National College of Ireland

Protecting the integrity of Android applications by employing automated self-introspection methods

MSc Internship
Cyber Security

Swapnil Jadhav
Student ID: 18212344

School of Computing
National College of Ireland

Supervisor: Ross Spelman

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name: Swapnil Ganesh Jadhav….

Student ID: 18212344……..

Programme: MSC CYB…… **Year:** 2019-2020……..

Module: Academic Internship…..

Supervisor: Ross Spelman ……

Submission Due Date: 17/08/2020……

Project Title: Protecting the integrity of android applications by employing automated self-introspection methods……

Word Count: 7250…… **Page Count**…19……..

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.
I agree to an electronic copy of my thesis being made publicly available on NORMA the National College of Ireland's Institutional Repository for consultation.

Signature: Swapnil Ganesh Jadhav……

Date: 17/08/2020……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Protecting the integrity of android applications by employing automated self-introspection methods

Swapnil Jadhav
18212344

Abstract

The Android ecosystem gained a huge popularity and market share owing to wider compatibility and its open-source nature. But it also suffers from circulation of counterfeit applications resulting from repackaging attacks. Such attacks are aided by easy reverse engineering and the application's poor self-defence mechanisms. It results in an adversary being able to modify the application to introduce additional functionalities like embedding a spyware, malware, ransomwares and other malicious codes. To defend against the repackaging attacks the android applications are protected with various obfuscation, anti-debugging, anti-reversing and anti-tampering schemes. However, the protection mechanism solely lies at the client side and are susceptible to the reverse engineering thereby rendering them ineffective. This paper proposes a new mechanism to detect the tampered apps and prevent it from communicating with the application server. The mechanism aims at separating the client-side tamper-detection logic from the main application and placing it in the android's system partition. The other part of verification logic will reside at the application server and co-ordinate with the client-side's logic via a cryptographic token. The proposed mechanism first detects the tampered application and later cuts off its communication with the application server thereby rendering it non-operational.

1 Introduction

Android-based smartphones have become an integral part of our daily lives. The Android offers a platform for developing a wide range of applications to be used in various sectors like health, finance, banking, education, gaming, etc. These applications are the most-likely vulnerable assets for any organisation from a security perspective since they open-up a doorway into the organisation's infrastructure. Hence it is pre-emptive that these applications run in an un-modified way in order to provide the intended functionalities. Results of the tampering may include intellectual property theft, data-loss, business and financial loss, reputational damage, etc. As per the OWASP Mobile Security Project[1] application code tampering has been among the top 10 threats for mobile security for an organisation.

The android applications can be broken down and analysed up to a granular level with techniques such as run-time modifications, reverse-engineering, re-patching, etc. Various Dynamic Binary Instrumentation (DBI) tools such as FRIDA, Intel's Pin, Xposed Framework, etc. allow modifying the application's binary instructions to be modified during runtime. For these tools to work either the underlying platform must be operated with

1 https://owasp.org/www-project-mobile-top-10/

elevated privileges or the tools must be bundled with the targeted application. FRIDA[2] in non-rooted mode works by using its FRIDA-gadget which is to be patched alongside the target application. Once the application is launched the FRIDA-gadget library is loaded in the process memory and builds a two-way communication path with the FRIDA-agent already-running on another device.

This report's scope is limited to defend against the code-tampering attempts on a non-rooted android device. Various android application anti-tamper mechanisms exist which slow down an adversary in their tracks. Popular protection measures include –

1) Calculation of CRC checksum over the "classes.dex" file and matching it with the stored values either on client or server side.
2) Obfuscating the application logic.
3) Encrypting the main application logic and then decrypting it at runtime.
4) Running tests that pre-check the underlying platform conditions before launching the main application.

All these methods suffer from a common single point of failure which is the placement of the verification logic alongside the main application. Before installing an application, the integrity checker code can be removed then, the application can be re-patched and resigned. After this, the now modified application will be treated as a third-party app by an android device. This new app can be installed on an android device since installation of third-party apps is allowed by android. And thus, counterfeit copies of the original application can be distributed amongst the non-suspecting users. Hence, it is necessary that the sensitive additional code must run outside the reach of a regular user.

In android certain apps that are pre-installed cannot be removed by a regular user. This is because these apps are marked as 'system' and the android O.S prevents these from being removed. So, can this feature of the android O.S be leveraged to protect the application's integrity? Hence, the objective of this report is to research the possibility of placing the integrity checker code in an environment that's part of the core system which can assess the integrity of the user application code. By doing so, any unauthorised modifications to the user app can be detected and the counterfeit app can be rendered unusable.

This research will help developers to build robust app protection mechanisms. It will ensure that only the approved builds releases would be allowed to run on an end user's non-rooted android device. The report is divided into sections as follows – 1) Introduction – Introduce the topic and briefly outline the various anti-tamper mechanisms in use. 2) Related Work – A brief discussion of the research work done by numerous researchers for protecting android applications. 3) Research Methodology – The research procedure and evaluation strategy. 4) Design Specification – The technical aspects of the solution covering the platform, tools, frameworks and other requirements. 5) Implementation – Description of the solution and the results produced. 6) Evaluation – Comprehensive analysis of the tests performed 7) Conclusion and Future Work – Research conclusion and possible future enhancements.

2 https://frida.re/docs/modes/

2 Related Work

This section showcases the prior research done by notable authors for protecting android app's integrity. An overview and comparison of numerous mechanisms have been discussed like dynamic code loading, stub-DEX, dynamic code injection, APK-overwrite, custom frameworks for APK integrity verification. It concludes with the emphasis of using the system partition of the android platform for providing reliable placeholder for the additional code.

2.1 In-app anti-tamper measures

This subsection discusses various tamper detection techniques that are included as a part of the main application logic. Also, the shortcomings of using them have been discussed in brief.

To protect an application from malicious attacks anti-reversing techniques are deployed. A research by Kundu Deepti [1] lists various methods such as inserting dead or irrelevant code, extended loop conditions and adding redundant operands. These techniques are just to add complexity to the application code during reversing but are immune to an experienced reverse engineer. Instead code obfuscation can be used to change the visual layout of the application code. Code Obfuscation is done to mask the original code so that an adversary is confused during reverse engineering of the application logic. Code obfuscators convert the application packages' original identifiers, classes and methods into an obscure representation. This slows down the application tampering attempts as it makes no literal sense of the application code upon reversing. These tools automatically locate and remove dead code too. DexGuard[3], Proguard[4], DexProtector[5], etc. are among the well-known Obfuscator tools used. A research on android code obfuscation was performed by Patrick Shulz [2] wherein various methods such as identifier mangling, string obfuscation, dynamic code loading and self-modifying code were discussed. As intended, only the visual representation of the reversed code was altered whereas the application control flow was unchanged. Hence, this basic technique doesn't provide much value to secure the application code. Sudipta Ghosh et al. [3] took this a step further and proposed an algorithm that increased the complexity of the application's workflow. The algorithm and the basic obfuscation technique now not only change the visual appearance of the code but also complicates the logic. But this can be bypassed with the help of tools such as DeGuard[6], java-de-obfuscator[7] and with some experience in reverse engineering. Another research by Yan Wang et al. [4] describes the techniques to identify the obfuscators used on an android application. Benjamin et al. [5] in their research work have shown the process to repackage an Android application package

[3] https://www.guardsquare.com/en/products/dexguard
[4] https://github.com/Guardsquare/proguard
[5] https://dexprotector.com
[6] http://apk-deguard.com/
[7] https://github.com/java-deobfuscator/deobfuscator

(APK) and removing the application's security logic. The de-obfuscated application code can now be re-packaged with FRIDA gadget to modify the application's runtime behaviour. In order to build upon the shortcomings of the previous technique other measures such as anti-debug and CRC checks were introduced and they are discussed as below.

Applications can be debugged at run-time to inspect the internal operations such as return values of critical methods and modify them. A research by Michael N. et al. [6] describes various methods to prevent reverse engineering by presenting techniques that attack the debuggers. A mechanism against debugging was proposed by Junfeng Xu et al. [7] wherein the debug state and environment can be pre-checked to protect the apps against dynamic debugging. The detection for 'Debug State' involves checking the usage of various debuggers such as GDB[8], IDA[9], 'strace[10]' by enquiring the parent process or using the Linux's inbuilt utility 'ptrace' to read the process status. 'Debug Environment' detection involves checking for any attributes that are specific to emulators. However, the techniques presented by Michael N. et al. [6] and Junfeng Xu et al. [7] can be bypassed as the 'debugger' feature can be enabled explicitly under the Android's developer options as highlighted by numerous security researchers in the Open Web Application Security Project's (OWASP) Mobile Security Testing Guide[11] (MSTG). Also, a research by A. Vasudevan [8] provides a technique that allows an application tester to set virtually unlimited breakpoints that cannot be detected. They used the concept of 'Stealth Breakpoints' which combines the simple stealth methods using hardware single-stepping and virtual memory mechanisms. Having enabled the feature, the app can now be launched in a debugged state and then the anti-debug mechanism can be safely disabled. In addition, the anti-debug mechanism can also be bypassed by re-packaging the app with the FRIDA gadget and making runtime modifications.

According to the OWASP's MSTG a CRC check also known as an APK integrity check over some files provides a basic anti-tamper mechanism. This technique can be circumvented by using the FRIDA gadget as the main application is executed alongside the CRC checks. The gadget enables an adversary to skip the CRC checks. To separate the check logic from the main app researcher Kyeonghwan Lim et al. [9] proposed a STUB-DEX method. In this method an initial pre-run environment check is performed by the application loader i.e the STUB and then if passed, the main app's DEX (Dalvik Executable) can be loaded in the memory. This approach prevents the main application logic from being executed unless the environment checks are fulfilled. However, this STUB-DEX is shipped with the application DEX to the client-side in an unencrypted format and this risks a malicious unauthorized tampering. Byungha Choi et al. [10] addressed this previous limitation in their research work and proposed an improved technique in which the application's DEX is decrypted at runtime and deleting this DEX after usage. But this technique too can be bypassed as the STUB-DEX can be modified to drop the pre-run checks by reversing the app's APK and not delete the

[8] https://www.gnu.org/software/gdb/download/
[9] https://www.hex-rays.com/products/ida/support/download_freeware/
[10] https://strace.io/
[11] https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide

decrypted main app DEX, as per the method introduced by Benjamin et al. [5] The STUB can be modified at runtime too by repacking the APK with the FRIDA gadget.

Hence, as seen from the above approaches these in-app anti-tamper mechanisms are built as a part of the main application. This shortcoming makes it easy to bypass them as the APK can be reversed to remove the mechanisms, repacked and re-signed to produce a version that has no defence mechanisms. APKs can be re-packed which is shown in the research work by Benjamin et al. [5] as mentioned earlier. Thus, the verification logic must reside outside the main application which will make it difficult to be accessible for any tampering attempts.
In the upcoming subsection tamper detection measures, which reside outside the main application APK are discussed.

2.2 Out of app anti-tamper measures

This subsection discusses various anti-tamper detection techniques which are in the form of an ad-hoc code which resides either at server or client-end. It further briefs on the shortcomings of each technique and emphasises on the need of a tightly coupled out of app anti-tamper measure.

In the previous subsection we saw how the client side APK based obfuscation tends to stall the reversing of the application and limitation of this technique. Yuxue Piao et al. [11] in their research proposed a server-based obfuscation module along with a client-side minimal obfuscated routine. Upon invocation the client app requests the server to send the encrypted and obfuscated main routine. The server sends in the requested routine along with the secret key to decrypt it. The client decrypts the received blob and executes the anti-tamper checks first, if successful then the main application logic is executed. This technique falls under ‘code offloading’ category wherein the application code resides on the server and is delivered to client at runtime. A research by Marco V. et al. [12] compared and analysed the pros and cons of code offloading. Certainly, in case of low bandwidth and high latency network connections code offloading cannot be used. Hence the defence mechanism itself will affect the application’s usability. This method addresses the client-side application reversing but fails to check the integrity of the client-side routine that makes the initial request. This makes it possible to tamper the client-side routine by re-packaging the application as per research by Benjamin et al. [5]. The FRIDA gadget can be included in the re-packaging process and the application’s functionalities can be modified at run-time. A similar exploitation was carried out by researchers Taehun Kim et al. [13] where they highlighted the inherit weakness of the APK based anti-tamper mechanisms and emphasized the need of a secure approach which is platform based.

Researchers Haehyun Cho et al. [14] proposed a dynamic code injection technique which will serve as a defence against application repackaging. They moved the tamper detection logic on the server side and injected it at client side where two other modules called the precursor and attestor will operate on it. The main application’s code instruction sequence and signature stays on the server. Upon executing the client-side module, the precursor connects

to the server and sends the application's information to the server. The server replies with the saved instruction sequence and the application signature. The attestor module on client-side will now match the received information with the one stored on the user device. The attestor will terminate the application execution if the comparison fails. The server-end logic isn't verifying the state of the precursor or attestor, hence both of these can be tampered. Also, this manoeuvre introduces a performance overhead as the attestor module is not coded in native language on android. This issue was addressed by Lina Song et al. [15] in their research work where they introduced an inter-locking dynamic dispatching guard net of time diversity. This net-like structure is generated from a defence static net which invokes the more efficient C/C++ codes. It too uses a client-server code attestation framework. However, like the previous solution by Haehyun Cho et al. [14] this technique too can be bypassed as no client-side integrity checks are present which will verify the state of the attestation logic and FRIDA gadget can be used for re-packaging the application and modifying its runtime behaviour.

A tamper detection scheme proposed by Jiwoong Bang et al. [16] segregated the signatures within an android application. Their custom framework known as the 'APK attester' acquires the developer's public key from the application's APK and sends it to the server for verification. The server then compares the received key with the one it has stored earlier. Successful operation will allow the application to execute, failure will terminate it otherwise. However, this technique has two flaws – one, that the application's digest value is not checked for integrity, - second, there is no provision for a backup plan in case the application fails to send the developer's public key. Hence, as mentioned before the application can be tampered to not send the developer's key to the server and will continue to function normally. A framework known as Stochas-tic Stealthy Network (SSN) was introduced by Lannan Luo et al. [17] in their research work which provides a repackage-proofing solution. The SSN operates by injecting the target application with multiple obfuscated detection nodes. These nodes will verify the integrity of the application by acquiring the secret key embedded and the application signature, then comparing it with the ones stored on the server. App terminates if the values differ. For this method to work it is essential that the end user should have this framework installed on their device, this may not be feasible for every user out in the market. Hence, for the users who don't have this framework on their devices the target application is left unprotected.

Hence, the tamper detection schemes built in the application are not enough to securely verify the integrity of the application. To overcome the shortcomings of these in-app integrity verification mechanisms numerous out of band techniques were designed that took the verification logic outside the main application. Although these methods did introduce some complexity in the design and made it difficult for an adversary to reverse the application, they had their own shortcomings. As long as an adversary has access to the verification logic, they can tamper the application to bypass the integrity checks. Also, just relying on the client-side application logic to do integrity checks is insufficient. Both the server and the client have to be involved in the tamper detection procedure. The android's 'system' partition can be used to park the application verification logic. This area of android is not accessible to the end user

in a non-rooted mode and can hence stays protected from the adversary's reach. Also, since it's a system app it can be pushed as a part of the android's OTA update after collaborating with the Original Equipment Manufacturer (OEM).

3 Research Methodology

As discussed in the previous sections the anti-tamper mechanisms that are built in the android application are not guaranteed enough to provide satisfactory outcomes. Also, the mechanisms that shift the tamper detection logic outside the application APK too have their own shortcomings. This is owing to the failure to verify the integrity of the client-side module running on the end user device. As a result, an application can be re-packaged with additional code or modified code and re-installed on a device thereby bypassing all the anti-tamper checks. This calls for placing the verification logic in such a location from where it can fulfil its function without having to worry about the tampering attempts.

The '/system' partition in an Android ecosystem is an area which holds the main operating system. It also includes the pre-installed applications like clock, gallery, calendar, etc. and other OEM applications. The system partition is a read-only partition and cannot be modified when the android is running. To modify it one must have a super user privileges otherwise known as 'root' and then remount it. Hence our verification logic can be hosted in this partition as a 'system app'. We cannot tamper any application that's available in the market as it would trigger legal proceedings as no prior approval from the application's rightful owner. Hence, for this project we will develop a very basic 2-tier android application in Java using Android Studio. Then we will try to tamper this user application using various techniques and evaluate if our solution is successfully in detecting the application integrity breach and take necessary actions. For this research work the solution would be deployed locally and an android emulator would be used to run the user and system apps.

The solution will be evaluated against the metric – Accuracy, i.e. how accurately can the verification logic detect the tampering attempts and prevents any use of the modified app.
This solution uses two applications – 1) a user application which will have the business logic 2) System app that will contain one part of the verification logic that will verify the integrity of the user app. The other part of the verification logic will reside on the back-end application server.

The methodology followed for this research comprises of placing the verification logic as a system application, then extraction of the user application signature and its secure transmission to the application server. The user application will receive a token from the server if it passes the integrity checks. The steps that will be followed in the research work are outlined as below-

1) The user app upon invocation will communicate with the system app via process and runtime Application Programming interfaces (API)s.

2) The system app will calculate an MD5 signature of the user app and sign it with a certificate with the jarsigner [12]tool.
3) The signing certificate will have the developers public key and the signed signature blob would be sent over a TLS backed secure channel to the backend application server. It is to be noted that the backend application server's public certificate would be used for certificate pinning in both the system and user applications.
4) The backend application server will now decrypt the received signature blob with the corresponding developer's private key. Then it will compare the signature with the one stored in its database.
5) If both the signatures values match then the server will issue a short-lived cryptographic token, AES-256 encrypt it with the server's private key. This token is sent back to the system app. The token is generated using a secure random number generator algorithm, stored in the database
6) The system app will verify the token's authenticity by AES-256 decrypting it with the server's public key. If verified, then the token will be passed on to the user app.
7) The user app will now include this token with its every network request to the backend application server. The server will verify the token with the stored ones before serving the user app's request.
8) Since this token is short lived the token will expire after sometime and the user app will keep requesting the new token at certain intervals throughout the application's session.
9) Absence of this token in the user application's request will generate a missing token error. The user app will once again request the system app to initiate the token generation process.
10) If the signature comparison fails on the server side then no token will be issued to the user app. And since the user app does not have the token the server won't serve the resources requested by the user app.
11) The signature comparison failure will indicate that the client-side user application is compromised or tampered and no resources will be served unless the correct signature value is passed to the server.
12) As mentioned before the system application will reside in the '/system' partition its code is inaccessible to the end user.

As compared to the research work by other authors mentioned in the 'related work' section this solution will accurately detect the tampered user app. Hence, tampering or using the counterfeit application will render it useless thus preventing its misuse. This solution ensures that the backend application server servers the requested resources for the user app only if it received the correct server issued token. The next section describes the design specification for the thesis implementation.

[12] https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jarsigner.html

4 Design Specification

Application development is carried out using Android Studio[13] Integrated Development Environment (IDE) for both the user and system app. The required software development kit (SDK) is installed by the Android Studio itself. The Java Development Kit[14] (JDK) path on the local system must be supplied to the Android Studio. The IDE is installed on Windows 10 Pro base operating system.

The user and system applications are developed in Java. Android Emulator is pre-installed in the Android Studio and an android virtual device (avd) is created from the same. The avd is used for deploying the system and user app. The server-side logic is developed in python along with the supporting flask framework. The server-side application logic is hosted on Pythonanywhere.com with MySQL database.

The User app is installed in the user partition '/data/app'. The system app is installed in '/system' partition. The server comprises logic for the token verification, token generation, signature verification, database for app signatures and the business. The architecture diagram of the solution is presented in Figure 1.

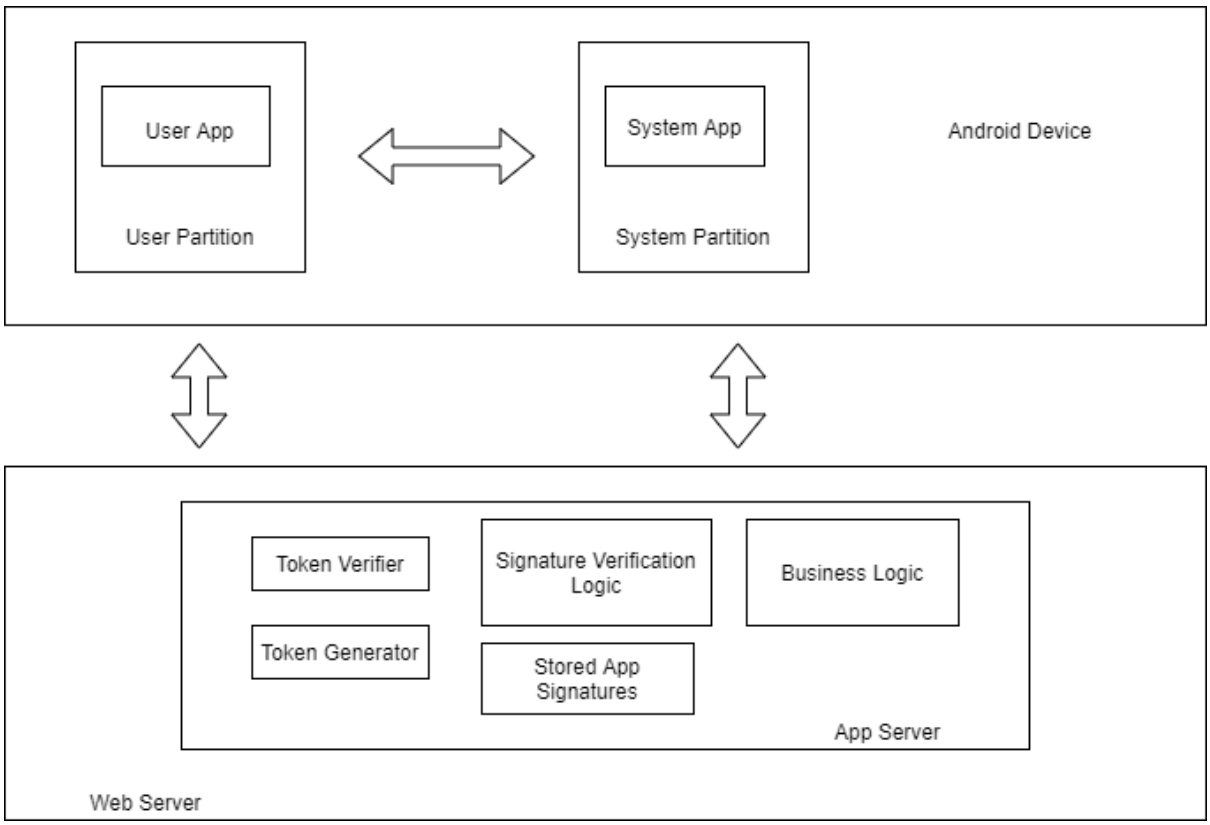

Figure 1: Architecture block diagram of the solution

[13] https://developer.android.com/studio/index.html
[14] https://www.oracle.com/ie/java/technologies/javase-downloads.html

The bi-directional arrows represent a two-way communication channel. The user app will communicate with the system over the android's system and process APIs. For creating the application's signature, the MD5 algorithm is used. The system app will use MD5 algorithm to sign the user application's signature. The application server's public certificate will be used for certificate pinning in both the user and system apps. The server uses hashlib.md5 function for comparing the received and stored application signatures. The server uses secure random number generator algorithm for generating the tokens. The tokens are short lived and must be renewed throughout the application's user session.

The functionality of the model implemented is as follows –

1) The verification logic is kept separate as a system app in the system partition.
2) The user app contains the business logic.
3) When launched, the user app will communicate with the system app.
4) The system app will generate the user app's signature and send it to the application server.
5) The application server will compare the received signature with the stored value.
6) If both of them match then a token would be dispatched to the system app, which in turn will pass on this token to the user app.
7) The user app must include the token in each of its server requests, failure to do so would receive a missing token error.
8) If the signature match fails at the server then no token would be issued and the user app cease to execute for that session.
9) The token generated by the server is short-lived, meaning it has to be renewed by the user app for the duration of the application's user session.
10) The server always checks the received token for its authenticity and validity before serving the user app's request.

5 Implementation

The final build of the solution was developed using Android Studio IDE. It was implemented and tested against various test cases which are mentioned in the next section. The solution was tested on two user apps – Userapp1 and Userapp2. Userapp1 is an untampered user application whereas Userapp2 is a user application which is tampered using reverse engineering. The Userapp1, Userapp2 and the system app were installed on an android virtual device created from the android emulator in the Android studio IDE.

The runtime behaviour of the Userapp1, Userapp2 and system app is as follows -

1) Upon launching Userapp1 the system app's verification logic was triggered and the Userapp1's signature was sent to the backend application server over a secure https channel.
2) The server compared the received signature with the value stored in the database.

3) A token was issued to the system app which relayed it over to the Userapp1 application.
4) Userapp1 included this token in it's every request to the application server and was able to communicate with the server hassle-free.
5) The token did expire in between the application's session but every time a new token was issued by the application server to the user app.
6) The Userapp1's traffic could not be intercepted using BurpSuite[15] as we had implemented certificate pinning of the application server on the user app.
7) Userapp1's activity was identified as the normal application's behaviour since it was not tampered and was accepted by the application server.
8) Userapp2 upon invocation communicated with the system app.
9) The system app generated Userapp2's signature and sent it to the application server.
10) The application server compared this received value with the Userapp1's signature as Userapp1 is the production approved genuine application.
11) The comparison failed as the signatures of both the applications differ.
12) The application server replied back to the system app about the signature mismatch and did not generate the required token.
13) The system app upon receiving the server's response terminated the Userapp2.
14) Userapp2 was modified again to not interact with the system app but connect directly to the application server.
15) Since the application server expects a valid token in every user app's request, it replied back with a missing token error message.
16) Having no way of obtaining the server's token without providing valid signature the Userapp2 was rendered functionless.
17) The system app in the system partition was not accessible for reverse engineering on the emulator as it required elevated privileges.

The application server uses the python's hashlib.md5 method for comparing the received user application's signature to the ones stored in its database. Secure random number generator algorithm was used to generate the token. The server validated the token (if any) received in the user application's request every time before serving the requested resources.

The entire runtime for both the user and system applications is summarized in a flow diagram presented in Figure 2.

[15] https://portswigger.net/burp/communitydownload

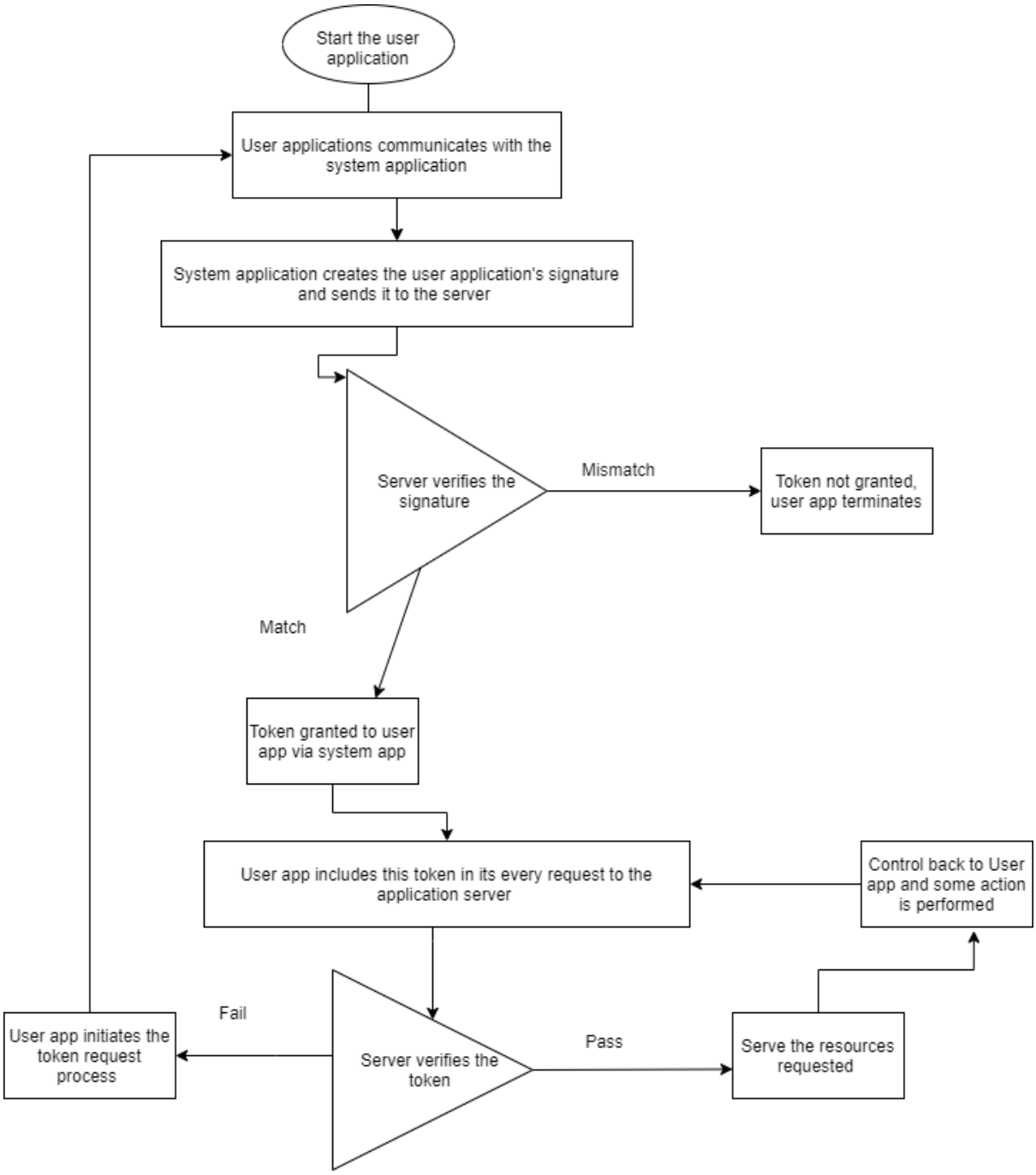

Figure 2: Code Flow diagram of the solution

The next section describes the test cases performed and the received / observed output.

6 Evaluation

This section showcases the various test cases performed, the results achieved, comprehensive analysis of the methods used, shortcomings of the solution and the possibilities of enhancing the solution.

6.1 Test Cases Performed and Results

The test cases performed and the outcome have been presented in a tabular format in the Table 1.

Table 1: Test Cases and Results

No.	Test Case	Activity Performed	Result
1	Tampered user app with additional code.	The user app was reverse engineered, modified to add new code / feature and executed.	Pass
2	Tampered user app with no connection to the system app.	The user app was reverse engineered, the communication with system app was removed and executed.	Pass
3	Tampered user app with Frida Gadget.	The app was repackaged with the Frida gadget and executed.	Pass
4	User App traffic interception.	Attempted to intercept the user app's traffic with the server.	Pass
5	Token Replay.	Tried using the old token for the server requests.	Pass
6	Token less server requests.	User app tried to access the server resources without the token.	Pass
7	Token Guessing.	User app tried to access the server resources by guessing the token.	Pass
8	Modifying the system app.	The system app's verification logic was targeted and attempted to modify the system app.	Pass
9	Run tampered user app without installing system app.	The system app was uninstalled and the tampered app was executed.	Pass
10	Run original user app without installing the system app.	The system app was uninstalled and the unmodified user app was executed.	Pass
11	Tampered user application execution in rooted environment.	The user app was repackaged with Frida gadget and executed on a rooted emulator environment.	Fail

The Pass / Fail outcome in the 'Results' Column is described as below –

Pass – The solution successfully detected the tampered app and terminated communication with it for test cases 1,3.

- For test case 2, since no system app is present to initiate the token generation process the user app did not have the required valid token. Hence, it couldn't access the server's resources due to absence of a valid token.
- For test case 4, the app's traffic couldn't be intercepted because of the certificate pinning in place. Hence the communication channel wasn't compromised.
- For test case 5, the application server rejected the replayed token because an expired token was used and the user app's request was denied.
- For test case 6, the application server didn't serve the user app's request since it did not have the required valid token.
- For test case 7, user app's request contained invalid tokens in an attempt to guess the valid token. The application server rejected the tokens and denied the user app's requests.
- For test case 8, the system app's logic was tried to tamper but since the /system partition is read-only the attempt was unsuccessful.
- For test case 9, 10 since no system app was present the token request was not initiated and the user app's request had no tokens. Hence, the app server denied these requests.

Fail – In a rooted environment the application which was tampered with the Frida gadget was able to hook into the system app's process memory and modify the application signature that was to be sent to the server. The unmodified application's signature was passed instead of the tampered application's signature to receive the valid token.

6.2 Discussion

The user application was tampered with many techniques aided by reverse engineering and then repackaged. The repackaged user application was executed on both non-rooted and rooted emulator environments. Numerous test cases were prepared and tests were carried out. The outcomes of each test case are discussed as below.

In the first test case the legitimate user application was reversed and some additional features were added. This is an example of malicious actors repackage the legitimate innocuous application with malwares, spywares, etc. The goal was to tamper the application and test if the solution detected the tampered app. As expected, the solution correctly identified and stopped the execution of the tampered application since its signature was different from that of the legitimate application. This was not possible in the case of the previous researchers' work as the verification logic was built in the main application which could be removed via reverse-engineering. The verification logic in the system app was shielded from tampering and hence the solution worked correctly.

In the second test case, the legitimate user application was modified so as to not interact with the system application. The modified user application was executed, since there was no communication with the system application no signature was sent to the application server. Hence, no token was generated for the user application and the requests to the server did not

have the required valid token. Hence, the server did not serve the user application's requests. This was not possible in the previous researchers' work as there was no provision for tokenized communication and hence the application server continued to serve requests from the tampered application.

In the third test case, the legitimate user application was only repackaged with Frida gadget and no code was modified. But still this classifies as a modified application and as a result the generated application signature differed from that of the legitimate one. Owing to this difference the server didn't issue a token for the modified application. Again, due to the lack of the tokenized communication the previous solutions failed to identify the requests from a tampered/modified user application.

In the fourth test case, two versions of the user application were used. The unmodified user application was executed first. The aim of this test was to sniff the token from the user app – server communication and use it for the requests from the modified app. But due to the certificate pinning in place the application traffic could not be intercepted and the token for the modified application couldn't be obtained.

In the fifth test case, the unmodified user application tried to use the old expired token in its requests to the server. The server correctly verified the old token usage and generated a token expiry error. Hence the user application was unable to access any resource from the server unless it passed on a new valid token.

In the sixth test case, the unmodified user tried to communicate to the serve without any token. This was done after the application passed the signature check. The application was coded to not use the token received from the system application. As its evident, due to absence of token in the legitimate user application's server requests the server refused to provide access to its resources. Valid tokens are required by the server even if the requests are coming from the legitimate user applications.

In the seventh test case, the user application was designed to not use the server assigned token. Instead, a random token was to be used with the user application's request. The server logic correctly identified the invalid token usage in the user application's request and declined all of them. Hence, as with previous test case a correct valid token is to be supplied by the user application if the server's resources are to be accessed.

In the eighth test case, the system app was targeted. Tried to install a new system application with modified verification logic that always passed the unmodified application's signature. Installing applications as 'system applications' require elevated privileges which was not possible on a non-rooted environment. Hence this test passed, but it would have failed on a rooted emulator environment.

In the ninth test case, the system application was removed and the tampered user application was executed. Since there was no system application present to initiate the token generation process the tampered application didn't have a token to interact with the server. Hence tokenization of the application communication ensures the server doesn't interacts with the counterfeit application which was not designed in the previous researchers' solution.

In the tenth test case, the system application was removed and the unmodified user application was executed. Since there is no system application, no token was generated for

the legitimate user application. Like the previous test case the server declined all the requests from the user application as no token was present.

In the eleventh test case, the user application was repackaged with the Frida gadget and executed in the rooted emulator environment. The Frida gadget was able to hook in to the system application's process memory and modify the application's signature. Instead of passing the tampered application's signature, the unmodified application's signature was passed to the server. The server-side's verification logic successfully verified the signature check and issues the token to the modified user application. This test case failed as in a rooted environment the FRIDA gadget is able to hook into almost all the system level APIs. This is a limitation of this solution.

As evident from the test cases in a non-rooted environment, the solution was able to detect and prevent the tampered user application from accessing the application server's resources. But the solution fails when it is deployed and executed in a rooted environment. Also, this solution must be installed as a 'system application' in the end user device. Hence, the business must work with the OEMs to include their solution with the respective android updates for their respective android devices. Assuming the OEMs have pre-installed this solution for the end users, the target application will communicate with the system application. Since a normal user's android device won't have the super user or root privileges unless the user explicitly chooses to do so, this solution works perfectly fine with a 100% accuracy. Hence mass distribution of counterfeit applications can be prevented as the system application that would be pre-installed in the users' devices and will detect and terminate the modified apps. But the application won't be protected against the reverse engineering as on a rooted environment the tampering possibilities are infinite. In order to improve / enhance the application's anti-reversing capabilities a defence-in-depth approach should be used. Other techniques such as code obfuscation, dynamic code offloading loading, code injection, etc. must be deployed in addition to this solution as per the business's requirements. A more promising security enhancement would be the use of a secure / trusted execution environment (TEE). The TEE is a separate execution environment that runs in parallel with the android operating system. The TEE resources are not accessible from the normal android O.S and hence is the ideal location for placing the entire business application or just the verification logic. Usage of the TEE is an advanced concept and can drastically improve the security of an android application.

7 Conclusion and Future Work

Existing solutions such as obfuscation, dead code injection, CRC checks, anti-debugging, etc. discussed previously in the related work section do introduce complexity in reversing and modifying the application. But these trivial application protection schemes lack the separation and verification of the anti-tamper mechanisms. Hence, these solutions fail to address the question of protecting an application's integrity.

For a non-rooted android device, the system partition provides a secure area which can host applications that cannot be tampered or uninstalled. This secure area can be used to host critical code of an android application which is responsible for its integrity. This code will be part of the logic that will detect modification of the intended user application and terminate it. From the tests performed in this research work, it has been evident that placing the integrity verification logic inside the system partition gave promising results with an accuracy of 100% for a non-rooted emulator environment. Splitting the verification between the client –

server and tokenizing the communication prevents a modified / counterfeit application from accessing the server's resources. It has been shown that any attempts at using the counterfeit application in presence or absence of the system application will be detected and would result in the user application being out of operation. Even if the legitimate user application is used without the system application installed access to the server's resources is denied. This research proves that on a non-rooted android device this solution gives reliable results. The research work carried out would result in a robust android application protection mechanism. This will also prevent the counterfeit applications from running on end user devices and provide the features that were intended to be used in the approved release builds.

This solution has few shortcomings. The first being the delivery of the solution. For this solution to work it is absolutely necessary that the end users have the solution's system application pre-installed on their devices. If the OEMs do not agree to install the solution then the target application remains unprotected and this might block the execution for their users since a valid token is required for communication with the application server. Another limitation is that this solution fails on a rooted android device or emulator and the modified user application runs with no hindrance. This solution provides no anti-reversing solution and must be combined with other techniques such as code obfuscation, dynamic code offloading, etc. which together will provide defence in depth.

Using TEE will greatly enhance the android application's self-protection capability. The TEE provide a TEE O.S which runs in parallel to the normal android O.S. The TEE's resources cannot be accessed from the android O.S but the converse is not true. The TEE can access the android world's resources because the TEE O.S runs at a super-elevated privilege. So, the mission critical part of the android's core logic e.g. the integrity verification logic can be placed and executed from this TEE O.S. This eliminated all the shortcomings of this research work as despite the android O.S operating in root mode the TEE O.S resources remains inaccessible due to its super-elevated privileges. Many OEMs have extended their support in collaborating for the TEE development. On Advanced RISC Machine (ARM) powered devices this TEE technology is known as ARM TrustZone[16]. For others but not limited to, the list is as - On Intel devices its Intel Software Guard Extensions [17](SGX), for Apple its Apple Secure Enclave[18] and for AMD its AMD Platform Security Processor[19]. TEE based research is an emerging and promising area of secure application development. The only challenge being lack of open-source contributions and emulator-based implementations for android. But as the era of android development evolves so will the availability of the TEE implementations for developers.

References

[1] D. Kundu, "JShield: A Java Anti-Reversing Tool", Scholarworks.sjsu.edu, 2011. [Online]. Available: https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1160&context=etd_projects. [Accessed: 17- Aug- 2020].

[2] P. Schulz, "Code Protection in Android", Citeseerx.ist.psu.edu, 2012. [Online]. Available:

[16] https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html

[17] https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html

[18] https://support.apple.com/en-ie/guide/security/sec59b0b31ff/web

[19] https://www.amd.com/en/technologies/pro-security

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.469.6113&rep=rep1&type=pdf. [Accessed: 17- Aug- 2020].

[3] S. Ghosh, S. Tandan, and K. Lahre, “Shielding android application against reverse engineering,” International Journal of Engineering Research and Technology, vol. 2, no. 6, pp. 2635–2643, June 2013.

[4] Y. Wang, "Who Changed You? Obfuscator Identification for Android - IEEE Conference Publication", Ieeexplore.ieee.org, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/7972730. [Accessed: 17- Aug- 2020].

[5] B. Davis, I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. National Science Foundation, 2012, p. 9 [Online]. Available: https://web.cs.ucdavis.edu/~hchen/paper/most2012iarmdroid.pdf. [Accessed: 17- Aug- 2020].

[6] M. Gagnon, "Software Protection through Anti-Debugging - IEEE Journals & Magazine", Ieeexplore.ieee.org, 2007. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4218560. [Accessed: 17- Aug- 2020].

[7] J. Xu, L. Zhang, Y. Sun, D. Lin, and Y. Mao, “Toward a secure android software protection system,” in Proc. of the 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT’15/IUCC’15/DASC’15/PICOM’15), Liverpool, United Kingdom. IEEE, October 2015, pp. 2068–2074.

[8] A. Vasudevan, "Stealth breakpoints", www.researchgate.net, 2006. [Online]. Available: https://www.researchgate.net/publication/4207378_Stealth_breakpoints. [Accessed: 17- Aug- 2020].

[9] K. Lim, "An android application protection scheme against dynamic reverse engineering attacks", Reseachgate.net, 2016. [Online]. Available: https://www.researchgate.net/publication/309529372_An_android_application_protection_scheme_against_dynamic_reverse_engineering_attacks. [Accessed: 17- Aug- 2020].

[10] B. Choi, "An APK Overwrite Scheme for Preventing Modification of Android Applications", ResearchGate, 2014. [Online]. Available: https://www.researchgate.net/publication/273225209_An_APK_Overwrite_Scheme_for_Preventing_Modification_of_Android_Applications. [Accessed: 06- Apr- 2020].

[11] Y. Piao, "Server-based code obfuscation scheme for APK tamper detection", Wiley Online Library, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/sec.936. [Accessed: 06- Apr- 2020].

[12] M. Barbera, "To offload or not to offload? The bandwidth and energy costs of mobile cloud computing - IEEE Conference Publication", Ieeexplore.ieee.org, 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6566921. [Accessed: 17- Aug- 2020].

[13] T. Kim, "Breaking Ad-hoc Runtime Integrity Protection Mechanisms in Android Financial Apps | Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security", Dl.acm.org, 2017. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3052973.3053018. [Accessed: 06- Apr- 2020].

[14] H. Cho, "Mobile application tamper detection scheme using dynamic code injection against repackaging attacks | The Journal of Supercomputing", Dl.acm.org, 2016. [Online]. Available: https://dl.acm.org/doi/10.1007/s11227-016-1763-2. [Accessed: 06- Apr- 2020].

[15] L. Song, "AppIS: Protect Android Apps Against Runtime Repackaging Attacks", Eprints.lancs.ac.uk, 2017. [Online]. Available: https://eprints.lancs.ac.uk/id/eprint/87578/1/ICPADS_2017_paper_160.pdf. [Accessed: 06- Apr- 2020].

[16] J. Bang, "Tamper detection scheme using signature segregation on android platform - IEEE Conference Publication", Ieeexplore.ieee.org, 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7391309. [Accessed: 06- Apr- 2020].

[17] L. Luo, "Repackage-proofing Android Apps", Faculty.ist.psu.edu, 2016. [Online]. Available: https://faculty.ist.psu.edu/wu/papers/ssn-dsn2016.pdf. [Accessed: 06- Apr- 2020].