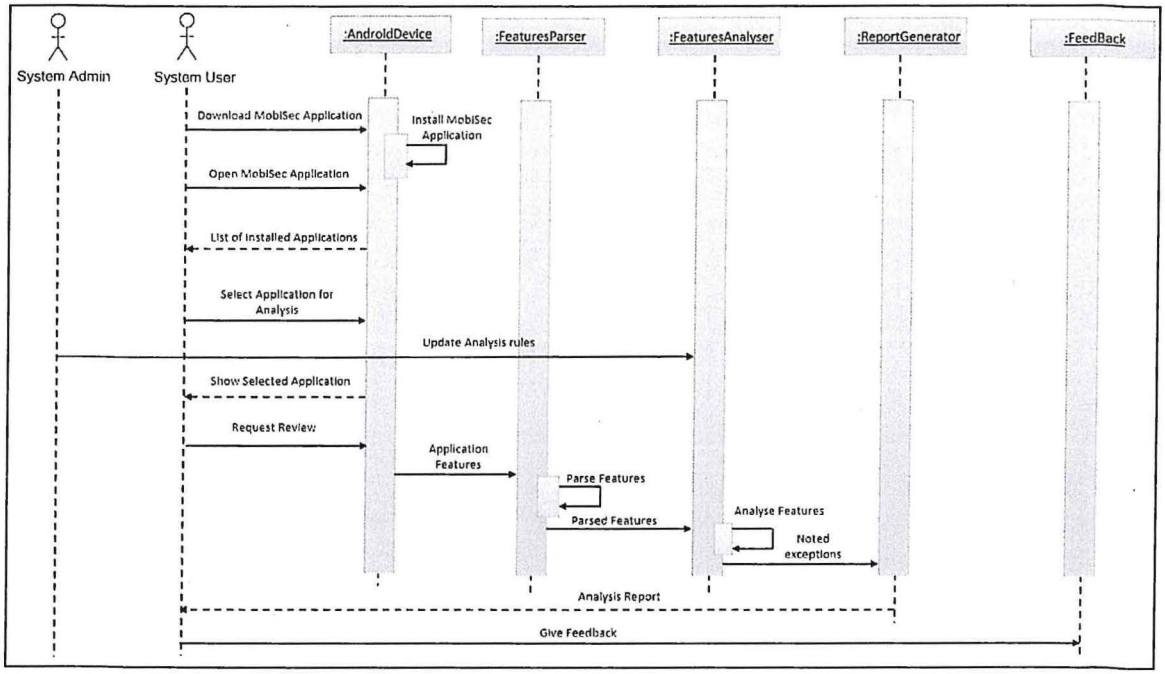


Figure 4.5: Android Components Auditing Platform Sequence Diagram

The following is a brief description of each of the key processes within the developed platform sequence diagram as illustrated in Figure 4.5.

- **Downloading and Installing the Application** – User interaction with the platform starts when the user downloads and installs the MobiSec application from Google Playstore. The user needs to have an updated version of Google Play Services and a supported Android device.
- **Selecting Application for Analysis** - After launching the MobiSec application, on the first page, the user is presented with a list of all installed applications and is able to

select one of the applications for review.

- **Features Parsing** - Through a parser, static and dynamic features are extracted from the selected application and then passed on to the analyzer for analysis.
- **Features Analyses** – The analyzer reviews the parsed features by validating the configurations against a set of rules and notes exceptions that can be considered a security risk to the application. These are passed on to the report generator for report preparation.
- **Report Generation** - The report generator organizes the report into the format the user desires and gives them the option for exporting it in different formats.
- **Feedback collection** – The user has an option to give feedback after testing and using the application. This is collected through the Google Pay feedback form.
- **Rules Update** – The system administrator occasionally updates the analysis rules especially when some updates are done on the Android platform.

4.5 Security Design

The platform developed in this study implements security by design as recommended by Google by enforcing security controls around network connection, permissions usage and internal data storage as further described.

4.5.1 SSL/TLS Certificate Pinning

To support secure network connectivity, the application developed in this study makes use of SSL/TLS to securely connect back to the backend server through an API. This enables the application to security fetch data with a background service from the backend server. HPKP was also used to fortify HTTPS-based pinning against MiTM attacks as illustrated in Figure 4.6.

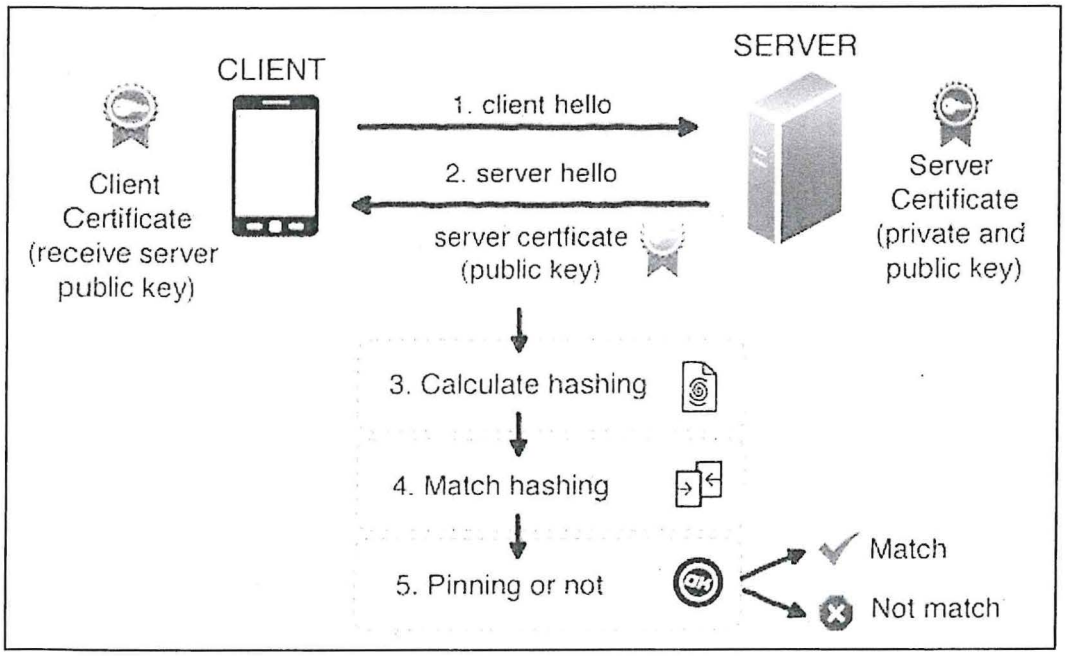


Figure 4.6: Android Application SSL Certificate Pinning Process

Source: Ramírez-López et al, (2019)

SSL/TLS certificate pinning is implemented through public key cryptography where an X.509 certificate is generated. The public key was shared with the Android clients as they connected to the server. The private key is on the other hand securely stored on the remote server and protected with file system permissions. The keys were set to expire after every year to ensure that they are regularly rotated in case of a compromise.

The certificate pinning process is implemented as a two-stage process where the mobile device first initiates communication with the server and the server responds whether it is active or not and sends its public key to the client. In the second step, the pinning process takes place where the mobile device or client checks the received certificate from the server against the locally saved certificate and if there is a match, the communication is allowed to continue.

4.5.2 Secure Use of Permissions

To ensure the principle of least privilege, the application developed in this study requests the minimum number of permissions necessary to function properly and where possible relinquishes some of these permissions when they are not needed. Additionally, intents have been used to defer permissions. This is applicable where actions which can be completed by other applications are passed on to them through intents instead of having the MobiSec

application request those permissions. The MobiSec platform requests for two permissions. The first permission allows the application to access the internet to allow connecting to the backend server for updating the analysis rules. The user is given an option to enable this permission only when they need to query for updates. The second permission allows the application to access the device's internal storage. This enables the application to save export reports into the user device.

4.5.3 Secure Internal Data Storage

All the sensitive data from the application developed in this study such as exported analysis reports are stored securely within the device internal storage. The exported data is saved in a restricted folder within the application and is only accessible by the MobiSec application.

A folder is created in the device to save exported reports. These reports are saved with a package name for the application reviewed together with a timestamp date. The saved reports are cleared every 24 hours after their export to reduce their risk of exposure.

4.6 Wireframes

Listing Installed Android Applications

MobiSec enables the listing of all applications installed within the user device. Each application is listed by its package name and launcher icon as illustrated in Figure 4.7.

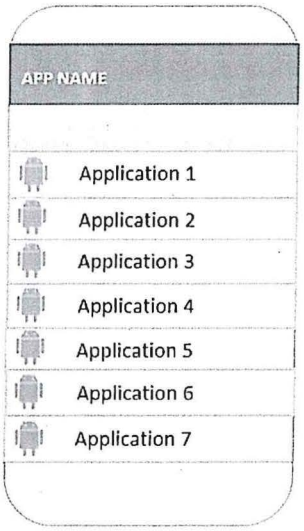


Figure 4.7: Wireframe Showing List of Installed Applications

Selecting Application for Review

As illustrated in Figure 4.8, the developed application displays basic details of an application when it is selected which include package name, version, date installed, date lastly updated and permissions. This view also has tab options to view a summary of audit findings per each component class as well as export the whole report.

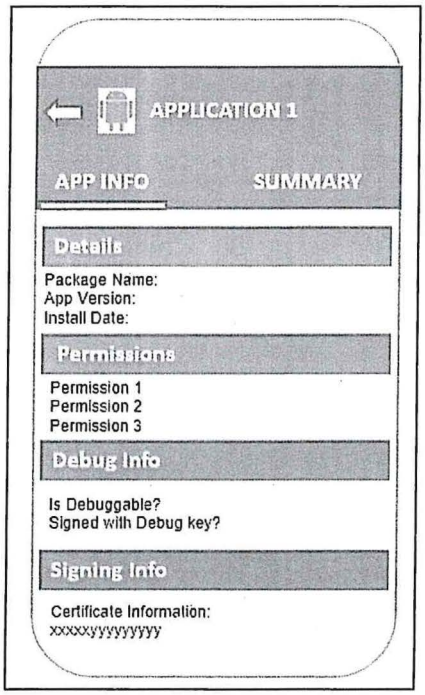


Figure 4.8: Wireframe Showing Application Selected for Analysis

Generating Audit Report

As illustrated in Figure 4.9, the developed MobiSec application will generate a report showing exceptions noted with the export of Android application components. A summary will be presented within this view providing the option to export the detailed report for further review.

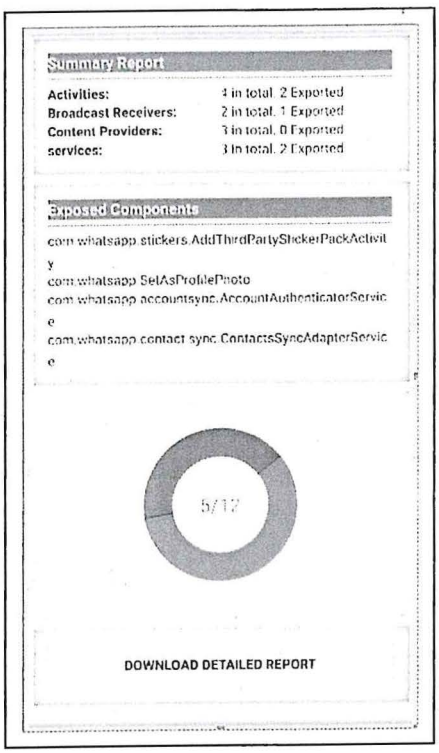


Figure 4.9: Wireframe Showing Analysis Report Summary

Chapter 5: System Implementation and Testing

5.1 Introduction

This chapter describes the implementation process for the system to audit for vulnerable Android application components. This focuses on highlighting key functionalities and system specifications. Additionally, this chapter discusses the various tests that were conducted on the system to validate its efficiency and effectiveness in identifying vulnerable Android application components.

5.2 Implementation Environment

5.2.1 Hardware Environment Specification

The following hardware specifications need to be met for successful installation and optimal operation of the MobiSec platform.

Android Device Hardware Requirements

The minimum Android device hardware requirements are as listed :-

- At least 512 MB RAM
- At least 1 GB of Internal device memory
- At least ARMv7 CPU with 1.0 GHz
- Support for Wi-Fi, 3G or 4G to work with data services

Backend Server Hardware Requirements

If the system administrator opts to run their backend server to support their MobiSec Android platform instance. They will need to set up the backend server with the following minimum hardware requirements:-

- 1 GB RAM Memory
- 25 GB Disk Space
- Dual-core processor with 2.4 GHz speed

5.2.2 Software Environment Specification

Android Device Software Requirements

The minimum requirements required to install and run the MobiSec platform are:-

- Android phone running OS 4.4.4 KitKat and newer
- Android OS with the latest version of the Google Play Services app

Backend Server Software Requirements

The following minimum software requirements must be met by the backend system supporting the MobiSec application API:-

- Windows Server 2012 or Unix based Operating System
- MySQL database system version 8.0.21 and above
- Apache Web Server version 2.4.6 or above

5.2.3 Network Environment Specification

The MobiSec platform employs the client-server network topology where a single powerful server services requests from multiple client devices. The clients connect to the remote central server through a network connection. The Android clients in this case need to support Wi-Fi, 3G or 4G network connectivity to be able to connect to the remote server hosting the APIs. The server on the other hand needs to be connected to the internet and configured with a public IP address.

5.3 Implementation Methodology

A simple rule-based classification System based on the CART analysis methodology was used to create as set of rules for implementation of the system to identify vulnerable Android application components.

CART analysis recursively splits a set of data into binary divisions until all homogeneous data is placed in a similar class membership. A series of rules are used to predict the likely class membership for unknown observations. The set of attributes in table 5.1 are evaluated as extracted from static and dynamic application features.

Table 5.1: Component Analysis Attributes

#	**Attribute**
1.	Exported Attribute component setting in Manifest.xml file
2.	Component has an intent-filter
3.	Component has a permission protection setting
4.	Activity is the launching activity

As depicted in Figure 5.1 , CART analysis is employed to iterate through application component attributes and determine if a component is exported or exposed to component-based attacks.

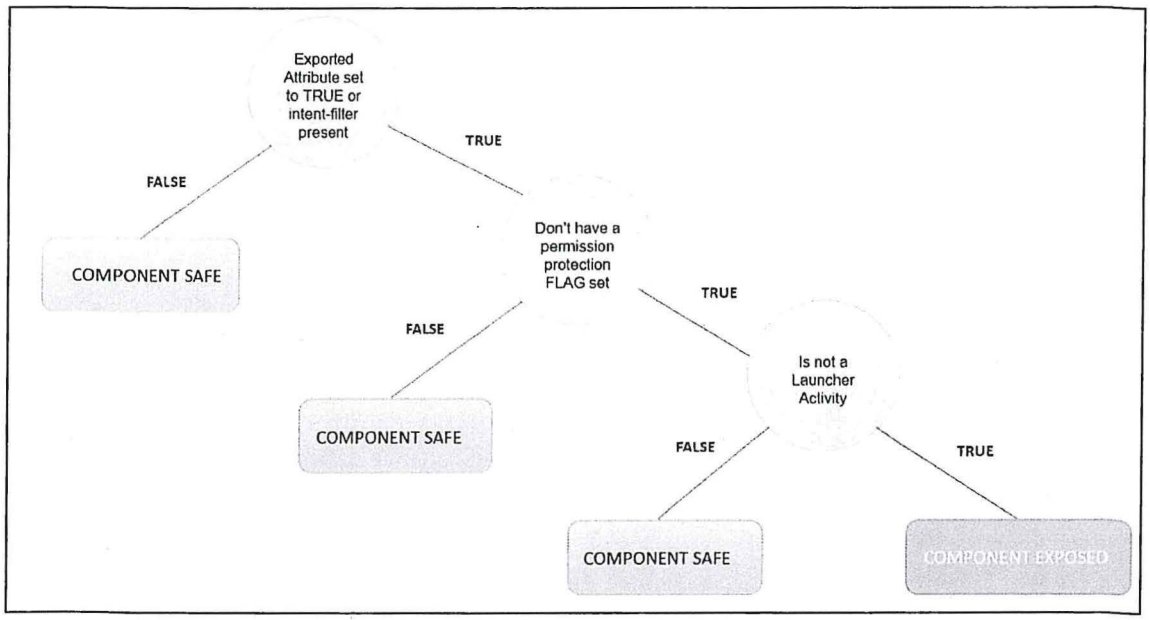


Figure 5.1: Android Component Analysis Regression Tree

5.4 System Modules

The developed platform composes of several modules both at the backend and frontend. These include user interfaces that users interact with at the frontend as well as various data processing and storage modules at the backend. These are organised as per the various platform functionalities.

5.4.1 Getting List of Installed Applications

The first page of the application allows the user to select one application for analysis from the displayed list of installed applications on the user's device. This list highlights each application's name, package name and launcher icon as depicted in the screenshot in Figure 5.2.

Figure 5.2: Installed Applications Listing on the First Page

The Android PackageManager class allows querying of installed applications' details including package name, application name and application icon as shown in Figure 5.3.

```
private void loadApps() {
    PackageManager pm = mContext.getPackageManager();
    List<ApplicationInfo> packages = pm.getInstalledApplications( flags: 0);
    for (ApplicationInfo packageInfo : packages) {
        if(pm.getLaunchIntentForPackage(packageInfo.packageName) != null) {
            // apps with launcher intent
            if((packageInfo.flags & ApplicationInfo.FLAG_UPDATED_SYSTEM_APP) != 0) {
                // updated system apps
            } else if ((packageInfo.flags & ApplicationInfo.FLAG_SYSTEM) != 0) {
                // system apps
            } else {
                // user installed apps
                AppInfo newApp = new AppInfo();
                newApp.setAppName(getApplicationLabelByPackageName(packageInfo.packageName));
                newApp.setAppPackage(packageInfo.packageName);
                newApp.setAppIcon(getAppIconByPackageName(packageInfo.packageName));
                myApps.add(newApp);
            }
            //myApps.add(newApp);
        }
    }
    Collections.sort(myApps, new Comparator<AppInfo>() {
        @Override
        public int compare(AppInfo s1, AppInfo s2) {
            return s1.getAppName().compareToIgnoreCase(s2.getAppName());

        }
    });
}
```

Figure 5.3: Android PackageManager Class Allows Listing of Installed Apps

5.4.2 Getting Application Component Information

In a tabbed view the system displays selected application's details that include package name, version, date installed, date lastly updated, permissions, debug information and signing certificate information as illustrated in Figure 5.4.

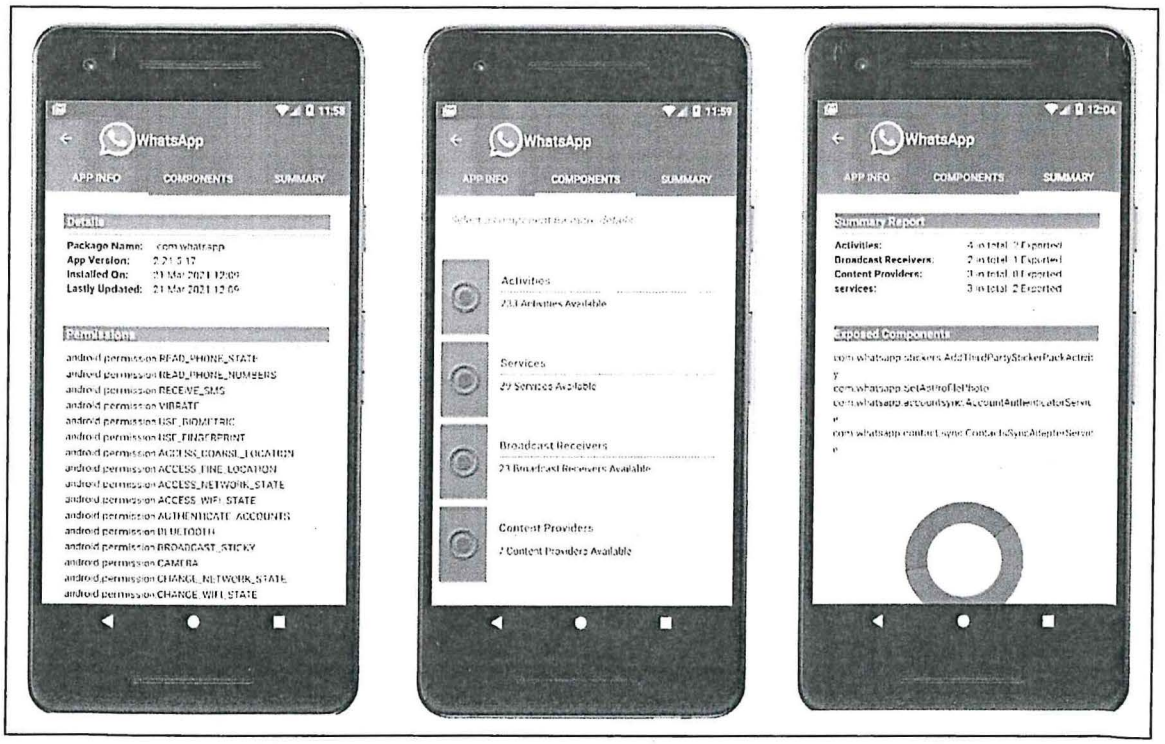


Figure 5.4: Tabbed Layout Showing Application Security Configurations

This view also shows a summary report of application components, noting those which are improperly exported.

```
public void getActivitiesInfo() {
    LinkedHashMap<String, String> actvs = new LinkedHashMap<>();
    try {
        PackageInfo packageInfo = getActivity().getPackageManager().getPackageInfo(app_package, PackageManager.GET_ACTIVITIES);
        ActivityInfo[] activities = packageInfo.activities;
        if (activities != null) {
            for (ActivityInfo activityInfo : activities) {
                boolean bool = activityInfo.exported;
                actvs.put(activityInfo.name, String.valueOf(bool));
            }
        }
    } catch (PackageManager.NameNotFoundException ignored) {}
    compItemsList.clear();
    for (Map.Entry<String,String> entry : actvs.entrySet()) {
        myarray=new ArrayList<CompItems>();
        String key = entry.getKey();
        String  value = entry.getValue();
        CompItems items = new CompItems(key, value);
        Log.d(TAG, msg: "Fragment Items:- " + items);
        compItemsList.add(items);
    }
    compItemsAdapter.notifyDataSetChanged();
}
```

Figure 5.5: Android PackageInfo Class to Query Component Information

The PackageInfo class from the PackagerManager as illustrated in Figure 5.5. allowed querying of application components' dynamic features as collected from the AndroidManifest.xml file. This information included details on whether a component is set to be exported, active or as the application's launching activity.

5.4.3 Generating an Analysis Report

Based on a simple rules analysis, the developed system generates an analysis report to alert the user on safe and exposed components as depicted in Figure 5.6.

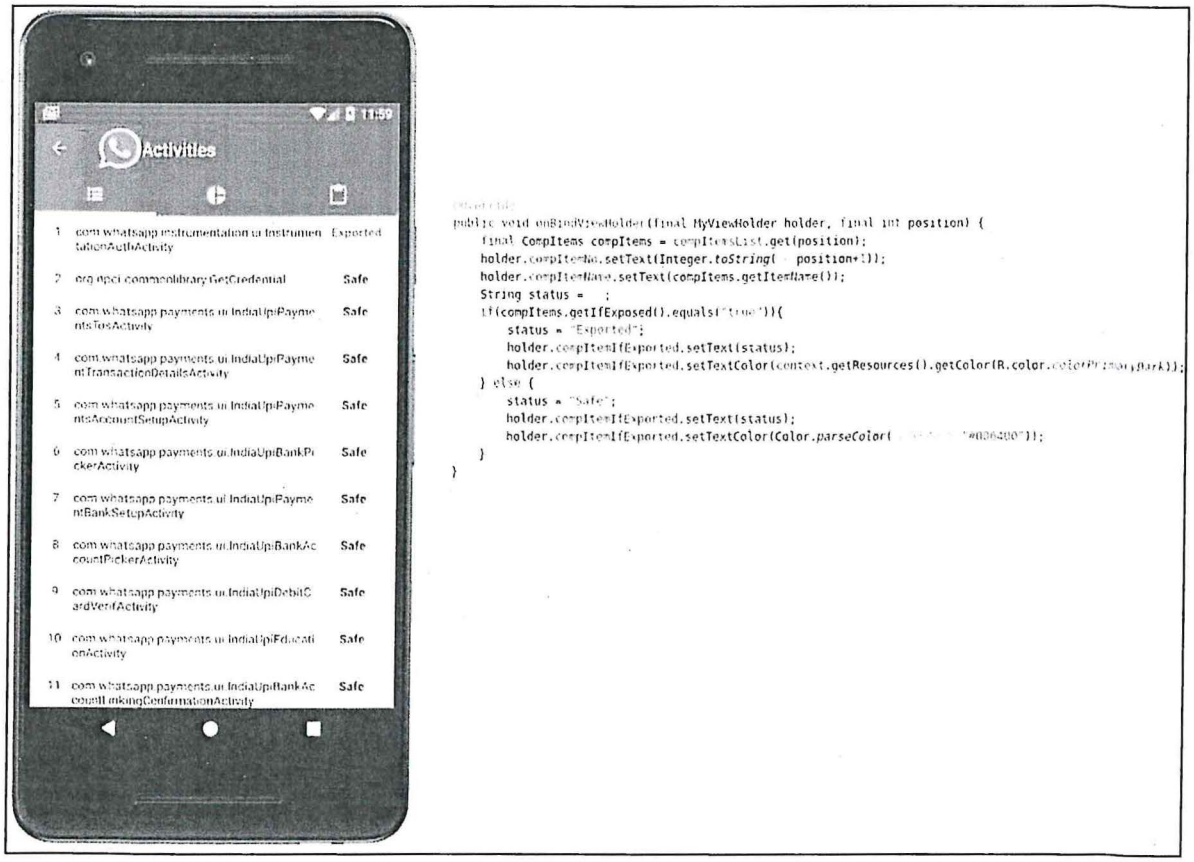


Figure 5.6: Activities Report Alerting on Safe and Exposed Components

Depending on the kind of component, the generated report advises the user on remediation steps for each exposed component.

5.4.4 API Connection to Backend Server

The MobiSec platform includes a backend API and database as well as a local application SQLite database to host a set of analysis rules. The mobile device regularly connects to the backend server through the API to update the rules database.

The backend API was developed using Laravel Sanctum to support token-based authentication. A valid user email and password are required to generate an API access token as illustrated in Figure 5.7.

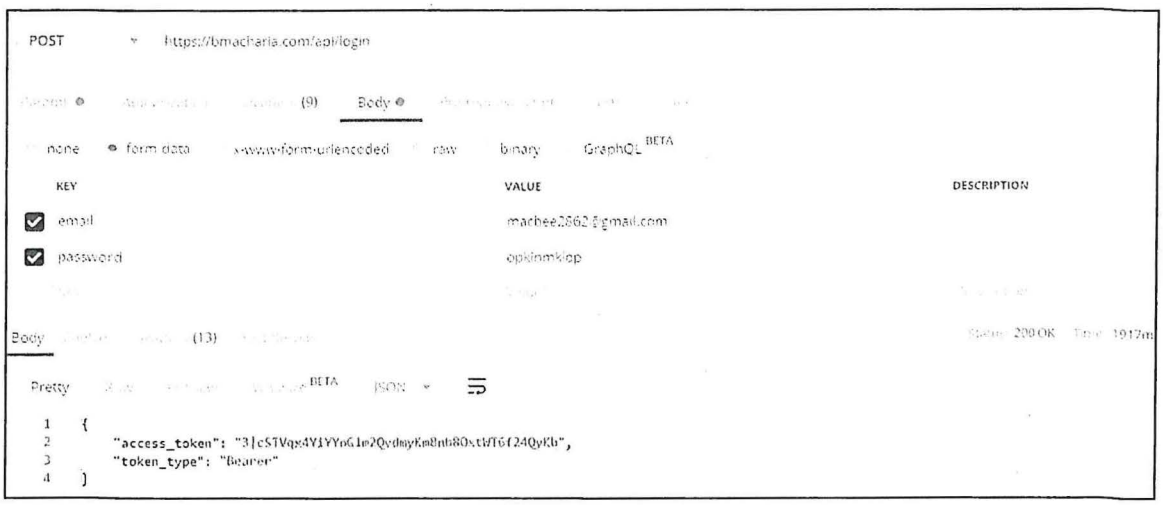


Figure 5.7: Generating Bearer Token for User Authentication

The bearer token is used for authentication on some of the restricted API end points as noted in the list of exposed end points in table 5.2.

Table 5.2: Exposed Backend API Endpoints

API	Method	Parameters	Authentication	URL
Registration	POST	name email password	None	https://bmacharia.com/api/register
Login	POST	email password	email and password	https://bmacharia.com/api/login
View Profile	POST	None	Bearer Token	https://bmacharia.com/api/me
Query Rules	GET	None	None	https://bmacharia.com/api/rules
Query Permissions	GET	None	None	https://bmacharia.com/api/permissions
Query User Feedback	POST	None	Bearer Token	https://bmacharia.com/api/feedback

The APIs for querying user permissions and rules are not protected by a token as they are meant to be consumed by mobile applications through a background service to provide updates, analysis rules and permissions.

Local SQLite Database

Through the API, the developed platform regularly queries the rules and permissions from the remote database and stores them locally within the device in an SQLite database.

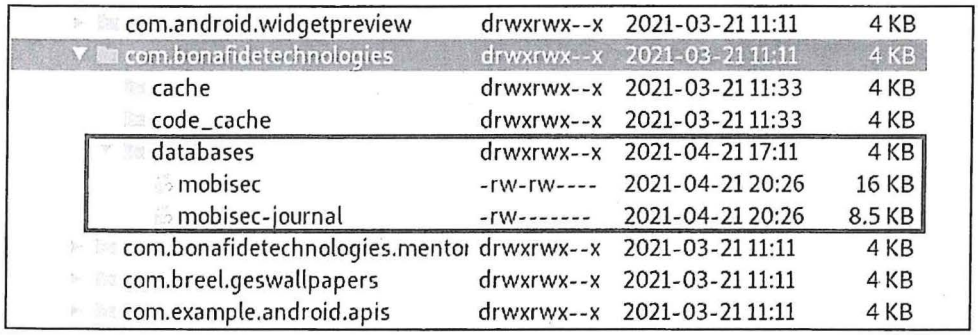


Figure 5.8: SQLite Database Saved Locally in the Device

The local SQLite database is stored in the data folder within the MobiSec application package folder as illustrated in Figure 5.8. The database is lightweight and persists upon device reboot and application updates. Also, to maintain consistency, the database is versioned and older versions are deleted after every update to save memory space.

5.4.5 Backend System Administration

A backend web-based administrative portal was created using Laravel Jetstream to manage the processes for adding analysis rules, permissions and for logging user feedback. Every administrator is required to create an account and log in before they can perform the mentioned administrative tasks.

The screenshots in Appendix D illustrates all the processes the administrator can perform after they log in into the administrative portal.

5.4.6 Security Implementation

To support secure network connectivity, the application developed in this study makes use of SSL/TLS to securely connect back to the backend server through an API. SSL/TLS certificate

pinning was implemented through public key cryptography where an X.509 certificate is generated. The following command was used to generate the certificate and keys in this study.

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/ssl/private/apache-selfsigned.key -out /etc/ssl/certs/apache-selfsigned.crt

The public key was shared with the Android clients as they connected to the server. The private key is on the other hand securely stored on the remote server and protected with file system permissions.

To abide by the principle of least privilege, the developed MobiSec platform implements only the required permission to function. These include the internet connection permission and permission to write to external storage as illustrated in Figure 5.9.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bonafidetechnologies.mobisec">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="MobiSec"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/MyMaterialTheme">
        <activity android:name=".activity.MainActivity"
            android:label="MobiSec"
            android:theme="@style/MyMaterialTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
```

Figure 5.9: Android Manifest.xml Defining Application Permissions

5.5 System Testing

This section highlights the different types of testing that were carried out on the developed platform. These included Unit Testing, Functional Testing, Compatibility Testing, UI Testing and Performance Testing.

5.5.1 Unit Testing

Unit testing was implemented and carried out during the development phase. Both local and instrumented unit tests were carried out using the JUnit 4unit testing framework for java applications and the Mockito framework that mocks required dependencies during testing.

Each function within every class was tested individually to identify any errors that may arise when its executed as illustrated in Figure 5.10.

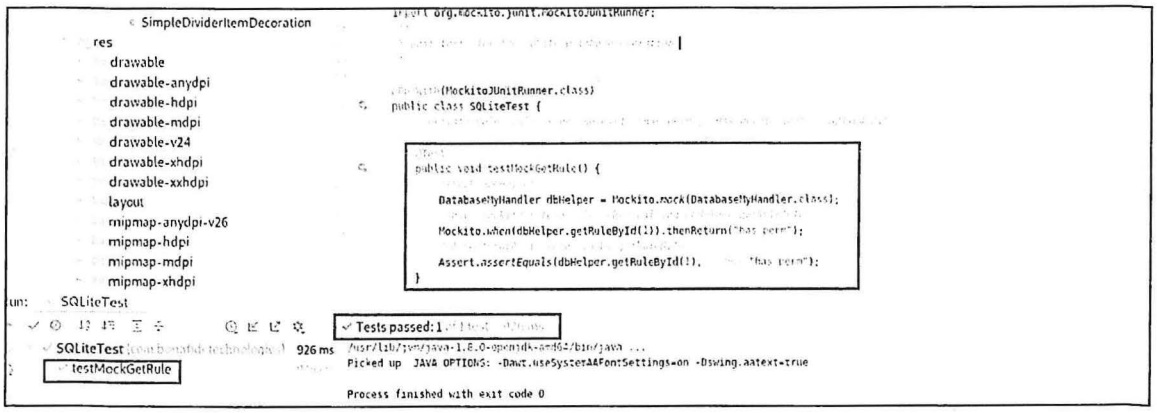


Figure 5.10: Application Running in a Tablet Device Emulator

Each test for every critical function was ensured to have a passed state before the whole application could be compiled for deployment.

Table 5.3: Critical Functions tested in Unit Testing

	Function Tested	**Class**	**Description**	**Test Result**
1	testMockGetRule	SQLiteTest	Test the process of querying rules from the local SQLite database	Pass
2	testMockUpdateRule	SQLiteTest	Test the process of updating of rules in the local SQLite database	Pass
3	testMockGetAPIData	ReviewAcivity	Test the process of querying API data from the backend server	Pass
4	testMockGetPermission	SQLiteTest	Test the process of querying permissions from the local SQLite database	Pass
5	testMockUpdatePermission	SQLiteTest	Test the process of updating permissions in the local SQLite database	Pass
6	testMockGetStaticFeatures	ReveiwActivity	Test the process of extracting static application features from a selected application	Pass
7	testMockGetDynamicFeatures	ReveiwActivity	Test the process of extracting dynamic application features from a selected application	Pass
8	testMockGetParsedFeatures	ReveiwActivity	Test the process of parsing features extracted from a selected application	Pass
9	testMockReviewFeatures	ReviewAcivity	Test the process for analysing a parsed application features to identify security vulnerabilities	Pass
10	testMockGenerateReport	ReportActivity	Test the process for generating user reports	Pass

Table 5.3 gives a list of the critical functions that were tested during the unit testing phase of software development and testing. To finalise on unit testing, interoperability tests between the individually tested functions within a class was also conducted before building the whole application.

5.5.2 Functional Testing

Functional testing was conducted to validate that the developed platform met the functional requirements. All the use case flows were tested as well as the business logic. Specifically, the tested use cases included:- reviewing any user-installed applications within the device, generating analysis reports as per user-selected format and remote API connection to update analysis rules.

Reviewing Random Applications on User Device and Emulator

The developed platform was tested with random applications downloaded from various marker stores such as Google Playstore and Samsung Galaxy Apps store as well as those obtained from developers but not yet deployed into public stores.

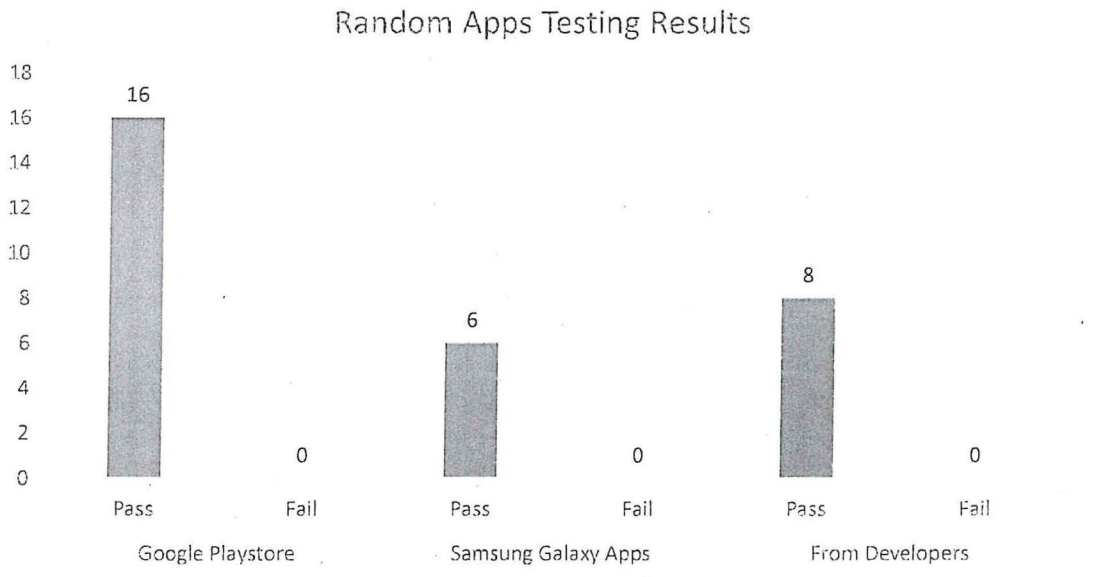


Figure 5.11: Testing the Developed Platform with Different Applications

As illustrated in Figure 5.11, the developed platform was tested with applications from different sources. Specifically, the tests were conducted using 16 applications downloaded from Google

Playstore, 6 applications from Samsung Apps Store and 8 applications obtained directly from developers. All the tests showed that a consistent review process was applied to generate the analysis report.

Generating User Report

The process for generating user reports was also tested out on the developed platform. Reports were generated in the two supported formats that are HTML and PDF. This was done in actual Android devices and in Emulators for the different supported Android versions.

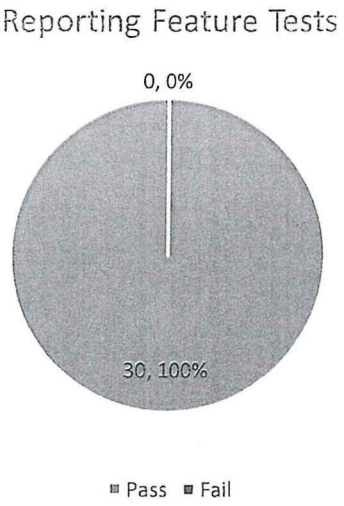


Figure 5.12: Reporting Feature Tested on Different Android Versions

The results of this test as illustrated in Figure 5.12 showed that a consistent report was generated from 29 devices of the total 30. The test failed in only one device running Android version 4.4.

Remote API Connection

Functionality tests were also conducted on the process of connecting to the backend server through the API. This also included tests on loading API data into the local SQLite database to update rules and permissions. Figure 5.13 shows the test results when the developed platform was tested on different Android devices.

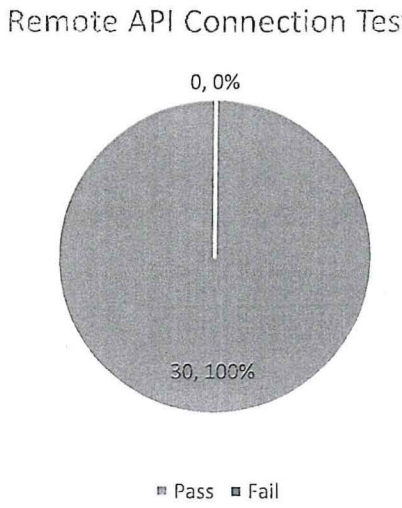


Figure 5.13: Remote API Connection Tested on Different Android Devices

The test result was a pass on all the 30 tested devices and emulators. All devices could successfully initiate a connection to the backend server through the API and query data to synchronise with the local SQLite database.

5.5.3 Compatibility Testing

These tests were conducted to evaluate the developed platform compatibility with a selected number of Android devices. The end-to-end process from installation to analysis of any random application and generation of reports was tested on each device. Table 5.4 highlights the results from the test.

Table 5.4: Reporting Feature Tested on Different Android Versions

Codename	Version	API Level	Tests Done	Pass	Fail
Android11	11	API level 30	1	1	0
Android10	10	API level 29	2	2	0
Pie	9	API level 28	3	3	0
Oreo	8.1.0	API level 27	5	5	0
Oreo	8.0.0	API level 26	4	4	0
Nougat	7.1	API level 25	3	3	0
Nougat	7	API level 24	4	4	0
Marshmallow	6	API level 23	3	3	0
Lollipop	5.1	API level 22	1	1	0
Lollipop	5	API level 21	2	2	0
KitKat	4.4 - 4.4.4	API level 19	2	1	1
Total			**30**	**29**	**1**

Percentage			100%	97%	3%

The test result was a pass on 29 devices running different Android versions which represents 97% of the total tests conducted. A fail result was noted in one of the devices running Android version 4.4. An error of an incompatible package was noted on this device while generating the analysis report in PDF format.

5.5.4 UI Testing

UI testing is a user-centric testing that tests the look and feel of the developed application on different screens. This concentrates on testing font visibility, size, resolution and data alignment on a selected number of screens of varying sizes as well as different orientations.

Table 5.5: UI Testing on Selected Screen Sizes

Device	Screen Size	Resolution	Test Result
HTC One X	4.7 "	1280 x 720	Pass
Nexus 5	5"	1920 x 1080	Pass
Galaxy Note III	5.7"	1920 x 1080	Pass
HTC One Max	5.9"	1920 x 1080	Pass
Galaxy Note II	5.6"	1280 x 720	Pass
Nexus 4	4.4"	1200 x 768	Pass
Galaxy Mega	6.3"	1280 x 720	Pass
Kindle Fire HD	7"	1280 x 800	Pass
Galaxy Mega	5.8"	960 x 540	Pass
Sony Xperia Z Ultra	6.4"	1920 x 1080	Pass
Nexus 7	7"	1280 x 800	Pass
Kindle Fire HD	8.9"	1920 x 1200	Pass
Galaxy Tab 2	10"	1280 x 800	Pass
ASUS Transformer 2	10"	1920 x 1200	Pass
Nexus 10	10"	2560 x 1600	Pass
Pass			**100%**
Fail			**0%**

On all the tested devices with different screen sizes and resolutions, the test result was registered as a pass if for all interfaces as shown in the table 5.5. all the user interaction components such as buttons and scroll bars were well visible within the screen.

5.5.5 Performance Testing

For this test, the developed platform performance was evaluated under circumstances where the selected devices were having low memory, low battery levels and poor network reception. To test on performance, the application was tested on a selected device and the report generation module observed on the time it took to execute given different memory and power states. The results for this test are as illustrated in table 5.6.

Table 5.6: Memory and Power Consumption Performance Test

Memory Performance Test				Power Consumption Performance Test		
RAM (MB)	**Time (ms)**	**Result**		**Battery (%)**	**Time (ms)**	**Result**
100	XX	Fail		10%	XX	Fail
200	2500	Pass		20%	1500	Pass
300	2000	Pass		30%	1000	Pass
400	2000	Pass		40%	1000	Pass
500	2000	Pass		50%	1000	Pass
600	1000	Pass		60%	1000	Pass
700	1000	Pass		70%	1000	Pass
800	1000	Pass		80%	1000	Pass
900	500	Pass		90%	1000	Pass
1000	500	Pass		100%	1000	Pass
Pass		**90%**				**90%**
Fail		**10%**				**10%**

The testing device failed to generate either the PDF or HTML-based report when the memory was 100MBs. The device froze and did an automatic restart when the report generation button was pressed hence the test was considered to have failed. Significant improvement in performance was also noted when the memory was increased gradually till 1 GB. The test also failed when the device battery was 10%. In several instances, the device would shut down while in the process of generating the user report. Charging the device battery up to 20% increased performance slightly, though not much more performance boost was noted after that.

5.6 System Validation

Validation of the developed platform was done by allowing potential users to test the application and recording their feedback. This helped in evaluating if the developed platform was applicable in real user and client environments. A total of thirty users were randomly selected to conduct this testing using a questionnaire (See Appendix B). The users were

requested to download and install the MobiSec application from Google Playstore, and then test all the modules before giving their responses. The list of users included Android application developers and Android security auditors. From the responses obtained, 75% were from security auditors and 25% from Android application developers.

All the users reported that they found the MobiSec platform user interface to be user-friendly while 75% of the respondents reported that they found the platform applicable at their work. Additionally, over 75% of the respondents rated their experience using the platform to be satisfying and agreed that they would highly recommend other people to download and use the Platform.

In summary, the results from validation show that the developed MobiSec platform can be reliably used to review Android applications for exposures through exported components. For more information on the validation testing results, refer to Appendix C.

5.7 System Performance

The MobiSec platform performance was evaluated by examining its detection rate. Five applications were selected and their APK files reviewed to identify the number of exported components as shown in table 5.7.

Table 5.7: MobiSec Application Performance Validation

Results for Exported Components Detected Per Application		
Application	**Version**	**Number of Exported Components**
WhatsApp Messenger	2.21.7.15	41
Facebook Lite	248.0.0.9.116	32
Instagram	185.0.0.38.116	50
TikTok	19.2.4	39
LinkedIn Lite	3.2.2	12

The results from the test showed that the MobiSec platform had a high detection rate and can therefore be reliably used to review Android applications for exposure through exported components.

Chapter 6: Discussion of Key Results

6.1 Overview

This chapter analyses key findings from the study and from the design, development and testing of a platform to audit for vulnerable Android application components. The developed platform is reviewed to ascertain that it met the research objectives. The study results were also reviewed to examine whether they concurred with literature review and that all the research questions were answered.

6.2 Objective 1: Examining Android Application Component Vulnerabilities

The first objective of this research was aimed at examining vulnerabilities associated with Android application components. The literature revealed that Android application components that include Activities, Services, Broadcast Receivers and Content Providers are normally exported to support inter-process communications between them. An exported component has at least one intent filter or has the exported flag set to true in its declaration in the Manifest file, therefore is exposed publicly for invocation by other components from the same application or other applications.

The literature further revealed that application developers make mistakes in editing configuration files resulting in improper exportation of components which brings forth intent-based vulnerabilities that include unauthorised intent receipt and intent spoofing vulnerabilities. Researchers have noted that malicious attackers develop malware targeting applications with such vulnerabilities to tamper with data in them or exfiltrate sensitive data. This relevant information provided the basis for this study and incentive to develop a platform to assist application developers in auditing for component-based vulnerabilities in Android applications.

6.3 Objective 2: Reviewing Components Audit Techniques, Approaches and Tools

The second research objective was to review techniques, approaches and tools applied to audit vulnerabilities associated with Android application components. The literature revealed that there exist several tools to audit for Android component vulnerabilities based on static analysis and dynamic analysis techniques.

Static analysis tools identify component vulnerabilities by analysing application source code obtained from the developers or extracted from disassembled APK files. Such tools include the ManifestInspector and ComDroid. The major drawback with the use of static analysis tools as researchers have noted is that these tools are no longer effective as developers are increasingly crafting their applications' code with obfuscation and encryption schemes to protect them against reverse engineering, therefore the results obtained from these tools while examining such applications is not accurate.

Dynamic analysis tools on the other hand focus on monitoring the execution of an APK to observe their runtime behaviours corresponding to selected test cases. Such tools include Kirin and TainDroid. Literature has revealed that dynamic analysis approach alone is not comprehensive as it leaves out examination of application static features.

The gaps identified from both static and dynamic analysis can be filled by the development of a hybrid analysis tool for review of component vulnerabilities. This approach was borrowed from malware analysis research and literature review.

6.4 Objective 3: Designing, Developing and Testing Components Auditing Platform

The third research objective was to design, develop and test a platform applied to audit vulnerabilities associated with Android application components.

This objective was delivered by employing the Agile software development methodology that guided the design, development and testing of the MobiSec tool. The MobiSec tool was designed and modelled using various tools that included data flow diagrams, context diagram, activity diagram, use case diagram, sequence diagram and wireframes.

Development of the MobiSec tool was done using Android Studio and Java Object-oriented programming language. A backend API was also set up using Laravel Sanctum for regularly updating the tool's analysis rules. Before delivery and deployment, several tests were carried out on the developed platform including functional tests, compatibility tests, UI tests and performance tests. User feedback was also collected using the standard Google Playstore feedback form. This was regularly reviewed to inform future improvements to the platform.

6.5 Objective 4: Validating Effectiveness of the Developed Audit Platform

The fourth and final objective was aimed at validating the effectiveness of the developed audit platform for Android application components. Validation of the developed MobiSec platform was carried out by allowing potential users to test the platform and recording their feedback. This helped in evaluating if the developed platform was applicable in real user and client environments. A total of thirty users were randomly selected to conduct this testing using a questionnaire.

The results from validation testing showed that on average, over 75% of the users found the developed platform relevant in their work and would recommend it to others. Comparison of the developed platform's analysis results with that generated from two other frameworks showed that the developed platform had a high accuracy rate and can be used to reliably review Android applications for security exposures through exported components.

6.6 Comparison with other Tools

The MobiSec platform performance was tested against ManifestInspector and TainDroid making use of five most popular applications from Google Playstore. These applications included WhatsApp Messager, Facebook Lite, Instagram, TikTok and LinkedIn Lite. The three tools were used to examine the selected applications for exported components and results noted down as show in table 6.1.

Table 6.1: MobiSec Comparison with Other Tools

		Tool Results for Exported Components Detected		
Application	**Version**	**ManifestInspector**	**TainDroid**	**MobiSec**
WhatsApp Messenger	2.21.7.15	40	41	41
Facebook Lite	248.0.0.9.116	31	30	32
Instagram	185.0.0.38.116	50	49	50
TikTok	19.2.4	40	38	39
LinkedIn Lite	3.2.2	12	12	12

The results showed that the MobiSec tool had a higher average detection rate for Android applications exported components compared to ManifestInspector and TainDroid.

Chapter 7: Conclusions, Recommendations and Future Work

7.1 Introduction

The chapter gives a summary of the main purpose of this study which was to develop a platform to aid users and especially application developers in auditing Android applications for component-based vulnerabilities. Through conclusions, this section summarises the research objectives and how they were achieved through literature review and the design, development and testing of the MobiSec platform. This section also gives recommendations on the use of the developed platform as well as suggestions for future work that can be done to extend this study.

7.2 Conclusions

This research focused on developing a platform to audit for vulnerable Android application components. Vulnerabilities that may arise when Android application components are exported to support inter-component communications are reviewed as well as existing tools to identify them. Literature review revealed gaps in existing tools that are based on static analysis and dynamic analysis techniques, prompting the need to develop a hybrid analysis technique to compensate on these deficiencies. The process to design, developed and test a platform based on hybrid analysis is then described leading to the development of the MobiSec application. The MobiSec platform obtained an average pass rate of over 95% in the conducted unit, functional, compatibility, UI and performance tests. To ensure that the developed platform met both the functional and non-functional requirements as well as the stated objectives, validation testing was carried out where the results showed that the platform can be used reliably for auditing Android applications for component-based vulnerabilities.

7.3 Recommendations

While testing the developed platform in various user Android devices and emulators, it was noted that for optimal performance the device should have at least 1 GB of RAM, though the minimum supported RAM is 512 MB. Chromium-based web browsers such as Google Chrome, Brave, Opera, Vivaldi, Microsoft Edge and Bromite are recommended to be installed on the Android devices as they were observed to give the best results while viewing the exported HTML based analysis results.

When exporting and viewing the analysis results in PDF format, it's recommended to use any of the popular Android PDF readers such Adobe Acrobat Reader, Google PDF Viewer, Foxit PDF Reader, WPS Office and PDF Viewer. These were tested and noted to be faster in rendering the reports and did not take much CPU power. Also, it's recommended that the user be regularly clearing the exported report files from the device to free up the device memory.

The MobiSec tool developed in this study is meant for use by Android security researchers and Android application developers. Any use with malicious intent is highly discouraged and the developer does not bear any responsibility for that kind of use. For any errors encountered while making use of the tool, the user should contact the developer through the feedback form provided within the tool.

7.4 Future Work

In this study, both static and dynamic Android application features were reviewed to identify vulnerable components. Specifically, the static features reviewed included component access level and component permission restrictions while the dynamic feature reviewed was data flows within API calls. To extend the results from this study and make the developed platform more robust, future work can focus on adding more dynamic features for review which can include examining data flows in conjunction with control flows within the application through a deeper examination of API calls.

Moreover, future work can extend the hybrid analysis technique for Android application components to include a review of dynamically loaded code through DCL. Android supports DCL that allows Android application developers to load additional code at runtime. This poses a security threat especially if the code loaded alters application component configuration to allow their export for inter-component communications.

References

Ahmed, O. & Sallow, A. (2017). Android Security: A Review. *Academic Journal of Nawroz University*. 6: 135-140. doi: 10.25007/ajnu.v6n3a97

AndroBugs. (n.d.). AndroBugs Framework. Retrieved February 10, 2020 10:00 HRS from https://github.com/AndroBugs/AndroBugs_Framework

Android Developers. (n.d.). Build effective unit tests. Retrieved February 10, 2020 10:20 HRS from https://developer.android.com/training/testing/unit-testing

Bhiwani, P. & Parekh, C. (2017). Different Android Vulnerabilities. *Advances in Computational Sciences and Technology*. 10:1449-1455

Chin, E., Felt, A.P., Greenwood, K., & Wagner, D.A. (2011). Analyzing inter-application communication in Android. *Proceedings of the 9th international conference on Mobile systems, applications, and services*. June 28-July 01, Bethesda, Maryland, USA.

CVE Details. Top 50 Products By Total Number Of "Distinct" Vulnerabilities. (n.d.). Retrieved February 10, 2021 11:00 AM from https://www.cvedetails.com/top-50-products.php

Dong, S., Li, M., Diao, W., Xiangyu, L., Liu, Z., Xu, F., Chen, K., Wang, X., Zhang, K. (2018). Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. *In: Beyah R., Chang B., Li Y., Zhu S. (eds) Security and Privacy in Communication Networks. SecureComm 2018. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, vol 254. Springer, Cham. https://doi.org/10.1007/978-3-030-01701-9_10

Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A. (2010). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Communications of the ACM*. 57. 393-407. 10.1145/2494522.

Guide to app architecture. (n.d.). Retrieved February 15, 2021 15:00 HRS from https://developer.android.com/jetpack/guide

Hur, J. & Shamsi, J. (2017). A Survey on Security Issues, Vulnerabilities and Attacks in Android based Smartphone. *International Conference on Information and Communication Technologies (ICICT)*. Karachi Pakistan, 30-31 Dec 2017. Karachi: IEEE

Jha, A. K., Lee, S. & Lee, W. J. (2017). Developer Mistakes in Writing Android Manifests: An Empirical Study of Configuration Errors. *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, Buenos Aires, 2017, pp. 25-36. doi: 10.1109/MSR.2017.41

Kabakus, A., Dogru, I. (2018). An in-depth analysis of Android malware using hybrid techniques, *Digital Investigation*. Volume 24, pp 25-33, https://doi.org/10.1016/j.diin.2018.01.001

Khan, J., & Shahzad, S. (2015). Android Architecture and Related Security Risks. *Asian Journal of Technology & Management Research*. Vol 05. Issue 2. [ISSN: 2249 –0892]

Leonardo, C., Pablo, T., Lisandro, D., Germán, C. (2019). A Study of Non-functional Requirements in Apps for Mobile Devices. *In book: Cloud Computing and Big Data* (pp.125-136). DOI: 10.1007/978-3-030-27713-0_11

Malik, Y., Campos, C. & Jaafar, F. (2019). Detecting Android Security Vulnerabilities Using Machine Learning and System Calls Analysis. *IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2019, pp. 109-113, doi: 10.1109/QRS-C.2019.00033.

MobSF. (n.d.). Mobile Security Framework (MobSF). Retrieved 10 February 2020 from https://github.com/MobSF/Mobile-Security-Framework-MobSF

Mohamed, I. & Patel, D. (2015). Android vs iOS Security: A Comparative Study. *2015 12th International Conference on Information Technology - New Generations*. Las Vegas, NV, 2015, pp. 725-730

Ramírez-López, F. J., Varela-Vaca, Á. J., Ropero, J., Luque, J., & Carrasco, A. (2019). A Framework to Secure the Development and Auditing of SSL Pinning in Mobile Applications: The Case of Android Devices. *Entropy*, *21*(12), 1136. MDPI AG. Retrieved from http://dx.doi.org/10.3390/e21121136

Sharma, A., Dash, S.K. (2014). Mining API Calls and Permissions for Android Malware Detection. *In: Gritzalis D., Kiayias A., Askoxylakis I. (eds) Cryptology and Network Security. CANS 2014. Lecture Notes in Computer Science*, vol 8813. Springer, Cham. https://doi.org/10.1007/978-3-319-12280-9_13

Sharma, S., Sarkar, D., & Gupta, D. (2012). Agile processes and methodologies: A conceptual study. *International journal on computer science and Engineering*, vol. 4 no. 05, p. 892.

Six, J. (2012). *Application Security for the Android Platform*. Sebastopol, California. O'Reilly Media, Inc.

Smartsheet. (n.d.). Agile Framework in Detail. Retrieved February 15, 2021 18:00 HRS from https://www.smartsheet.com/content-center/best-practices/project-management/project-management-guide/agile-methodology

Snyder, H. (2019). Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333-339, https://doi.org/10.1016/j.jbusres.2019.07.039

Sommervile, I. (2015). *Software Engineering* (10th ed.). Edinburgh Gate: Pearson Education.

Statista. Number of apps available in leading app stores as of 1st quarter 2019. (May 6, 2019). Retrieved July 20, 2019 23:00 HRS from https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores

Thangaveloo, R., Jing, W., Chiew, K. & Abdullah, J. (2020). DATDroid: Dynamic Analysis Technique in Android Malware Detection. *International Journal on Advanced Science, Engineering and Information Technology*. 10. 536. 10.18517/ijaseit.10.2.10238.

Xiau, Y., Watson, M., (2019). Guidance on conducting a systematic literature review. *Journal of Planning and Research*, 39:93–112, https://doi.org/10.1177/0739456X17723971

Appendices

Appendix A: MobiSec Application Technical User Manual

Installation

MobiSec application is available for download from Google Playstore through the URL https://play.google.com/store/apps/details?id=com.bonafidetechnologies

Review and Feedback

After installing the application. The user can give a rating within the range of between 1 to 5 stars basing this on their experience using the application. The user is also able to write their comments as feedback to the developers of the application.

Updating

The user will be regularly prompted to update the MobiSec application through the Google Playstore application when the developer publishes a new version of the application.

Appendix B: Validation Test Questionnaire

Questionnaire Url: https://forms.gle/QJwfRJLN44purnmeA

MobiSec Application User Experience Survey

This is a survey form for user experience on the MobiSec Android application

*Required

1. I downloaded the MobiSec application on: *

2. I am a: *

Mark only one oval.

- Application Developer
- Security Analyst/Auditor
- Other:

3. The MobiSec application is easy to use and a has user friendly interface *

	1	2	3	4	5	
Strongly agree						Strongly disagree

4. I have used the MobiSec application at my work as an Android developer or an Android security analyst *

Mark only one oval.

- Yes
- No

5. I would highly recommend any other person to download and use the MobiSec application *

	1	2	3	4	5	
Strongly agree						Strongly disagree

6. Overall, my experience using the MobiSec application has been: *

	1	2	3	4	5	
Very satisfying						Very dissatisfying

7. What recommendations or suggestions would you give to help improve the MobiSec application to better meet your needs

Appendix C: Validation Test Questionnaire Results

8 responses

Accepting responses

Summary | Question | Individual

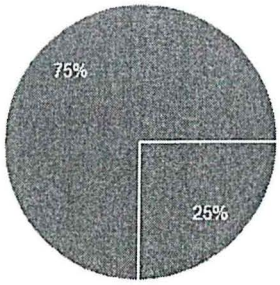
I downloaded the MobiSec application on:

8 responses

May 1991	2
Apr 2021	1 20 30
May 2021	1 2

I am a:

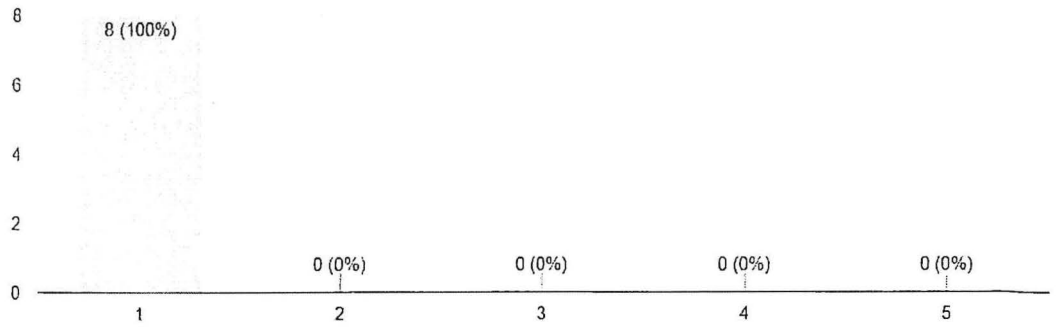
8 responses



Application Developer
Security Analyst/Auditor

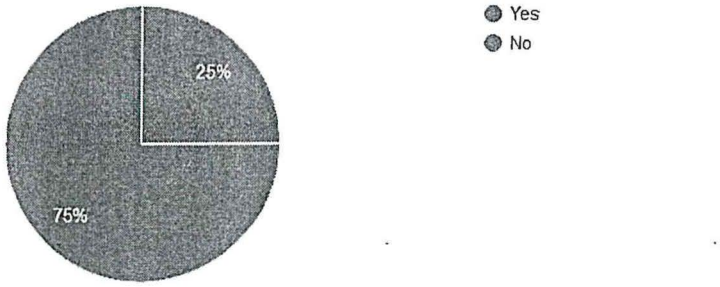
The MobiSec application is easy to use and a has user friendly interface

8 responses



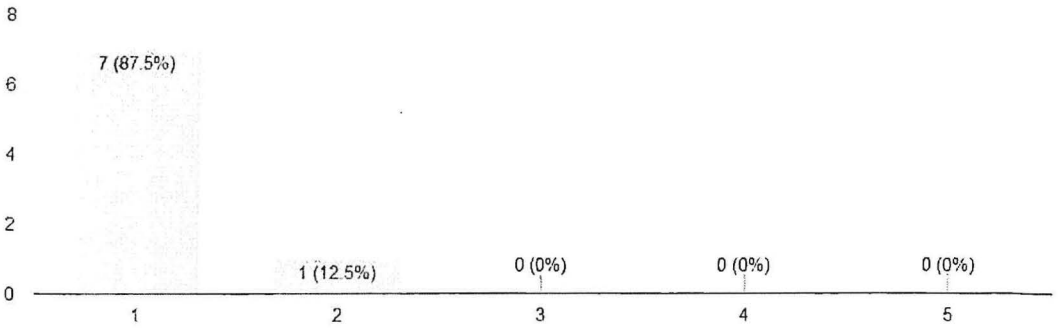
I have used the MobiSec application at my work as an Android developer or an Android security analyst

8 responses



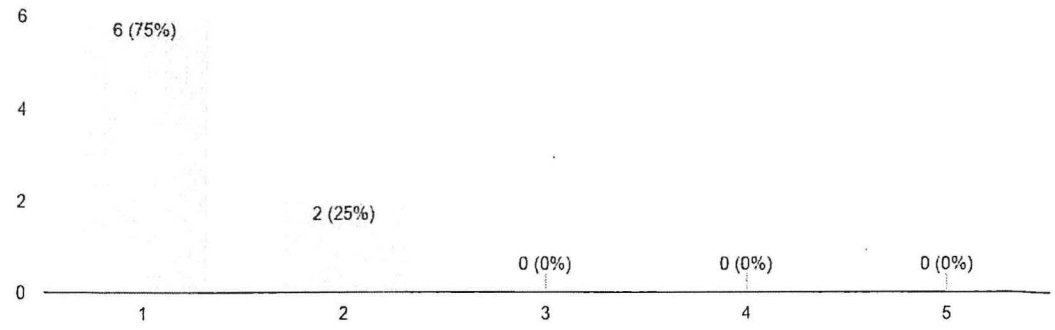
I would highly recommend any other person to download and use the MobiSec application

8 responses



Overall, my experience using the MobiSec application has been:

8 responses



What recommendations or suggestions would you give to help improve the MobiSec application to better meet your needs

3 responses

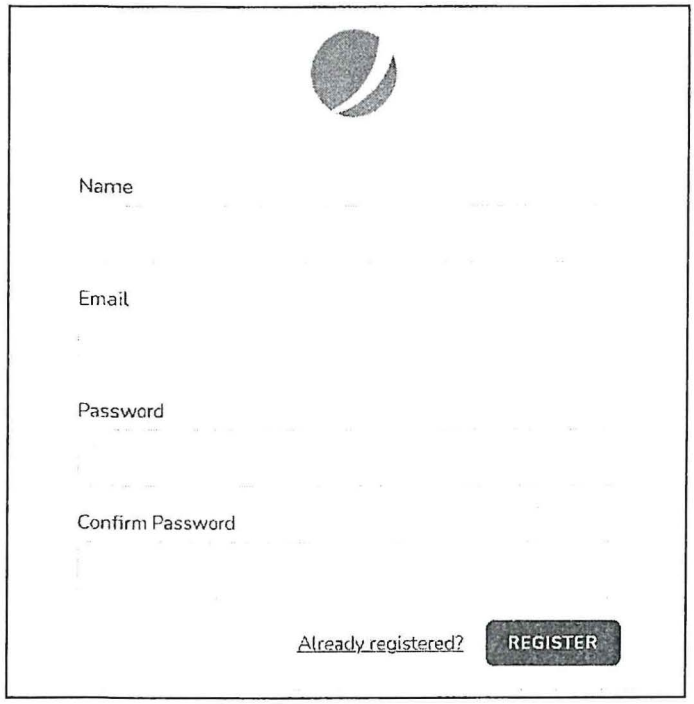
I am satisfied with the app for now

Add more features

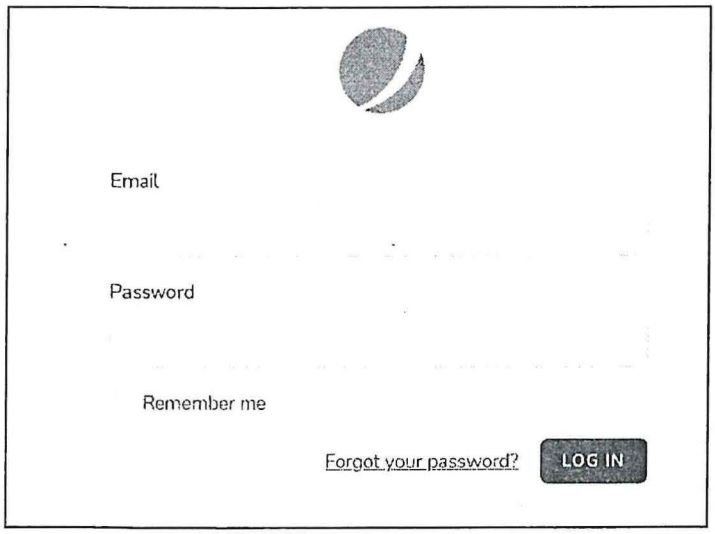
none noted so far

Appendix D: MobiSec Application Backend Web Forms

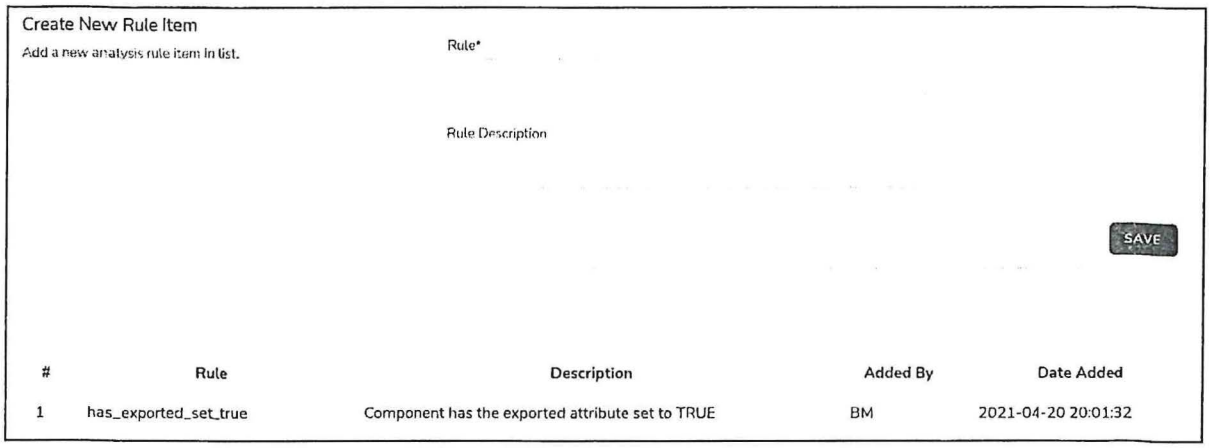
Administrator Registration:



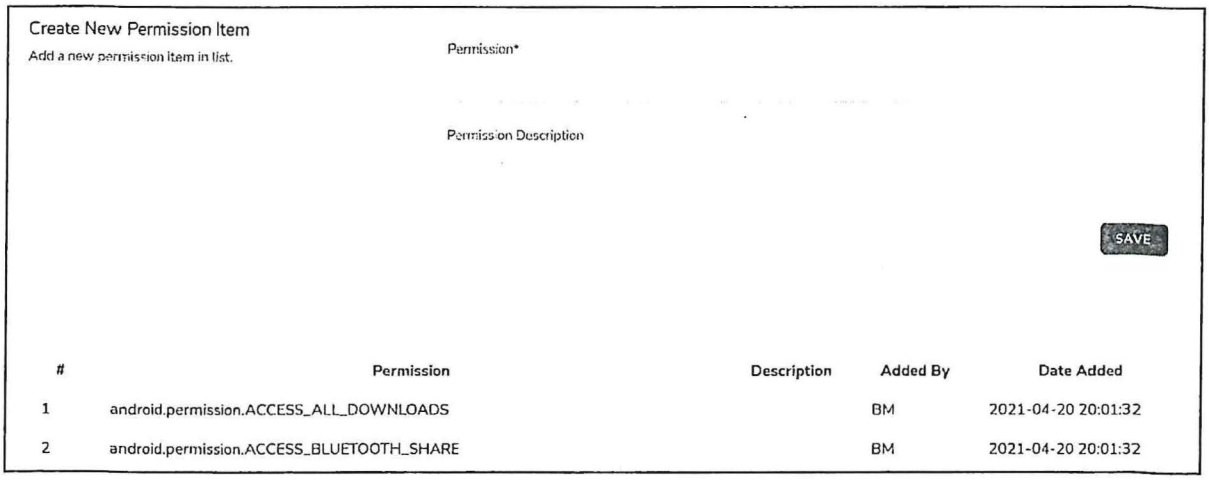
Administrator Login:



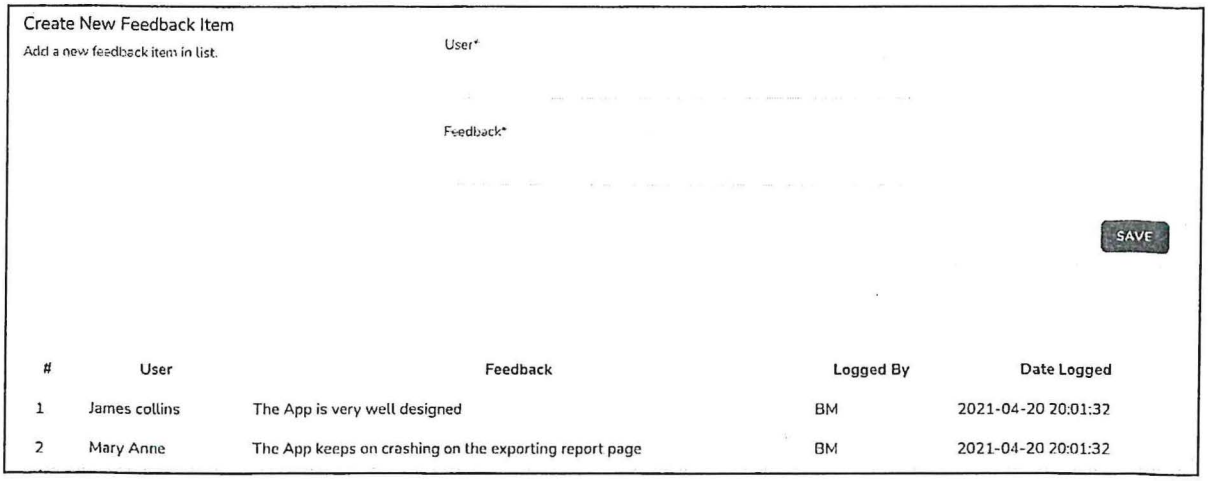
Rule Creation:



Add new Permission:



Log User Feedback:



Appendix E: Similarity Check Report

URKUND

Urkund Analysis Result

Analysed Document:	Benson_Macharia_a_platform_to_analyze_android_application_compor (D115484654)
Submitted:	10/17/2021 10:51:00 AM
Submitted By:	benson.macharia@strathmore.edu
Significance:	2 %

Appendix F: Ethical Approval: SU-IERC1182/21

Strathmore UNIVERSITY

18th October 2021

Mr Macharia Benson,
benson.macharia@strathmore.edu

Dear Mr Macharia.

RE: A Platform to Audit for Vulnerable Android Application Components

This is to inform you that SU-IERC has reviewed and **approved** your above SU- **master's** research proposal. Your application reference number is **SU-IERC1182/21.** The approval period is **18th October 2021 to 17th October 2022.**

This approval is subject to compliance with the following requirements:

i. Only approved documents including (informed consents, study instruments, MTA) will be used
ii. All changes including (amendments, deviations, and violations) are submitted for review and approval by SU-IERC.
iii. Death and life-threatening problems and serious adverse events or unexpected adverse events whether related or unrelated to the study must be reported to SU-IERC within 48 hours of notification
iv. Any changes, anticipated or otherwise that may increase the risks or affected safety or welfare of study participants and others or affect the integrity of the research must be reported to SU-IERC within 48 hours
v. Clearance for export of biological specimens must be obtained from relevant institutions.
vi. Submission of a request for renewal of approval at least 60 days prior to expiry of the approval period. Attach a comprehensive progress report to support the renewal.
vii. Submission of an executive summary report within 90 days upon completion of the study to SU-IERC.

Prior to commencing your study, you will be expected to obtain a research license from National Commission for Science, Technology and Innovation (NACOSTI) https://research portal.nacosti.go.ke/ and also obtain other clearances needed.

Yours sincerely,

for: Prof Fred Were,
Chairperson; SU-IERC

18 Oct 2021

Ole Sangale Rd, Madaraka Estate. PO Box 59857-00200, Nairobi, Kenya. Tel +254 (0)703 034000
Email admissions@strathmore.edu www.strathmore.edu