International Journal of Mathematical Archive-2(11), 2011, Page: 2475-2479
Available online through www.ijma.info ISSN 2229 – 5046

AUTHORIZING STOCKPILE ATTACKS ON ANDROID

[1]B. V. S. S. R. S. Sastry* & [2]K. Akshitha

[1]Department of Information Technology, Aurora's Engineering College, Andhra Pradesh, India

E-mail: sastry_38@yahoo.com

[2]Department of Information Technology, Aurora's Engineering College, Andhra Pradesh, India

E-mail: koluguri.87@gmail.com

(Received on: 27-10-11; Accepted on: 10-11-11)

ABSTRACT

Android is a software platform and Operating System for mobile devices, based on the Unix kernel, developed by Google and later the Open Handset Alliance. It allows developers to write managed code in Java language. Android is a modern and popular software platform for smart-phones. Among its predominant features is an advanced security model which is based on application-oriented mandatory access control and sandboxing.[1] This allows developers and users to restrict the execution of an application to the privileges it has (mandatorily) assigned at installation time. The exploitation of vulnerabilities in program code is hence believed to be confined within the privilege boundaries of an application's sandbox.[2] However, in this paper we show that a Authorizing Stockpile attack is possible. We show that a genuine application exploited at runtime or a malicious application can escalate granted permissions. Our results immediately imply that Android's security model cannot deal with a transitive permission usage attack.

KEYWORDS: ANDROID, SMARTPHONES, ATTACKS.

1. INTRODUCTION:

Mobile phones play an important role in today's world and have become an integral part of our daily life as one of the predominant means of communication. Smartphones are increasingly prevalent and adept at handling more tasks from web-browsing and emailing, to multimedia and entertainment applications (games, videos, audios), navigation, trading stocks, and electronic purchase. However, the popularity of smartphones and the vast number of the corresponding applications makes these platforms also more attractive targets to attackers. Currently, various forms of malware exist for smartphone platforms, also for Android [33, 8]. Moreover, advanced attack techniques, such as code injection [17], return-oriented programming (ROP) [29] and ROP without returns [6] affect applications and system components at runtime. As a last resort against malware and runtime attacks, well-established security features of today's smartphones are application sandboxing and privileged access to advanced functionality.

Resources of sandboxed applications are isolated from each other, additionally; each application can be assigned a bounded set of privileges allowing an application to use protected functionality. Hence, if an application is malicious or becomes compromised, it is only able to perform actions which are explicitly allowed in the application's sandbox. For instance, a malicious or compromised email client may access the email database as it has associated privileges, but it is not permitted to access the SMS database.

Android implements application sandboxing based on an application-oriented mandatory access control. Technically, this is realized by assigning each application its own UserID and a set of permissions, which are fixed at installation time and cannot be changed afterwards. Permissions are needed to access system resources or to communicate with other applications.

Android checks corresponding permission assignments at runtime. Hence, an application is not allowed to access privileged resources without having the right permissions. However, in this paper we show that Android's sandbox model is conceptually awed and actually allows Authorizing Stockpile attacks. While Android provides a well-structured permission system, it does not protect against a transitive permission usage, which ultimately results in allowing an adversary to perform actions the application's sandbox is not authorized to do. Note that this is not an implementation bug, but rather a fundamental aw. In particular, our contributions are as follows:

**Corresponding author: [1]B. V. S. S. R. S. Sastry*,*E-mail: sastry_38@yahoo.com*

- Authorizing Stockpile attacks: We describe the conceptual weakness of Android's permission mechanism that may lead to Authorizing Stockpile attacks (Section 3). Basically, Android does not deal with transitive privilege usage, which allows applications to bypass restrictions imposed by their sandboxes.

- Concrete attack scenario: We instantiate the permission escalation attack and present the details of our implementation (Section 4). In particular, in our attack a non-privileged and runtime-compromised application is able to bypass restrictions of its sandbox and to send multiple text messages to a phone number chosen by the adversary. Technically, for runtime compromise we use a recent memory exploitation technique, return-oriented programming without returns [9, 6], which bypasses memory-protection mechanisms and return-address checkers and hence assumes a strong adversary model.

As a major result, our findings imply that Android's sandbox model practically fails in providing confinement boundaries against runtime attacks. Because the permission system does not include checks for transitive privilege usage, attackers are able to escape out of Android's sandbox.

2. ANDROID:

Before we elaborate on our attack, we briefly describe the architecture of Android and its security mechanisms.

2.1 ANDROID ARCHITECTURE:

Android is an open source software platform for mobile devices. It includes a Linux kernel, middleware framework, and core applications. The Linux kernel provides low-level services to the rest of the system, such as networking, storage, memory, and processing. A middleware layer consists of native Android libraries (written in C/C++), an optimized version of a Java Virtual Machine called Dalvik Virtual Machine (DVM), and core libraries written in Java.

The DVM executes binaries of applications residing in higher layers. Android applications
are written in Java and consist of separated modules, so-called components. Components can communicate to each other and to components of other applications through an inter component communication (ICC) mechanism provided by the Android middleware called Binder1.

As Android applications are written in Java, they are basically protected against standard buffer overflow attacks [1] due to the implicit bound checking. However, Java-applications can also access C/C++ code libraries via the Java Native Interface (JNI). Developers may use JNI to incorporate own C/C++ libraries into the program code, e.g., due to performance reasons. Moreover, many C libraries are mapped by default to fixed memory addresses in the program
memory space. Due to the inclusion of C/C++ libraries, the security guarantees provided by the Java programming language do not hold any longer. In particular, Tan and Croft [31] identified various vulnerabilities in native code of the JDK (Java Development Kit).

2.2 ANDROID SECURITY MECHANISMS:

Discretionary Access Control (DAC): The DAC mechanism is inherited from Linux, which controls access to files by process ownership. Each running process (i.e., subject) is assigned a User ID, while for each file (i.e., object) access rules are specified. Each file is assigned access rules for three sets of subjects: user, group and everyone. Each subject set may have permissions to read, write and execute a file.

Sandboxing: Sandboxing isolates applications from each other and from system resources. System files are owned by either the \system" or \root" user, while other applications have own unique identifiers. In this way, an application can only access files owned by itself or files of other applications that are explicitly marked as readable/writable/executable for others.

Permission Mechanism: The permission mechanism is provided by the middleware layer of Android. A reference monitor enforces mandatory access control (MAC) on ICC calls. Security sensitive interfaces are protected by standard Android permissions such as PHONE CALLS, INTERNET, SEND SMS meaning that applications have to possess these permissions to be able to perform phone calls, to access the Internet or to send text messages. Additionally, applications may declare custom types of permission labels to restrict access to own interfaces. Required permissions are explicitly specified in a Manifest file and are approved at installation time based on checks against the signatures of the applications declaring these permissions and on user confirmation. At runtime, when an ICC call is requested by a component, the references monitor checks whether the application of this component possesses appropriate permissions. Additionally, application developers may place reference monitor hooks directly into the code of components to verify permissions granted to the ICC call initiator.

Component Encapsulation: Application components can be specified as public or private. Private components are accessible only by components within the same application. When declared as public, components are reachable by other applications as well, however, full access can be limited by requiring calling applications to have specified permissions.

Application Signing: Android uses cryptographic signatures to verify the origin of applications and to establish trust relationships among them. Therefore, developers have to sign the application code. This allows to enable signature-based permissions, or to allow applications from the same origin (i.e., signed by the same developer) to share the same UserID. A certificate of the signing key can be self-signed and does not need to be issued by a certification authority. The certificate is enclosed into the application installation package such that the signature made by the developer can be validated at installation time.

3. AUTHORIZING STOCKPILE ATTACKS ON ANDROID:

In this section, we describe security deficiencies of Android's permission mechanism, which may lead to Authorizing Stockpile attacks instantiated by compromised applications. We state the problem like following:

An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee). In other words, Android's security architecture does not ensure that a caller is assigned at least the same permissions as a callee.

Figure 1 shows the situation in which Authorizing Stockpile attack becomes possible. Applications A, B and C are assumed to run on Android, each of them is isolated in its own sandbox. A has no granted permissions and consists of components CA1 and CA2 . B is granted a permission p1 and consists of components CB1 and CB2 . Neither CB1 nor CB2 are protected by permission labels and thus can be accessed by any application. Both, CB1 and CB2 can access components of external applications protected with the permission label p1 , since in general all application components inherit permissions granted to their application. C has no permissions granted, it consists of components CC1 and CC2 . CC1 and CC2 are protected by permission labels p1 and p2 , respectively, that means that CC1 can be accessed only by components of applications which possess p1 , while CC2 is accessible by components of applications granted permission p2 .

As we can see in Figure 1, component CA1 is not able to access CC1 component, since p1 permission is not granted to the application A. Nevertheless, data from component CA1 can reach component CC1 indirectly, via the CB1 component. Indeed, CB1 can be accessed by CA1 since CB1 is not protected by any permission label. In turn, CB1 is able to access CC1 component since the application B and consequently all its components are granted p1 permission.

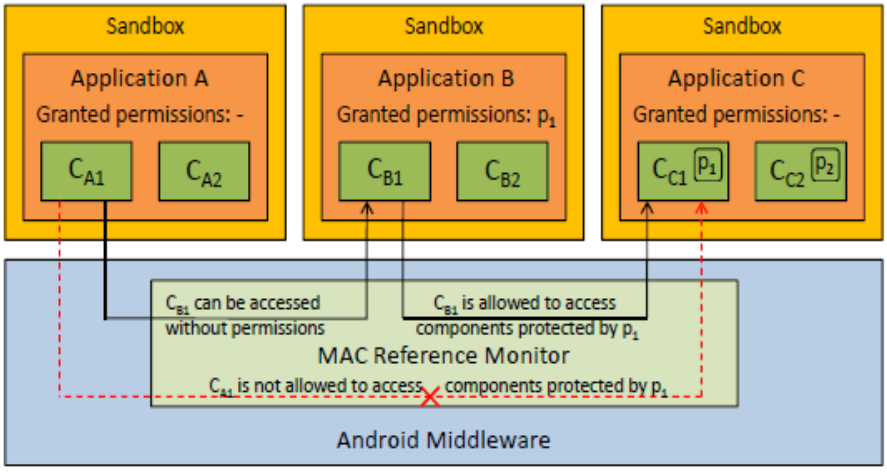

FIGURE 1: COMPONENT BASED AUTHORIZED STOCKPILE ATTACK

To prevent the attack described above, the application B must enforce additional checks on permissions to ensure that the application calling CB1 component is granted a permission p1 . Generally, it can be done by means of reference monitor hooks included in the code of the component. However, the problem is that the task to perform these checks is delegated to application developers, instead of being enforced by the system in a centralized way. This is an error-prone approach as application developers in general are not security experts, and hence their applications may fail to implement necessary permission checks.

5. CONCLUSIONS:

In this paper, we showed that it is possible to mount Authorizing Stockpile attacks on the well-established Google Android platform. We identified a severe security deficiency in Android's application-oriented mandatory access control mechanism (also referred as a permission mechanism) that allows transitive permission usage. In our attack example, we were able to escalate privileges granted to the application's sandbox and to send a number of text messages (SMS) to a chosen number without corresponding permissions.

For the attack, we subverted the control flow of a non-privileged vulnerable application by means of a sophisticated runtime compromise technique called return-oriented programming (ROP) without returns [9, 6]. Next, we performed a Authorizing Stockpile attack by misusing a higher-privileged application.

Our attack illustrates the severe problem of Android's security architecture: Non-privileged applications can escalate permissions by invoking poorly designed higher-privileged applications that do not sufficiently protect their interfaces. Although recently proposed extensions to Android security mechanisms [20, 11] aim to address the problem of poorly designed applications, they suffer from practical shortcomings. Saint [20] provides a means to protect interfaces of applications, but relies on application developers to define Saint policies correctly, while Kirin [11] can detect data flows allowing Authorizing Stockpile attacks, but results in false positives.

In our future work we aim to enhance Android's security architecture in order to prevent (as opposite to detect) Authorizing Stockpile attacks without relying on secure development by application developers.

ACKNOWLEDGEMENTS:

The authors would like to thank everyone, who supported in the preparation of the paper.

REFERENCES:

[1] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, 49(14), 1996.

[2] D. Barrera, H. G. Kayacik, P. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission based security models and its application to Android. In ACM CCS' 2010, Oct 2010.

[3] J. Burns. Developing secure mobile applications for Android. http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf, 2008.

[4] J. Burns. Black Hat 2009. Mobile application security on Android, 2009.

[5] A. Chaudhuri. Language-based security on Android. In PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, pages 1{7, 2009.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In ACM CCS 2010, Oct 2010.

[7] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In International Conference on Distributed Computing Systems, pages 409{417. IEEE Computer Society, 2001.

[8] cnet news. First SMS-sending Android Trojan reported. http://news.cnet.com/8301-27080_3-20013222-245.html, August 2010.

[9] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on ARM. Technical Report HGI-TR-2010- 002, Ruhr-University Bochum, July 2010.

[10] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. http://www.trust.rub.de/media/trust/veroeffentlichungen/2010/03/20/ROPdefender.pdf, March 2010.

[11] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, Sep 2008.

[12] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certi_cation. In ACM CCS '09, pages 235{245. ACM, 2009.

[13] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. IEEE Security and Privacy, 7(1):50{57, 2009.

[14] S. Gupta, P. Pratap, H. Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In WODA '06, pages 65{72, New York, NY, USA, 2006. ACM.

[15] A. Lineberry, D. L. Richardson, and T. Wyatt. Therse aren't the permissions you're looking for. BlackHat USA 2010. http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf, 2010.

Authors:

B. V. S. S. R. S. Sastry has been graduated with B. Tech in 2009 from Aurora's Engineering College, Bhongir, Andhra Pradesh, India. He is currently pursuing M. Tech from Aurora's Engineering College, Bhongir, Andhra Pradesh, India. Contact him at **sastry_38@yahoo.com.**

K. Akshitha has been graduated with B. Tech in 2009 from Royal Institute of Technology and Science, Chevella, R. R. Dist, Andhra Pradesh, India. She is currently pursuing M. Tech from Aurora's Engineering College, Bhongir, Andhra Pradesh, India. Contact her at **Koluguri.87@gmail.com**

****

