Native Code Execution Control for Attack Mitigation on Android

Conference Paper · November 2013

DOI: 10.1145/2516760.2516765

Marcel Kulicke
Fraunhofer Institute for Applied and Integrat…
3 PUBLICATIONS **38** CITATIONS
SEE PROFILE

Julian Schütte
Fraunhofer Institute for Applied and Integrat…
43 PUBLICATIONS **136** CITATIONS
SEE PROFILE

Native Code Execution Control for Attack Mitigation on Android

Rafael Fedler, Marcel Kulicke, and Julian Schütte
Fraunhofer AISEC
Garching near Munich, Germany
{fedler,kulicke,schuette}@aisec.fraunhofer.de

ABSTRACT

Sophisticated malware targeting the Android mobile operating system increasingly utilizes local root exploits. These allow for the escalation of privileges and subsequent automatic, unnoticed, and permanent infection of a target device. Poor vendor patch policy leaves customer devices vulnerable for many months. All current local root exploits are exclusively implemented as native code and can be dynamically downloaded and run by any app. Hence, the lack of control mechanisms for the execution of native code poses a major threat to the security of Android devices. In this paper, we present different approaches to prevent local root exploits by means of gradually controlling native code execution. The proposed alterations to the Android operating system protect against all current local root exploits, while limiting the user experience as little as possible. Thus, the approaches we present help to avert automatic privilege escalation and to reduce exploitability and malware infection of Android devices.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Android; malware; mobile security; exploit mitigation

1. INTRODUCTION AND MOTIVATION

Malware targeting the Android platform increasingly relies on local root exploits. Examples include DroidKungFu [7], DroidDream [2], GingerMaster [6], and Bsmart (RootSmart) [9], among others. These exploits allow for the escalation of privileges and, as a consequence, for the circumvention of the Android operating system's security measures, ultimately giving unlimited access to the attacked device. Download and execution of such exploits underlies no restriction whatsoever. It facilitates the automatic and unnoticed infection of target devices with malware. Such malware is, when installed with root privileges, irremovable and has unlimited access to a device. In contrast, malware not incorporating root exploits lacks all of the above and requires user interaction for infection. Vulnerabilities attacked by root exploits often remain unpatched by device manufacturers for months or years.

SPSM'13, November 8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2491-5/13/11 ...$15.00.
http://dx.doi.org/10.1145/2516760.2516765.

As a countermeasure to the threats posed by malware using local root exploits, we propose several means to mitigate exploits. Our methodology focuses on ways to control the execution of native code. Therefore, our measures are able to control even dynamically downloaded and executed binaries, which was previously not enforceable in the Android framework. This also holds true if a device, once equipped with an Android version incorporating our proposed changes, remains unpatched: If execution of native code is well controlled, native code-based exploit execution is not possible.

The rest of this paper is organized as follows. Section 2 introduces the reader to the Android platform and its security model, whereas we focus on aspects relevant to root exploits based malware. Here, we will explain why malware using root exploits is the most dangerous threat to Android users. In Section 3, we will explain in detail our approach to control the executability of native code. The approach covers both code given as native standalone binaries, which is the form used exclusively by malware at the moment, and native libraries. The latter may be used to circumvent measures aiming to control the executability of standalone binaries. In Section 4, we review work related to ours. Section 5 concludes the paper.

2. BACKGROUND

Both the Android platform and Android malware differ significantly from traditional desktop and server operating systems and malware. The Android platform includes various concepts intended to secure app, user, and system resources from unauthorized access by attackers or malware. We will shortly revisit these aspects. Furthermore, we will cover typical malware methodology and discuss ways to execute native code on Android devices.

2.1 Android Platform and Security

The Android platform includes various security features, described in detail in [1]. Of those, we will introduce the most relevant measures for access control in the following.

File system.

Traditional operating systems assign all applications run by a user the same User ID (UID) and Group ID (GID).

Thus, all applications run by a user have access to all files by all other applications run by the same user, and the user's personal files. In contrast, Android assigns UIDs on a per-app basis [1]. As a result, any app may only access its own files, and no other files in the file system. Ultimately, this concept establishes a file system sandbox [1].

Using root exploits, however, an app may break out of this sandbox. It can then read or write any file stored on the device. On devices without hardware-based write protection, it can even use its root capabilities to remount the read-only system partition for write access and install sophisticated malware on this partition. For instance, the RootSmart malware uses this technique [9].

Permissions: System Resource Access Control.

The permissions concept allows to manage access to resources like peripherals, specific APIs, or data sources like the user's address book. Permissions define which resources an app can access, and therewith which functionality it can provide [4]. They are essential to a user's privacy and security, but also to the device's security: To access sensitive user data, malware requires the corresponding permissions. Some exploits also need access to crash logs or sockets to exploit certain vulnerabilities, which are protected via permissions (`READ_LOGS` and `INTERNET` permission, respectively).

Each permission is assigned a *protection level.* These levels further divide permissions depending on their hazardous potential and impose restrictions on which apps can use the respective permissions. There are four permission levels. Of those, *signature* and *signatureOrSystem* protect very high risk assets. They are usually not granted to any app installed by the user. However, *signatureOrSystem* clearance can be acquired by malware by means of root exploits. Root exploits can grant write access to the usually read-only system partition, which can be used to install malware there.

2.2 App Installation

Apps are most commonly obtained from app markets such as Google Play or websites. Users have to explicitly allow the installation of any new apps. The installation itself consists of the app's package file being placed in the device's code directory (in `/data/app/`) and the creation of a working directory for the app. Ownership for this working directory is set to the app's UID. However, using root access, an otherwise unprivileged app can write to the designated code directory, imitating the installation process.

The file system sandboxing mechanism prohibits alteration of package files of installed apps. Apps may, however, download arbitrary files from the Internet dynamically after installation. This way, they can install native code software and currently also execute it, without the system's or user's consent or knowledge.

2.3 Android Malware

Although the objectives of traditional desktop and server malware on the one hand, and Android malware on the other hand, are similar, the implementation of malicious functionality differs. Access to potentially sensitive user data is simplified by unified and easy to use interfaces. Privileged access to a device for permanent infection and other typical malware functionality like user input eavesdropping is significantly more difficult due to file system sandboxing and permissions. In the following, we will present the basics of Android malware, contrasting malware using root exploits and malware not using them.

2.3.1 Malware Methodology

Current root exploit based malware for Android has a largely similar modus operandi. The general course of action of such malware can be summarized by the examples of RootSmart [9, 11] and GingerMaster [6]. Both use the GingerBreak local root exploit for privilege escalation.

1. Initial propagation, e.g., disguised as a legitimate, low-permission app, or in a repackaged app
2. Download of a root exploit as native binary, in case it is not shipped with package file
3. Mark root exploit binary as executable by setting the executable bit with `chmod`
4. Execute root exploit and carry out payload, typically including permanent infection and escalation of permissions

Alternatively, some malware like GappII [8] also uses pre-supplied privilege escalation facilities (`su` utility) on phones rooted by their user instead of using exploits.

2.3.2 Impact of Root Exploits

While on desktop operating systems malware propagation and infection is primarily accomplished through vulnerabilities in end-user applications, these attack vectors cannot be used on Android devices to the same effect. Android apps are restricted in their access to the system through permissions, and to file system resources through file system sandboxing. Attackers will thus not obtain sufficiently privileged access for further steps such as permanent infection. Furthermore, Android apps are developed in Java, which is typically not easy to exploit due to static typing, bytecode verification and an automatic memory management for dynamically allocated objects [5]. Thus, Android apps are inconvenient targets for exploits and malware infection.

As a consequence, malware developers have to rely on users installing malware on their devices "deliberately". This is usually achieved by disguising malware as legitimate applications, or by hiding malware inside repackaged apps [15]. Thus, usual Android malware is completely dependent on unalert users and an autonomous infection without user interaction can usually not be achieved.

The exception, however, is malware utilizing local root exploits. If root exploits are used, malware can circumvent all measures of access control and security. File system access controls no longer apply when root privileges were acquired and any file may be read or altered. Given root privileges, malware can be installed without the user having to grant any permissions, as apps can just be placed in designated directories. Also, the read-only system partition can be remounted temporarily with write access in order to install malware which cannot be removed by the user, as done by RootSmart [9].

2.4 Vendor Patch Policy

The Android patch process is complex and involves many parties, whereas it turned out that the bottleneck is currently at the phone manufacturers [14]. Patches often reach customers only many months after their release by Google, or not at all. For this reason, measures against uncontrolled dynamic loading and execution of native code have already been demanded by other authors [15]. Such measures are this paper's contribution.

2.5 Native Code on Android

Android apps are usually developed in Java to make use of the Android API's app framework, and are emulated at runtime or translated from Java bytecode to native code prior to execution. Though rare, they may call native code and interact with it. Most commonly, this is done by games for efficient access to graphics hardware.

2.5.1 Execution of Native Code

To call and interact with native code, it needs to be executed by the system. Two ways to execute native code exist: The execution of standalone binaries, and the loading of native libraries.

Binaries can be executed from inside Android apps. For example, this can be achieved through methods provided by the classes `ProcessBuilder`, `Process` and `Runtime`, which are also part of Android's app framework.

Before a binary is executed, the operating system checks if the *executable bit* is set. If not, a binary is not executable and execution will be denied. To mark a binary as executable, the `chmod` utility, i.e. the respective `(f)chmod` system calls can be used.

Unlike binaries, **libraries** do not need to be marked executable before they can be loaded and run. Java Native Interface-compatible libraries can be loaded and executed via the `System` class, which is accessible from Android apps.

Apart from interfaces specifically provided for loading and execution of native code, machine code may also be directly loaded into memory and executed. To this end, the standard library's `mmap()` or the respective system call may be executed with the `PROT_EXEC` option.

2.5.2 Deployment of Native Code

Binaries or libraries to be executed can be supplied to Android apps through three different means: (1) They are preinstalled on an Android device. This applies for standard utility binaries like `rm`, `ls` and `chmod`, among others. Various native libraries are also contained in a standard Android installation, e.g., for graphics hardware access. (2) They are bundled with an Android app's package file. In this case, they are installed alongside with the Android app upon its installation. (3) They are downloaded at runtime by an Android app, which has a negative impact on Google Bouncer's ability to detect malicious code. Effectively, this option is a way of adding software after the actual installation which had been confirmed by the user. This circumvents the authority of the user and of the package manager over installed software.

2.5.3 Relevance of Native Code for Apps

Only 4.52% of all apps make use of native code at all. The vast majority of those (4.24% of all apps) place their native code in the standard directory for presupplied libraries, while 0.28% place native code in non-standard directories [16]. It can be argued that the aforementioned majority primarily uses libraries, because of the directory used. Furthermore, due to better integration with Java apps, binaries are of less use to legitimate apps which use native code mainly for performance reasons.

As can be seen, more than 95% of Android apps will not be affected at all by measures against uncontrolled loading and execution of native code. Measures specifically geared towards binaries would, with a very high probability, affect even far less legitimate apps.

2.5.4 Relevance for Malware and Exploits

Currently, all local root exploits are implemented as native code – or, more specifically, as binaries – due to various reasons. For one, native code is more flexible when manipulating memory or when operating close to hardware. Some exploits rely on excessive forking and thus spawning new processes quickly to exhaust the maximum process number, others deploy return-oriented programming techniques or overwrite other processes' memory. Also, header files exist for different data structures used by the Android operating system, which can conveniently be used in native code implementations. For the aforementioned reasons, native code is highly convenient for reliable and effective exploit development. Some exploits may even be unable to work completely when lacking features which a non-native implementation will not permit.

3. APPROACH

As shown in the previous sections, native code can be downloaded and executed at runtime without any limitations or control. This allows any app to install additional software without user or system consent. All known root exploits for Android are developed as native code standalone binaries. However, less than 5% of all apps use native code, and almost none of them use standalone binaries. We conclude that, as the ability to execute native code is primarily used by malware and as it is the main factor making Android malware a considerable threat, it ought to be controlled and limited.

In the following, we present measures to control the execution of native code on the Android platform, both for standalone binaries and for libraries. Although currently known exploits rely on standalone binaries, future exploits may be switch to libraries to circumvent measures solely targeted at binaries. However, some exploits may even be impossible to implement as libraries. The measure discussed herein are suited to successfully prevent dynamic downloading and execution of all current exploits by an app.

3.1 Binary Files

Binary files need to be marked executable via the `chmod` utility or system call before they can be executed. However, it should not be possible for any app on a device to mark standalone binary files as executable, as this opens up the possibility to download any file from the Internet and execute it. Thus, to prevent the execution of arbitrary files, the capability to set the executable bit has to be controlled. It has to be always kept in mind that binaries are even far less used than native libraries.

3.1.1 Executable Bit Control

To prohibit the uncontrolled manipulation of the executable bit of arbitrary files, limitations have to be introduced into the `chmod` family of system calls (cf. Figure 1, Label 6). This will prevent Android apps from calling `(f)chmod` sys-

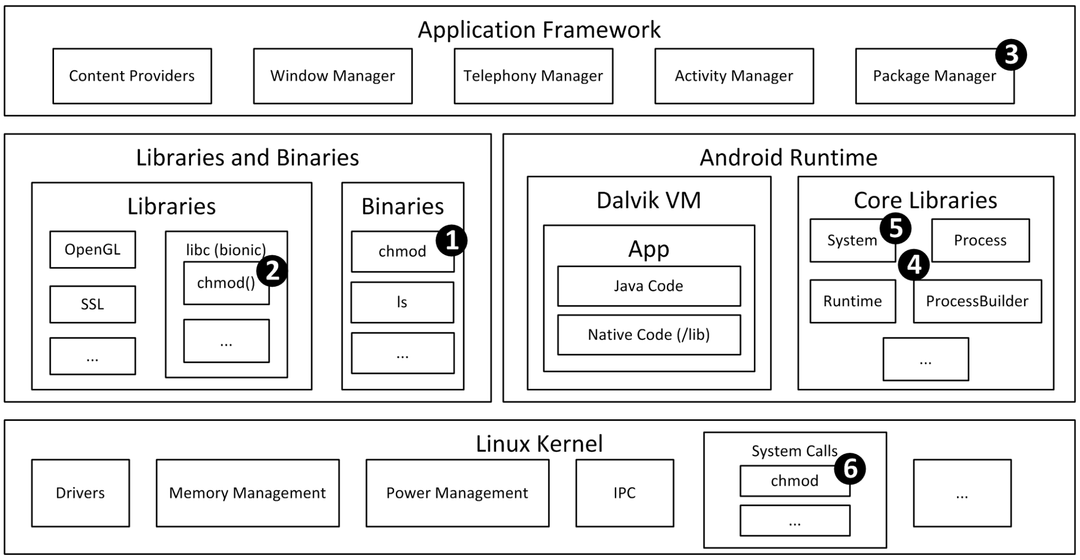

Figure 1: Integration of native code execution control into the Android architecture

tem calls or running the presupplied `chmod` binary, e.g., via methods provided by `ProcessBuilder` or `Runtime` classes (Label 4), to set the executable bit for files. As there is no legitimate reason to set the executable bit for files, this should be prohibited completely. For directories, however, the executable bit has a different purpose. Thus, the `chmod` calls should include checks, for example via `fstat` calls, to check if the target of an operation to set the executable bit is a directory or a file. If it is a directory, setting of the executable bit can be allowed. If not, it must be denied.

By introducing these checks into `(f)chmod` at kernel level, unauthorized usage of the `chmod` utility (Label 1) and libc method stubs (Label 2) are also controlled, as they build on the aforementioned system calls. The checks need to be at kernel level to also cover code directly invoking the system calls. These checks will prohibit Android apps from (a) running the `chmod` utility to mark binaries as executable, (b) from using libraries to achieve the same goal, (c) directly invoking related system calls, or (d) using ROP techniques to the same end. Preinstalled binaries will not be affected, as their executable bit can be set by direct manipulation of the file system image during device development and manufacturing.

3.1.2 UID/GID-Based Exceptions

Developers and users of the Android operating system may still wish to be able to set the executable bit for binaries under certain circumstances. For example, the openness for modification of the Android platform would be reduced if users were unable to upload certain utilities to their devices which are given as binaries. It is feasible to retain the possibility of setting the executable bit for binaries which come bundled with an app's package file which is downloaded and installed from an app market such as Google Play.

These aims can be achieved through user ID (UID) or group ID (GID) checks. The Android operating system has predefined UIDs and GIDs for some privileged functionality and for access control means. For example, the user for USB shell access has a predefined user and group ID of 2000. As on any Linux-based operating system, the root user has a UID of 0. If `chmod` (Labels 1/2/6) not only checks if the target of a operation to set the executable bit is a file or a directory, but also if the operation is carried out by a specific UID, it can allow this operation for files if the UID matches a certain value, e.g., 0 or 2000. Alternatively, it may also be checked whether the requesting process is member of a certain GID. This allows users with USB access to modify their devices, while leaving restrictions for usual apps intact, as they are assigned different, unique UIDs.

Also, the package manager (Label 3) can be assigned a predefined user ID via the SUID bit or be added to the GID allowing for manipulation of the executable bit. This would allow for app package files containing binary files which can exclusively be marked executable only by the package manager. Such files could be defined in an app's manifest file. Consequently, developers would still be able to provide binaries at installation time. Download of native code by apps at runtime would be prevented.

Instead of the UID approach, the permission to set the executable bit can also be associated with one GID. This would allow for more flexible permissions.

The aforementioned possibilities to still set the executable bit are optional. They allow users to modify their devices or developers to provide binaries in their apps' package files, while making it impossible for Android apps to download and execute binaries and thus exploits uncontrollably. It should be evaluated if these possibilities ought to be retained, especially in the case of the package manager, as native binaries are hardly used at all by legitimate apps.

3.1.3 Permission-Based Control

Instead of controlling executability of binaries at the file system level, the Android operating system can control the ability of Android apps to run binaries. This can be achieved by permissions securing the affected classes, such as `Process Builder` and `Runtime` (Label 4). That way, the user can decide to allow or prohibit the execution of native binaries to specific apps. Given the fact that binary execution is a hardly used feature, it would have to be decided if these permissions are assigned a *protection level* of *dangerous* or *signatureOrSystem*. In the latter case, only system apps preinstalled on a device are able to run binaries, further increasing device security. The advantage of this measure is that users are still able to allow single apps to execute native binaries. A permission warning text informing the user of the rareness of legitimate usage scenarios of binaries should be displayed.

3.2 Libraries and JNI

Android apps can not only run standalone native binaries via `ProcessBuilder` or `Runtime` class methods, they can also load native libraries at runtime via the `System` class's `load` and `loadLibrary` methods (Label 5). Libraries, in contrast to binaries, do not require the executable bit to be set in order to be executable. Thus, the aforementioned measures targeted at binaries fail.

3.2.1 Loader- and File System-Based

To extend the measures devised for binaries onto libraries, the executable bit can be used. It could be required by the Android API implementation of Java's `System.load()` and `System.loadLibrary()`. These methods can be changed to check files to be loaded for the executable bit. If the executable bit is not set, loading of the library will be denied by the aforementioned `System` methods.

If implemented, native libraries preinstalled on Android devices need to be marked executable as well. The measures designed for binaries would apply to libraries too, eliminating the need for specific measures.

3.2.2 Permission-Based

The `System` class can be secured via a permission. This would place the power of decision on an app's capability of executing native libraries into the hands of the user. The user can then decide at installation time if an app should be allowed to execute native libraries. A suitable warning text which is displayed for each requested permission can inform the user of potential risks, and that most commonly such a permission is only required by games or other performance-critical apps.

3.2.3 Path-Based Restriction

By restricting `System.load()` and `System.loadLibrary()` to the default path for preinstalled libraries on an Android device's system partition, no native library added after manufacturing would any longer be executable. This constrains developers to the usage of presupplied native libraries, though in a highly inflexible way.

3.3 Complete Prohibition

As a very drastic measure, the execution of native code not preinstalled on an Android device can be prohibited completely. By removing the possibility to set the executable bit from the `(f)chmod` system call, and by removing the respective methods from the `System`, `Runtime`, `Process` and `ProcessBuilder` classes, the capability of executing native code can be removed. This, however, would eliminate the compatibility with many current apps and the openness of the Android platform. Consequently, we do not consider this a realistic option.

3.4 Summary

In this section, we proposed several measures for controlling native code on the Android platform. The following list shall provide a better overview of the various options available:

Binaries

- Executable bit control: Checks in `(f)chmod` system call
  - Check if target of a "chmod +x" operation is a directory or a file; if file: prohibit
  - Possible UID-based exceptions for UIDs 0 (root) and 2000 (USB shell access) to retain Android's openness to modification
  - Package manager may be given a fixed UID via SUID file system bit and can be added to the UID exception list, so that developers can still provide binaries in an app's package file which can be marked executable by the package manager
  - For more flexible assignment of the permission to set the executable bit, a GID can be introduced instead of a UID
- Permission-based: introduce permissions for `Process`, `ProcessBuilder` and `Runtime` classes

Libraries

- Extend executable bit to libraries; introduce checks for executable bit into `System.load()` and `loadLibrary()`
- Permission-based: introduce permission for `System` class
- Restrict `System.load()` and `System.loadLibrary()` to the operating system's default path(s) for libraries

Binaries & Libraries

- Removal of the option to set executable bit via `chmod` and of relevant methods from the `System`, `Runtime`, `Process` and `ProcessBuilder` classes

3.5 Discussion

The devised methods provide different levels of protection against uncontrolled execution of native code. The permission-based options would only require minor alterations to existing apps using native binaries or libraries. Our permission approach shifts control to users, who then have to allow apps at installation time to execute native components. A suitable warning message can inform users of the potential risks and common usage scenarios like games. On the downside, malware authors could apply the repackaging model to games, and hide malicious native code specifically in games.

But even more drastic measures, like the proposed changes to the `(f)chmod` system call (cf. Figure 1, Label 6), would affect only very few legitimate apps, as less than 5% of all apps incorporate native code [16]. With our approach of UID/GID filtering, the openness to modification is kept in tact. If the package manager (Label 3) is assigned a fixed UID through the SUID bit, native code can still be installed from package files at app installation time. Thus, we conclude that our measures presented in this paper are effective at controlling the executability of native code, without major negative impact on existing legitimate apps.

We acknowledge that our approach to control the execution of native code does not protect against exploits targeting flaws in the Dalvik VM. Also, the Dalvik VM, unlike

Oracle's Java VM, was not designed with focus on security mechanisms. Furthermore, we did neither consider protection against remote exploits of vulnerable communication stacks such as GSM, NFC, or USB, nor attacks at the bootloading process.

As mentioned in Section 2.5.1, `mmap()` may be used to directly load executable machine code. However, it is not callable from within non-native code. Thus, in order for this interface to be abused, an app needs to be able to execute native code in the first place. If so, however, such an app may use `mmap()` to extend its otherwise limited capabilities.

A possible result of the integration of our measures could be a shift of exploit authors towards Java, against which our approach would not prove effective. While the Java language is not as qualified for exploit development and thus currently hardly used to that end, it would still be possible, though more difficult.

4. RELATED WORK

To the best of our knowledge, our approach is the first to focus specifically on controlling native code execution in Android. However, work concerned with extending Android's permission system and integrating additional security mechanisms is related to ours and will be reviewed here.

Mandatory access control systems such as SEAndroid [10] and Tomoyo [13] have been used in a practical approach to domain isolation. They prevent unauthorized access to other apps' resources [3] and also exploit execution. However, they have two drawbacks: First, as the enforcement is implemented at kernel level only, applications trying to invoke forbidden system calls will rather crash or end up in unpredictable states. Our approach is implemented at the middleware layer and thus allows to properly inform the user about blocked executions. Second, MAC-based approaches are heavyweight as they require modifications of the kernel and the specification of appropriate policies, which is not trivial in practice.

Other publications also identify native code as a potential security risk on the Android platform, and present the possibility to remove native code execution capabilities from the Dalvik VM [12]. We achieve the same goal, but keep general native code execution capabilities in tact.

5. CONCLUSION

Currently, any app with Internet access can download and execute any native code from the Internet, including root exploits. Our approach provides a solution to this problem. It may not provide absolute security, but it will raise the hurdles of successful exploitation to a point where every current native code-based exploit would fail. It decreases the exploitability of Android-powered devices, as rogue apps can no longer download and run arbitrary native files from the Internet. These results come with nearly no negative impact on current legitimate apps, as more than 19 out of 20 apps use no native code at all [16], and standalone binary files are used even less. Our approach is much less intrusive than SEAndroid, and does not require policy rule sets to work.

6. REFERENCES

[1] Android Open Source Project. Android Security Overview. http://source.android.com/tech/security/ (18.02.2013).

[2] M. Balanza, K. Alintanahin, O. Abendan, J. Dizon, and B. Caraig. DroidDreamLight lurks behind legitimate Android apps. In *6th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 73 –78, oct. 2011.

[3] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 51–62, New York, NY, USA, 2011. ACM.

[4] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *Security & Privacy, IEEE*, 7(1):50–57, jan.-feb. 2009.

[5] J. C. Foster and M. Price. *Sockets, Shellcode, Porting & Coding*. Syngress Publishing, 2005.

[6] X. Jiang. GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.(Gingerbread), August 18, 2011. http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/ (18.02.2013).

[7] X. Jiang. New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets, June 23, 2011. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu .html (18.02.2013).

[8] X. Jiang. New GappII Trojan Found in Alternative Android Markets, April 27, 2012. http://www.csc.ncsu.edu/faculty/jiang/GappII/ (18.02.2013).

[9] X. Jiang. New RootSmart Android Malware Utilizes the GingerBreak Root Exploit, February 3, 2012. http://www.csc.ncsu.edu/faculty/jiang/RootSmart/ (18.02.2013).

[10] SELinux Project. SEAndroid. http://selinuxproject.org/page/SEAndroid (18.02.2013).

[11] M. Spreitzenbarth and F. Freiling. Android Malware on the Rise. Technical report, University of Erlangen, Dept. of Computer Science, April 2012. Tech. Rep. CS-2012-04.

[12] A. Stavrou, J. Voas, T. Karygiannis, and S. Quirolgico. Building security into off-the-shelf smartphones. *Computer*, 45(2):82 –84, Feb. 2012.

[13] G. L. Tona. TOMOYO Linux on Android, May 2009. http://sourceforge.jp/projects/tomoyo/docs/Part2_CELF_Android.pdf (18.02.2013).

[14] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: a survey of current android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, pages 81–90, Berkeley, CA, USA, 2011. USENIX Association.

[15] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy 2012*, pages 95 – 109, May 2012.

[16] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium*, February 2012.