I. J. Computer Network and Information Security, **2020, 1, 20-26**
Published Online February 2020 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2020.01.03

A Novel Android Security Framework to Prevent Privilege Escalation Attacks

Ahamed K. H. Hussain[1], Mohsen Kakavand[2], Mira Silval[3], and Lingges Arulsamy[4]
Department of Computing, Sunway University, Bandar Sunway, 47500, Malaysia
E-mail: [1]khalid.hh.hussain@gmail.com, [2]mohsenk@sunway.edu.my, [3]16095267@imail.sunway.edu.my, [4]linggest90@gmail.com

Received: 10 November 2019; Accepted: 27 November 2019; Published: 08 February 2020

***Abstract*—**Android is the most popular operating system in the world, with numerous applications having been developed for the platform since its inception, however, it has its fair share of security issues. Despite security precautions taken by developers and the system itself when it comes to permission delegation for applications, privilege escalation attacks are still possible up till Android API level 25. Unfortunately, many existing detection and prevention solutions fall short of the standard necessary or are taxing in resources not found on most Android devices. Proof is shown that a custom created malicious application can elevate its privileges, beyond the permissions it was given, in the existing Android system. In this paper, a modification to the existing Android framework is proposed, one that can detect inter-component communication messages between malicious apps attempting to elevate their privileges and benign applications. Part of this framework is the ability for the user to decide if permissions should be elevated, allowing them some measure of control. The results of the experimental evaluation demonstrate that the solution proposed is effective in preventing privilege escalation attacks on Android API level 24.

***Index Terms*—**Android Security, Privilege Escalation, Permission Escalation.

I. INTRODUCTION

Android is one of the most prevalent operating systems on the planet and sees the highest pervasiveness amongst mobile devices. With this widespread prevalence, however, comes an abundance of security issues, one being privilege escalation attacks. Privilege escalation attacks on Android are a form of attack whereby a nefarious application can utilize a legitimate, yet vulnerable, application's privileged permissions to execute commands that it itself would be incapable of doing so. While there are three forms of privilege escalation attacks [1], there are only two real-world solutions, dynamic or static. Dynamic solutions often include adding to or modifying the existing Android security framework so as to constantly be able to detect privilege escalation attacks and block them as they are being executed such as [2-5]. Static solutions, like those employed in [6-7], meanwhile involve analysis of the applications and systems at certain times and not constantly; i.e it is not real-time protection.

A. Problem Statement

Unfortunately, while methods against privilege escalation have been implemented from API level 26 onwards, devices employing API level 25 and below compose the bulk of the Android market at 61.3 % and are still susceptible to privilege escalation attacks. Furthermore, due to the lack of knowledge about this threat, most consumers do not implement the available solutions; moreover, many of the solutions proposed so far require a certain level of technical skill and knowledge to implement, which are barriers that make it hard for these solutions to be adopted.

B. Objectives

The goal of this work is to modify the existing Android security framework to monitor situations where the Inter-Component Communication messages between applications are being exploited to achieve privilege escalation. This solution will maintain the state of applications as they run whilst overseeing ICCs in between different applications similar to [8], unlike existing static methods that cannot be run in real-time or other dynamic methods that are costly in terms of resources.

The objectives shall be achieved by making modifications to the existing Android framework, specifically the Activity Manager as well as creating two new components, to inspect ICCs between applications.

The rest of this paper is in the following order: we discuss related solutions proposed by other third parties in section II. Section III details the components Android applications and the requirements they need to function. Section IV demonstrates current privilege escalation vulnerabilities using modified applications. Section V goes into detail about the proposed modifications we make to the Android framework to prevent privilege escalation. Section VI describes the experimental evaluation of our modified Android framework. The main conclusions are briefed in section VII.

Copyright © 2020 MECS

II. Related Works

In [8] the authors opted for a dynamic solution involving a mixture of security mechanisms would be the best defense in. The primary of those being a modification of the existing Android security framework that would analyze third-party applications and mediate the intercomponent communication (ICC) between said apps, along with a configurable policy system with capability-based rule system for users to modify as well as a corresponding risk mitigation mechanism for reducing risks incurred by user-made policies. Lastly, a sophisticated access decision cache is created to store information about applications and their states and security policies. The authors of this paper ultimately tested it in an environment of 60 apps, 5 of them being customized malicious apps. While the study was mostly successful in blocking malicious actions, a number of false positives were also reported as well as the blocking of benign ICCs. However, the paper did not experiment in an environment where malicious apps could work together for collusion attacks and did not cover any other communication methods apps could utilize, other than ICC.

XManDroid [9] also creates a security framework that extends Android's existing monitoring mechanism to detect and privilege escalation attacks at the application level, based on a system centric policy, much like [8]. XManDroid would dynamically analyze the transitive permission usage of applications, allowing for effective detection of covert channels between system services and content providers whilst minimizing the rate of false positives, depending on the system policy, which can be defective. However, this paper does not take into account privilege escalation attacks at the kernel level, or application level attacks that are controlled by the underlying kernel. The study tested XManDroid against 7 scenarios involving a combination of 2 applications, one of them being the vulnerable application, in certain scenarios. XManDroid was able to detect and prevent all attacks.

Quaintroid [10] utilized a quantitative approach towards detecting and preventing privilege escalation attacks. The authors utilized a variant of TaintDroid [11], dubbed Quantdroid, as well as an additional service called the Flowgraph which monitors the ICCs of apps on the go. Unlike the previous studies, the authors of this study touched upon collusion attacks. Ultimately, the authors were able to utilize both the FlowGraph and Quantdroid to detect privilege escalation in both collusion attacks and unprotected interfaces, something neither of the previous studies have been able to do.

Meanwhile, RoppDroid [12] provides a resource virtualization framework to mitigate permission leak threats caused by ICCs without ruining usability of the app in question. This is done by dynamically virtualizing specific resources, so as to mitigate privilege escalation problems by considering the interactions of ICCs amongst apps. Therefore, malicious apps can access only virtualized resources in a sandbox as opposed to real resources.

AppScalpel [13] is a privacy-preserving system to prevent malicious utilization of sensitive data. The authors utilized static analysis to extract contextual information about data usage behaviors within applications. Once these behaviors had been analyzed so as to identify ones involving nefarious usage of sensitive data, rule enforcers on each data-flow path would be implemented. Care was taken to only block undesirable usage of sensitive data and not to affect usability. Data usage behavior was extracted using [9-10]. This behavior was then analyzed by AppScalpel, which would categorize said behaviors as either common or suspicious. Suspicious behavior could optionally then be manually identified. Rules about the applications involved would then be generated and subsequently enforced. To evaluate their solution, the authors utilized four datasets of applications obtained from the Google Play Store, MalGenome, Drein, and VirusShare, for a total of over 5766 applications. Due to being a static analysis method, the authors agree that it consumes large amounts of time and memory; the latter not being available in large amounts on smartphones. As such, it is not suited for real-time defense.

Another such static solution is ICCTA [9], which is a taint analyzer that detects privacy leaks amongst the components in an application. The authors describe how the specificity of Android applications make them statistically difficult to analyze. To overcome this problem, the authors designed their tool with a two-step approach; ICC links extraction and the taint flow analysis for the ICCs. Link extraction refers to the steps the authors incorporated to detect components that held sensitive information, called sources, and components that would access that information, called sinks; furthermore, the link between these two components is what will be deciphered, so as to be analyzed. Meanwhile, taint flow analysis attempts to follow the flow of the sensitive information across components, despite the short-lived nature of some Android application components. To evaluate their solution, the authors of this paper utilized a dataset of 22 custom applications, the MalGenome dataset of 1260 malware applications, and 15,000 applications from the Google Play Store. Unfortunately, because this is a static solution; it cannot prevent real-time dynamic privilege escalation. Furthermore, this solution only tracks ICC leaks within an application not across applications.

III. Android Application Components & Requirements

Android applications are composed of four different components: Activities, Services, Broadcast receivers and Content providers. Furthermore, for most applications to function they also make use of the following: Permissions, Intents, and optionally, intent filters.

Activities represent the single display users see when they have an application running and as such are the primary way users can interact with an application.

Copyright © 2020 MECS

Services allow applications to run in the background to perform long-running processes that do not have a visual interface, or Activity.

Broadcast receiver is a component that allows the applications to respond to system-wide broadcasts from either the system itself or from other applications, even when the receiving application itself isn't running.

A content provider is just that, it provides content or data to other applications or the system when they query for it, assuming they are allowed access to such data.

Another important feature needed for most applications to function are their permissions, which are declared in the Manifest file of the Android application. Some components of the applications, such as services or activities may not function properly or at all if not granted these permissions.

Intents are types of Inter-Component Communication messages that allow three of the four components mentioned above, activities, services, and broadcast receivers, to be activated by a separate application or even the system. As such these Intent-based ICC messages are the simplest way applications can communicate with each other to start activities, pass information, or to query for information. As a result, they form the basic building block of privilege escalation attacks. There are two types of intents: implicit intents and explicit intents. Implicit intents do not name a specific component or application, but instead declare an action that needs to be executed. The Android system then proceeds to query for components that can handle the action and will ask the users input on the selection. Meanwhile, explicit intents specify exactly which component or application is needed to perform an action. Generally explicit intents are only used to call components within an application, but they can be used by malicious applications to call unprotected benign applications to execute actions. It should be noted that explicit intents do not require the input of the user, who remain oblivious that such an intent occurred.

Lastly, intent filters are used to declare what intents an application can respond to. In the case of an implicit intent, if an intent matches an intent filter in another application, that application can be selected by the user to perform the action specified by the application sending the intent. However, in the case of explicit intents, intent filters need not be declared; the action can be executed so long as the initiating application can specify the name of the component in the receiving application. Intent filters are declared in the manifest file alongside the permissions.

IV. Vulnerability Testing

The privilege escalation this study will be testing for is between two different applications; one that is malicious and one that is benign. The benign application is the SendSMS application that comes with the Droidbench test suite. The second, and malicious application, is a custom variant of the SendSMS application that is called Read_ID. It should be noted that the existing SendSMS benign application was also slightly altered for this test. The malicious Read_ID application is granted the ability to read the phone IMEI number through the READ_PHONE_STATE permission that is declared in its manifest file. The SendSMS application is only given the ability to send an SMS, with the relevant permission declared in the manifest.

As seen by the data flow in Fig.1, when the user interacts with the malicious Read_ID application, the Read_ID application acquires the IMEI from the system (1) but does not have the permission to send an SMS so it is unable to do so as seen by (2). Instead it passes the IMEI and a preprogrammed phone number as an explicit intent in an ICC message to the SendSMS application (3). The SendSMS application, upon receiving the intent through an unprotected intent filter, automatically sends a text message containing the IMEI to the preprogrammed number (4).

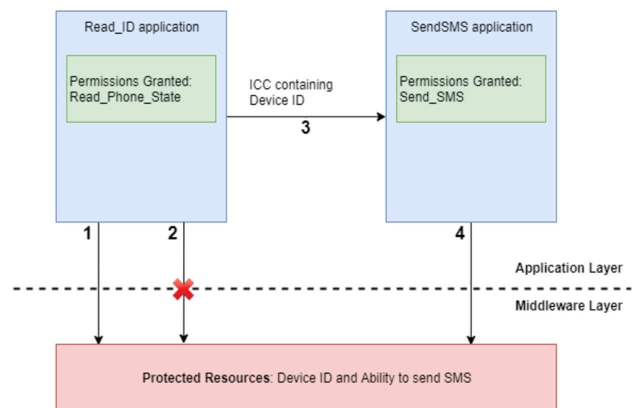

Fig.1. Android Framework Modifications

V. Modification to Android Framework

The proposed modifications to the Android Framework will take place in Android's middleware layer. To defend against privilege escalation Android's Activity Manager will be modified to prevent privilege escalation by applications that have not been granted the privileged permission explicitly. The Activity Manager provides information and interacts with, activities, services, and the processes being run on the Android system. Furthermore, a new component called a Resolver will be introduced, along with a Decision cache to store information.

The proposed framework regarding the modification of the Android Framework and the order of steps that will take place can be seen in Fig.2. The first thing the Activity Manager should do is check if the ICC was between two third-party applications (step 1), by utilizing the checkComponentPermission() method. This information is obtained through Android's Package Manager Service (step 2), a service that maintains runtime information for each application, such as User ID (UID), granted permissions, etc. If the ICC is found to be between two third-party applications (step 3), the next step to occur should be the invocation of a new module, the Resolver (step 4).

Copyright © 2020 MECS

The Resolver upon being activated proceeds to compare the two applications involved in the Inter-Component Communication intent message as well as their designated permissions. When identifying that the two applications are different, it then invokes a new component called UserInputConfirm, which presents the user of the device an alert box that gives them the option to allow or deny the Inter-Component Communication intent message. Upon receiving the user's choice, the UserInputConfirm module then relays that choice back to the Resolver, which then either allows the Activity Manager to grant or deny the connection.

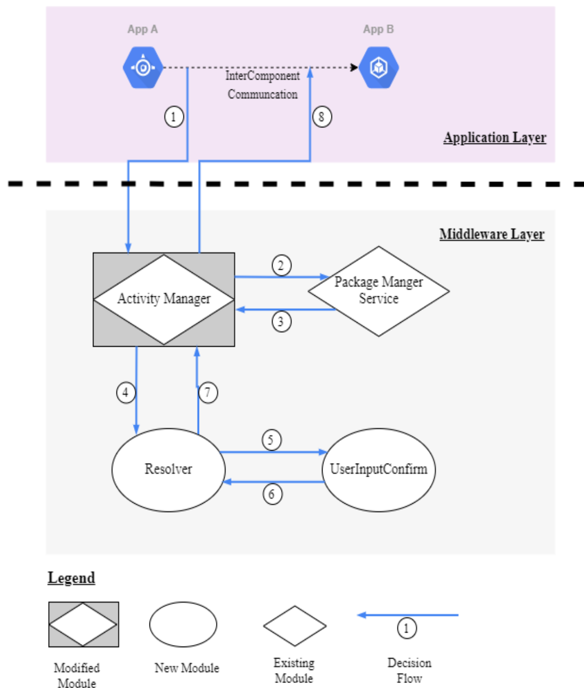

Fig.2. Android Framework Modifications

A. Activity Manager Modifications

One of the most important modules in the Android Framework is the Activity Manager. The Activity Manager is responsible for interacting with activities, services, and other processes. Key amongst its responsibilities is mediating ICC interactions between applications. To ensure that on this version of Android that privilege escalation can be minimized, one of the functions in the Activity Manager, specifically the checkComponentPermission() function is modified.

The checkComponentPermission() function is called by the Android System when a third-party application initiates a connection with another application via an ICC that involves the use of a permission. As of Android API level 24, the checkComponentPermission() function will merely check if only one of the applications has the permission that is being accessed. To make this check, it accepts the following data when it is called: the permission required, the UID of the calling application, and the UID of the application owning the permission. In the scenario above, the nefarious application is the calling application and the benign application is the application owning the permission. However, it should be noted that it does not check to compare if both applications have the permission being accessed.

As a result, changes made to the checkComponentPermission() method include an IF statement that compares the IDs of the applications that initiated the checkComponentPermission() method. Due to how the checkComponentPermission() method is structured when it comes to decision making, the custom IF statement had to be placed first, becoming the first virtual barrier check. The checkComponentPermission() then goes on to create a new instance of the Decision Maker method from the custom Resolver class. As it creates the new instance, it also passes along the variables involved, i.e: the permission required, the UID of the calling application, and the UID of the application owning the permission. At this point it effectively hands over the decision process to the Resolver class and will execute what the Resolver.DM() method returns.

B. Resolver

The custom Resolver class is a custom class created to do very simple comparisons between the two applications involved in the Inter-Component Communication message. It was placed in the same directory, or package, as the Activity Manager for easier referencing when it came to coding. To ensure the custom Resolver class was able to communicate with the other classes involved, the android.app and android.content.pm packages had to be imported, with the former containing system classes and methods such as the Activity Manager and the latter pointing towards classes and methods associated with the Package Manager service. One of the methods in the Resolver class is the Decision Maker method.

The first thing the Decision Maker method does is acquire the package names for the UIDs it has. It does this by invoking the getPackagesForUid() method in the Package Manager. The getPackagesForUid method is an abstract method that retrieves the names of all packages that are associated with a particular UID. In most cases, this will be a single package name, the package that has been assigned that UID. In this instance it invokes it twice, first as getPackagesForUid(1) and then as getPackagesForUid(2). The package names are returned as strings that are then stored in the u1package or u2package global variables.

The reason Android does not solely work off of UIDs is due to the fact that UIDs are assigned to applications/packages when they are installed; when those applications/packages are uninstalled, the UIDs are freed up to be used by other applications/packages that may get installed. Thus, UIDs can be seen as an abstract pointer to the applications/packages, which is actually used by the Android system, such as the Package Manager Service, to contain information related to permissions, installation dates, etc.

The next step executed in the Decision Maker method is to verify whether the applications involved in the Inter-Component Communication message possess the permission stored in the global variable p1. To do this,

the Decision Maker invokes the checkPermission() method in the Package manager. The checkPermission() method is an abstract method that accepts two string values, the permission name and the package name. It returns an integer value upon execution. It invokes the method twice, first as checkPermission(p1, u1package) and then again as checkPermission(p1, u2package), checking the same permission across the two different packages, which correlate to the two applications involved in the ICC. This information about the two different packages were retrieved in the previous step by the getPackagesForUid() method. The results of the checkPermission() method; an integer value that indicates whether the application has the permission or not, is then stored in the global variables u1permission and u2permission respectively.

The Decision Maker then goes to check if the nefarious application has the permission or not in an If statement. When the condition of the If statement is met (nefarious application does not have the permission that belongs to the benign application), a new custom activity called UserInputConfirm is started.

C. User Input

The UserInputConfrim activity is a simple popup dialog box that asks the user to confirm whether or not to allow the nefarious application to access the permission used by the benign application. For the purpose of this project, the message asked is hardcoded to represent the nefarious and benign applications that will be tested; furthermore, the activity class is located in the same package as the Resolver for easier referencing.

The Android class used to build this popup is the AlertDialog class. Users are presented an alert box that provides the user with options of allowing the nefarious application to use the permissions of the benign application. If the user chooses to not allow the nefarious application to access the permissions of the benign application, then the UserInputActivity will return the integer value -1 to the Decision Maker and close. Similarly, if the user chooses to allow the nefarious application to access the permissions of the benign application, the activity will return the integer value 0 to the Decision Maker and then close.

The Decision Maker, upon receiving the value the user has chosen proceeds to store it in the global variable “decision”. After storing the decision, the Decision Maker then delves further into a nested If statement, that compares the value in the “decision” variable with either -1 or 0. If the value is equal to -1, it indicates that the user has chosen not to grant permissions to the nefarious application and the as such the Decision Maker will return the Package Manager method PERMISSION_DENIED to the Activity Manager, which will proceed to terminate the Inter-Component Communication message between the two applications. However, if the user has chosen to grant the permission to the nefarious application, the Decision Maker will instead return the Package Manager method PERMISSION_GRANTED to the Activity Manager, which will in turn allow the Inter-Component Communication message to proceed.

VI. Experimental & Discussion

Upon completion of the modifications to the Android Framework, a fresh system image containing the changes made within the framework was built on the Ubuntu virtual machine and then flashed to an emulator as a new Android Virtual Device (AVD) based on a Nexus 6P. This AVD was then loaded with 50 of the top most downloaded applications from the Google Play Store as well as a custom malicious application called Read_ID and a custom benign application called Send_SMS. For this project the type of Inter-Component Communication functions that was tested was explicit intents. The malicious application will attempt to send the benign application malicious ICC messages containing the Device ID, which will then be sent as an SMS.

The solution proposed in this paper is expected to prevent the malicious ICC message between applications , ensuring that the subsequent SMS is not sent.

A. Functional Effectiveness

Due to the modifications made to the Android framework, the user is now presented with an alert dialog box, as seen in Fig.3, asking the user if they would like to allow the malicious application to utilize the permissions of the benign application via the Inter-Component Communication message.

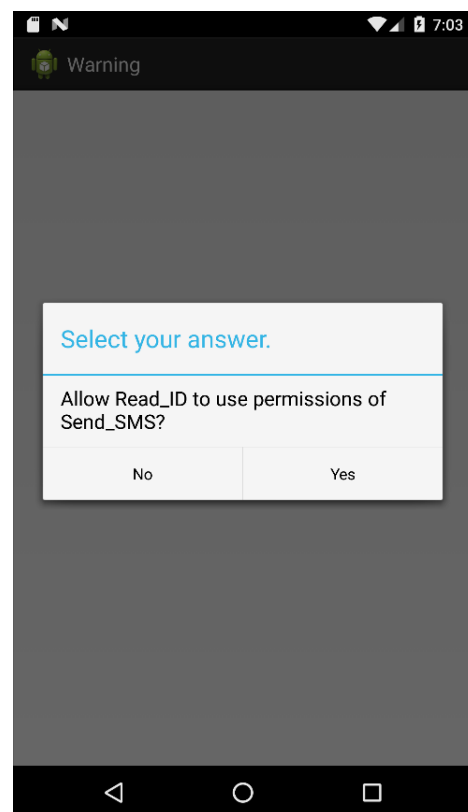

Fig.3. Alert dialog box notifying user of privilege escalation

The user can then proceed to either choose to allow access, upon which the ICC message will be successful and a text message will be sent to the preprogrammed number or they could choose to decline the ICC message; upon which no message will be sent and the attempt at privilege escalation via the Inter-Component Communication message will be prevented. Furthermore, the alert dialog should bring attention to the user of the nefarious application on the system, which should lead to its removal.

As a result, in this new environment, the modifications to the Android Framework successfully prevented the malicious Inter-Component Communication messages from being sent; assuming the user chose to decline giving the malicious application access to the permissions of the benign application. This can be seen by the lack of a text message containing the device ID being sent. therefore, it is safe to assume that the modifications to the Android Framework successfully prevented privilege escalation attacks from occurring on the Android platform, at least in the case of the dataset used.

B. Performance Evaluation

Activities and interactions between the two applications were first tested and benchmarked without the modifications to the Android Framework. These benchmarks of time it takes for the ICC messages to being sent were recorded and then compared to a benchmark taken after the implementation of the solution. Due to the testing environment being an emulator; the time cost between sending the malicious intent and the benign application's receiving the intent was measured five times to get an average.

As we are measuring the time it takes for the intent to be received, for the post-solution test we are allowing the malicious ICC instead of blocking it. Furthermore, both application processes had to be killed via the force-stop option in the system settings to ensure a fair environment. These time costs are presented in Table 1, where the performance overhead post-solution is measured at 331.8 milliseconds, compared to the pre-solution value of 219.6 milliseconds, for a difference of 112.2 milliseconds, on average. As can be seen, the increase in the performance overhead is negligible and should not be noticeable to the user. However, the greatest variance that may affect this statistic when it comes to the post-solution system is the response time of the user when selecting if they should allow the ICC message through or if they should deny it.

Table 1. Performance Evaluation of ICC cost times

System	Average Time (ms)	Min Time (ms)	Max Time (ms)
Android API 24 Pre-Solution	219.6	187	268
Android API 24 Post-Solution	331.8	267	373

C. Limitations

Currently, the solution proposed in this paper can help resolve explicit Inter-Component Communication messages between applications; however, it cannot provide a defense against collusion attacks, which involve more than one nefarious application working in conjunction to enable privilege escalation. Another issue that can crop up is when malicious application can set its UID to the UID of the benign application; which is possible by malicious applications altering settings in the Package Manager Service. Lastly, this solution is only viable if the malicious program does not have root access; as root access applications will be able to bypass many of the defenses and in some cases, disable those defenses.

VII. Conclusions & Future Work

Android privilege escalation attacks are some of the easiest attacks to perform both due in part to app developers who do not have the necessary security knowledge to prevent their apps from being utilized as confused deputies, as well as the fact that most users are also not technologically adept. However, the most glaring cause of Android privilege escalation attacks up till API level 25 is the existing Android security framework that allows for this type of attack.

In an aim to take the burden off both parties, this project modified and added to the existing Android security framework via changes made using the Android Open Source Project. This project successfully mitigated privilege escalation attacks by monitoring inter-component communication between applications on API level 24 and resolved instances of perceived privilege escalation that triggered the mechanism built into the framework, with help from the user.

However, since this project has a specific scope, future work should also look at ways of mitigating privilege escalation attacks that are based on collusion. This can be done by analyzing the flow of data between multiple applications as opposed to just two. Furthermore, a policy should be implemented that can be used so that the system can make smart decisions about whether allowing certain inter-component communication messages through or not, instead of having to rely on the user. Another feature that could be added is informing the users exactly which permissions are trying to be obtained by the nefarious application.

References

[1] Z. Fang, W. Han, and Y. Li, "Permission based Android security: Issues and countermeasures," *Computers and Security*, vol. 43. Elsevier Ltd, pp. 205–218, 2014.

[2] R. H. Niazi, J. A. Shamsi, T. Waseem, and M. M. Khan, "Signature-based detection of privilege-escalation attacks on Android," in *Proceedings - 2015 Conference on Information Assurance and Cyber Security, CIACS 2015*, 2016, pp. 44–49.

[3] Y. Park, C. Lee, J. Kim, S.-J. Cho, and J. Choi, "An Android Security Extension to Protect Personal Information against Illegal Accesses and Privilege Escalation Attacks," *J. Internet Serv. Inf. Secur.*, vol. 2, pp. 29–42, 2012.

[4] H. T. Lee, D. Kim, M. Park, and S. J. Cho, "Protecting data on android platform against privilege escalation

Copyright © 2020 MECS

attack," *Int. J. Comput. Math.*, vol. 93, no. 2, pp. 401–414, Feb. 2016.
[5] B. Kong, Y. Li, and L.-P. Ma, "PtmxGuard: An Improved Method for Android Kernel to Prevent Privilege Escalation Attack," *ITM Web Conf.*, vol. 12, p. 05010, Sep. 2017.
[6] X. Zhong, F. Zeng, Z. Cheng, N. Xie, X. Qin, and S. Guo, "Privilege Escalation Detecting in Android Applications," in *Proceedings - 2017 3rd International Conference on Big Data Computing and Communications, BigCom 2017*, 2017, pp. 39–44.
[7] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "COVERT: Compositional Analysis of Android Inter-App Permission Leakage," *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 866–886, 2015.
[8] Y. Xu, G. Wang, J. Ren, and Y. Zhang, "An adaptive and configurable protection framework against android privilege escalation threats," *Futur. Gener. Comput. Syst.*, vol. 92, pp. 210–224, Mar. 2018.
[9] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Tech. Univ. Darmstadt, Tech. Rep. TR-2011-04*, pp. 1–18, 2011.
[10] T. Markmann, D. Gessner, and D. Westhoff, "QuantDroid: Quantitative approach towards mitigating privilege escalation on Android," in *IEEE International Conference on Communications*, 2013, pp. 2144–2149.
[11] W. Enck *et al.*, "Taint droid: An information flow tracking system for real-time privacy monitoring on smartphones," *Commun. ACM*, vol. 57, no. 3, pp. 99–106, Mar. 2014.
[12] T. Dai, X. Li, B. Hassanshahi, R. H. C. Yap, and Z. Liang, "ROPPDROID: Robust permission re-delegation prevention in Android inter-component communication," *Comput. Secur.*, vol. 68, pp. 98–111, 2017.
[13] Z. Meng, Y. Xiong, W. Huang, L. Qin, X. Jin, and H. Yan, "AppScalpel: Combining static analysis and outlier detection to identify and prune undesirable usage of sensitive data in Android applications," *Neurocomputing*, vol. 341, pp. 10–25, 2019.
[14] L. Li *et al.*, "IccTA: Detecting inter-component privacy leaks in android apps," *Proc. - Int. Conf. Softw. Eng.*, vol. 1, pp. 280–291, 2015.
[15] S. Arzt *et al.*, "FLOWDROID: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, 2014.

Authors' Profiles

AHAMED K. H. HUSSAIN is currently pursuing a BSc (Hons) Information Technology (Computer Networking & Security) at Sunway University's Department of Computing & Information Systems, in Malaysia, where he is also a student researcher. His research interests include computer, network, and mobile security.

MOHSEN KAKAVAND received his Ph.D. degree in intelligent computing from the University Putra Malaysia (UPM), Malaysia in 2017. He is currently a Lecturer with the Department of Computing and Information Systems, Faculty of Science and Technology, Sunway University in Malaysia. His research interests include aspects of data mining, intelligent computing, machine learning, intrusion detection systems (IDSs), and cybersecurity.

MIRA SILWAL is currently pursing dual degree in BSc (Hons) Information Technology (Computer Networking and Security) from Sunway University and Lancaster University. She is a prolific IT enthusiast whose research interests includes the aspects of data mining, machine learning, intrusion detection systems (IDSs) and cybersecurity.

LINGGES ARULSAMY is currently pursuing dual degree in BSc (Hons) Information Technology (Computer Networking and Security) from Sunway university and Lancaster university. He is an active researcher in the area of Artificial Intelligence, Crypto-Ransomware Anomaly Classification, Image Transfiguration using Horizontal Feature Simplification, Encryption Through Pangram, Smart Light Using Electromagnetic Wave and Cybersecurity.

How to cite this paper: Ahamed K. H. Hussain, Mohsen Kakavand, Mira Silval, Lingges Arulsamy, "A Novel Android Security Framework to Prevent Privilege Escalation Attacks", International Journal of Computer Network and Information Security(IJCNIS), Vol.12, No.1, pp.20-26, 2020. DOI: 10.5815/ijcnis.2020.01.03

Copyright © 2020 MECS