Controlled Android Application Execution for the IoT Infrastructure

Michael N. Johnstone, Zubair Baig(✉), Peter Hannay, Clinton Carpene, and Malik Feroze

School of Science and Security Research Institute, Edith Cowan University, Perth 6027, Australia
z.baig@ecu.edu.au

Abstract. Android malware has grown in exponential proportions in recent times. Smartphone operating systems such as Android are being used to interface with and manage various IoT systems, such as building management and home automation systems. In such a hostile environment the ability to test and confirm device health claims is important to preserve confidentiality of user data. This paper describes a study to determine whether an Android device could be secured to prevent malware from executing in parallel with trusted applications. The research also sought to determine whether the system image could be protected from unauthorised modifications. A prototype scheme for meeting the above requirements was developed and tested. It was observed that the prototype succeeded in preventing unauthorised modification to the system image of the test device. However, the prototype failed to prevent unauthorised IPC calls when in single process mode.

Keywords: Static malware analysis · Dynamic malware analysis · Android platform

1 Introduction

The evolution of distributed computing, most recently evinced by the Internet-of-Things (IoT) is considered to be an essential driver of contemporary information technology. The use of embedded microprocessors within a plethora of heterogeneous device types has been facilitated by substantial increases in available network bandwidth. Consequently, household devices such as televisions, refrigerators and washing machines are all embedded with computers that monitor device activity, log events, and transmit useful and pre-defined information to a centralised server for further processing and/or action. Mobile devices such as smartphones and smartbooks can serve as decentralised data collection and processing points within an IoT infrastructure. The privacy and security requirements within an IoT context will vary depending upon the IoT devices in use. For instance, a mobile phone connected through a 3G/4G communication channel to the Internet will demand a set of security controls based on large keys and

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2016
B. Mandler et al. (Eds.) IoT 360° 2015, Part I, LNICST 169, pp. 16–26, 2016.
DOI: 10.1007/978-3-319-47063-4_2

a requirement for extensive information processing. Alternatively, a resource-constrained RFID tag may likely require a less resource demanding secret-key verifier based on smaller key lengths and efficient algorithms.

Whilst mobile devices provide an ideal central controller for household IoT devices as seen by the expansion of Home Automation Networks, the underlying operating system poses serious security concerns. The popularity of mobile devices has increased exponentially, and applications installed and executed by end-users routinely process and transmit sensitive data (e.g., banking and online shopping applications). Private user data can be compromised by an adversary through simple social engineering efforts as well as through sophisticated step-wise attacks that may involve the installation of malware onto a mobile device for subsequent invocation. The threat landscape for mobile platforms has been increased many-fold because popular end-user applications can be installed at the click of a button (thus highlighting the tension between competing non-negotiable requirements, namely security and ease of use). An example is an augmented version of the ZeuS banking Trojan [1] which appears as a legitimate application that upon execution transmits user credentials to a remote (attacker) machine. Emerging threats against mobile devices pose an even greater challenge for security architects of the IoT infrastructure. End-users do not identify the same security issues with mobile devices (compared to desktop devices) as noted in a survey by Valli et al. [2]. The need for controlled application execution on mobile devices, within a trusted context, cannot be understated. In this paper, we document and examine a prototype of a controlled mobile device trust platform for Android devices that ensures a secure application execution environment.

The remainder of the paper is organised as follows. Section 2 discusses related work done in the area of mobile device security. In Sect. 3, we present the prototype implementation of a controlled execution environment for Android applications. Section 4 discusses the tests that were performed against the Controlled Access Prototype. Section 5 describes the results of the study. Finally, in Sect. 6, we conclude and provide suggestions for further research.

2 Related Work

As pointed out by Löhr et al. [3], secure boot is a basic trusted computing concept. Löhr et al. observe that secure boot requirements for mobile devices have been collected in the Open Mobile Terminal Platform recommendations. The problem of secure boot is bound by three constraints or properties, described by Löhr et al. as: (1) The integrity of software loaded on the system must be preserved, otherwise malicious software could run without being detected; (2) The system should always boot to a defined secure state (or fail to boot at all), else attackers could violate security by forcing the system into an insecure state; and (3) Modifications of the operating system or application binaries must still be allowed, otherwise, software updates would be impossible.

Probably the first attempt at secure boot on an x86 platform was the AEGIS system [4] which used digital signatures as integrity checks and allowed recovery

using a trusted repository. This concept still exists in Windows 8, which implements secure boot via a signature check for each item of boot software, although this has already been compromised (see [5]). Mobile devices, by their nature, may not have access to a trusted repository.

Kostiainen et al. [6] discuss several security issues that exist on several mobile device operating systems (including Android), but their analysis appears focussed on post-boot issues such as access control and permission granularity. Such issues would be important, however, if a secure boot failed i.e. it booted a device into an insecure state. Shabtai et al. [7] assert that Android devices are well-guarded in their normal state. Shabtai et al. conducted a risk analysis of the Android platform and concluded that corrupting or modifying private content in various forms, was a minor impact risk with an unlikely or possible likelihood of occurrence. We content that this is an optimistic analysis, especially in the context of Gostev's [8] comment that "two years of smartphone virus evolution are equivalent to twenty years of work in computer viruses".

Shabtai et al. [7] also notes that one of the security mechanisms of Android requires that "each application runs in its own virtual machine". This provides a measure of safety, but King et al. [9] evaluated virtual machine-based rootkits, the result being that they were able to subvert Linux-based systems with a proof-of-concept virtual machine-based rootkit. More specifically, Vidas et al. [10] provide a survey of current Android attacks.

Dietrich and Winter [11] highlight the need for a secure boot process on mobile devices and state that, whilst the Mobile Phones Working Group (MPWG) have outlined a secure boot process, the detail has been left to individual manufacturers to implement, thus giving manufacturers some flexibility. This flexibility, however, can lead to security issues because of the multiplicity of approaches that fit the outline provided by the MPWG.

Hendricks and van Doorn [12] contend that existing secure boot standards are not enough and that all devices should be checked, not just those attached to the CPU. Their claim was based on the assertion that modern computers consisted of semi-autonomous sub-systems containing field-upgradable firmware. Despite this assertion being over a decade old, it bears some similarity to devices that populate the Internet of Things, latterly the Internet of Everything.

3 Controlled Access Prototype Implementation

By design, the Android application framework includes various components to encourage functionality reuse. For example if an application that wishes to make a web request, it could request that another application make the request on its behalf. This reuse is achieved through application programming interfaces (APIs) such as activities, services, broadcast receivers and content providers [13]. Each of these features enable inter-process communication in various ways. Some of these are focussed on directly addressed communication and others are simply ways to request a specific item of functionality, allowing any other application to serve these requests.

A number of vulnerabilities have been identified with these APIs. These include SQL injection attacks and information theft [14]. In order to mitigate these issues it has been left to the developers of Android applications to implement defences in order to protect their application from any others that may be running. As new attacks are developed it may be that even the most carefully developed application becomes susceptible to these attacks. Of course there is still the threat posed by vulnerabilities to the Android operating system itself, which may not be able to be mitigated by the application developer. Existing approaches to address this issue have relied on implementing security layers on top of the existing Android framework, however this doesn't solve the underlying issue for the potential of interprocess communication based attacks or attacks based on other mechanisms [15]. Attacks have been implemented which do not rely on inter-process communication but instead through shared memory, showing the potential for non-inter-process communication driven attacks [16].

The proposed prototype aims to address this problem by preventing simultaneous application execution outright, via modification of the underlying Android operating system and making use of integrity checking measures to ensure that the operating system itself is not compromised.

The prototype developed aims to address two key requirements of the Android mobile platform:

- Device state at time of execution: The state of the device can always be verified to not have malicious applications running in parallel with the current application.
- Device base state: The mobile device should have protection mechanisms against unauthorised modification to the base system.

A two-part solution is proposed to meet the requirements. The first part is to limit the number of running applications to one, preventing malware from running in parallel with the application. Second, to use secure boot which prevents operating system modifications. This process results in a situation in which malware cannot be injected into the system image or cause other unauthorised changes to the platform.

Limiting the number of running applications to one, ensures that only one application is running at any point in time. As a result the operating system will force any applications other than the foreground application, to terminate (excluding core system processes).

The verified boot feature, which was introduced in Android 4.4, aims to assure the device state. This process uses an optional kernel feature called "device-mapper-verity (dm-verity)". The dm-verity feature provides a means for the device to conduct integrity checking of block devices. Consequently, dm-verity can be used to prevent rootkits, and other unauthorised modifications to the system image.

Any process on the Android platform that runs with root privileges can bypass detection by anti-malware applications or indeed the operating system itself. The software has this capability as it is running with privileges higher than

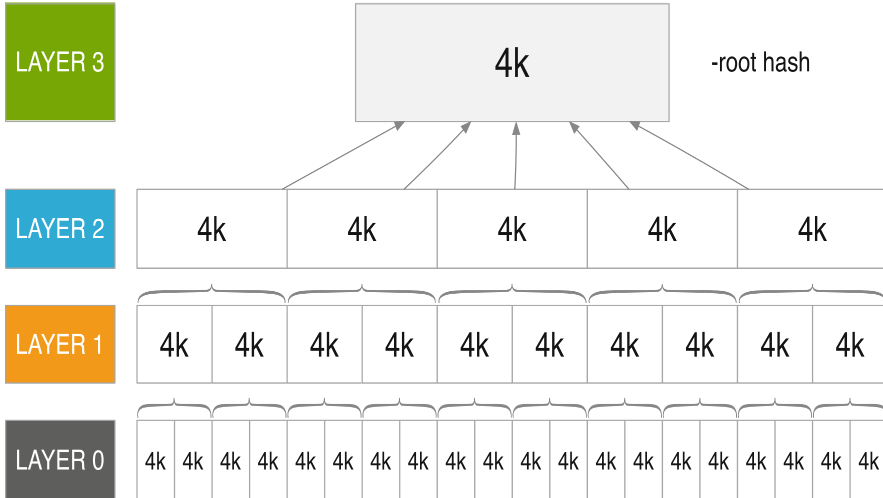


Fig. 1. Cryptographic hash tree [17].

or at the same level as the software intended to detect malware. These privileges essentially enable the software to misrepresent itself or conceal itself altogether.

Block devices form the underlying storage layer for Android systems. They can be examined using dm-verity to ensure that the device matches an expected configuration. Configuration validation is achieved through the use of a cryptographic hash tree. Every block (which is typically of 4k block size) is hashed using SHA-256 and the result stored in the hash tree. These hashes are aggregated through each layer (as depicted in Fig. 1) until Layer 3, where the root hash is computed. In practical terms this means that only the root hash needs to be verified to confirm integrity of the entire hash tree. In order to modify any of the blocks, the cryptographic hashing function (SHA-256 in this case) would need to be broken for identically sized inputs. At access time, each block is verified, which reduces overhead at boot time.

This solution ensures that a device meets the health requirements at any given time. Using Secure Boot ensures that the operating system has not been modified in any way. If the system image is changed in anyway, the device will fail to boot. If the device boots successfully, it is a guarantee that the device did not allow malware to load at boot time. Secure Boot's task is finished once the device has booted. As such, need is established for post boot mechanisms to allow execution of software without the risk of malware running in parallel, thus compromising the integrity of the whole system. For this purpose, the application limiting functionality is put in place. Limiting the number of parallel applications to one guarantees that no malware can run in the background while the target application is being executed.

To limit the number of parallel applications, we need to modify the Android source code. In order to achieve this functionality, there are two files that

need to be changed, namely, `DevelopmentSettings.java` and `ActivityManagerNative.java`. `DevelopmentSettings.java` is located at: `/packages/apps/Settings/src/com/android/settings/`. The changes required in this file are as below:

```
Line 1251: mAppProcessLimit.setValueIndex(i);
    to: mAppProcessLimit.setValueIndex(1);
Line 1251: mAppProcessLimit.setValueIndex(i);
    to: mAppProcessLimit.setValueIndex(1);
Line 1252: mAppProcessLimit.setSummary(
        mAppProcessLimit.getEntries()[i]);
    to: mAppProcessLimit.setSummary(
        mAppProcessLimit.getEntries()[1]);
Line 1256: mAppProcessLimit.setValueIndex(0);
    to: mAppProcessLimit.setValueIndex(1);
Line 1257: mAppProcessLimit.setSummary(
   mAppProcessLimit.getEntries()[0]);
    to: mAppProcessLimit.setSummary(
   mAppProcessLimit.getEntries()[1]);
Line 1264: int limit = newValue !=
       null ? Integer.parseInt(newValue.toString()) : -1;
to: int limit = newValue != null ? 0 : -1;
```

ActivityManagerNative.java is located at `/frameworks/base/core/java/android/app/`. The changes required in this file are as below.

```
Line 1163 int max=data.readInt();
    to: int max = 0;
Line 1171 int limit = getProcessLimit();
    to: int limit = 0;
```

Once this is done, we can build a system image from this modified Android source code. This system image is then used to implement dm-verity for ensuring secure boot. The steps are as follows:

1. Create system and boot image,
2. Create hash trees for both images,
3. Generate dm-verity tables,
4. Sign generated tables,
5. Concatenate dm-verity table and signature block to generate verity metadata, and
6. Concatenate tables, signature block and metadata.

The hash tree is at the core of the security control that enables secure boot, and is implemented by the dm-verity kernel feature. Every block (in this case 4k in size) in the block device is cryptographically hashed with SHA-256. These hashes form layer 0 of the hash tree. Next, the SHA-256 hashes of layer 0 are

concatenated into 4k blocks, and again each block is hashed with SHA-256. This forms layer 1 of the hash tree. This process is continued until the resulting layer n can be condensed into a single 4k block. Finally, the hashes at layer n are aggregated and hashed using SHA-256 to form a single, root hash value that serves as the integrity checker for the entire filesystem. If there is a block in a layer that is not naturally filled to the block size, the block is padded out with zeroes.

Subsequent to the creation of the root hash and salt, the dm-verity table is created and signed. The table itself is comprised of a reference to the block device, the block size, salt and root hash value. The table is signed with an RSA key of length 2048 bit ([17]). Next the verity metadata block is generated through the concatenation of the signature and the dm-verity table.

4 Controlled Access Prototype Evaluation Process

The device state and malware claims are both evaluated individually. Evaluation of the malware claim is trivial. Once the operating system has booted the settings menu can be accessed and the *Background* process limit setting examined. It should be set to "no background processes". If an attempt to change this value is made it should not take effect, on re-examining of this setting "no background processes" should again be observed. Subsequently, we can run multiple applications and to test if any of them are running in parallel, this can be accomplished through examining the running processes list under developer tools. It can be seen that only the last process launched is running, as such it can be confirmed that previous processes are killed.

To evaluate the secure boot implementation, the system image must be modified in such a way that it no longer conforms to those calculated in the initial hash tree. This task can be accomplished in various ways, from the flashing of a completely different system image, through to modification of a single byte within the system partition and/or boot partition currently located on the device. It should be noted that the entire image is not verified on boot, instead each block is verified on access. In this way boot times are reduced while the system and boot images are effectively secured. As such, if alteration of a single byte is selected it may be useful to target a byte within a particular system application, so that this application can be launched and the security feature reliably triggered.

It should be noted that if the boot image is modified the device will fail to boot and the device will be rendered near-permanently inoperable, this is by design. Recovery from this state during testing required the use of a non-public exploit. As such, it is suggested that during third party validation the system image be altered rather than the boot partition, in order to preserve device functionality.

5 Results

The Controlled Access Prototype was subjected to a number of tests to evaluate its suitability. These tests were described in Sect. 4. The results of the tests

Table 1. Details of alterations to system and boot images with details of event to trigger read and associated result

Alteration made	Triggering event	Observed result
Replace entire system image (zeros)	Power on device	Fail to boot
Replace entire system image (random)	Power on device	Fail to boot
Change single byte of system image (within application)	Launch application	Displays security warning
Change single byte of system image (at random)	Read random byte	Displays security warning
Replace entire boot image (zeros)	Power on device	Fail to boot
Replace entire boot image (random)	Power on device	Fail to boot
Change single byte of boot image (within boot code)	Power on device	Fail to boot

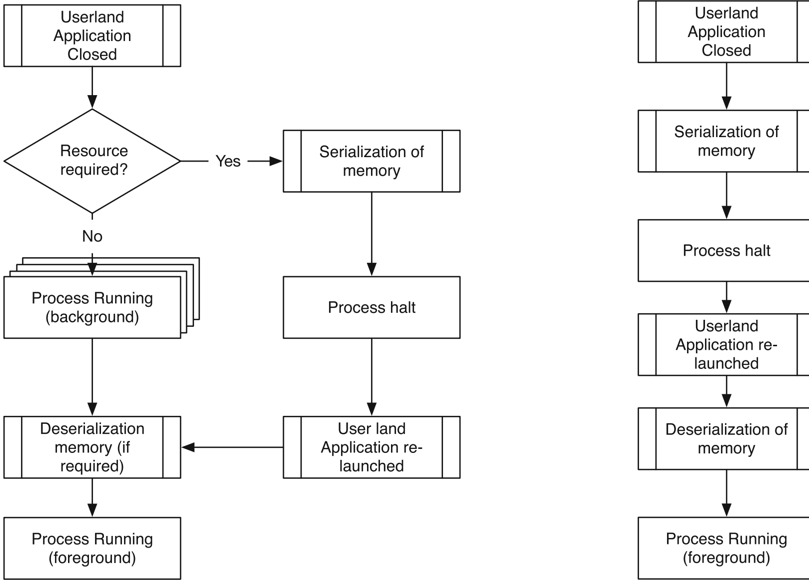
against the secure boot process are included in Table 1. It was observed that under almost all experimental conditions, the resulting image failed to boot once modified. This aligns with the expectations of the research. It is interesting to note that modifying a single byte of the system image, either at random, or from within an application, only caused an error to be displayed. The system otherwise functioned as normal.

The execution state testing involved using applications in the Android OS' single process mode. The outcomes of these tests are included below in Table 2. For the test to pass, any IPC calls from the malicious application could not be completed successfully (i.e. the calls were prevented).

It can be seen from the results that the IPC calls were able to successfully complete in all of the tested scenarios, indicating that the tests unanimously failed. The reason this approach does not work can be attributed to the architecture that Android uses to manage processes for applications. The application closing does not inherently mean that the process has been terminated (depicted in Fig. 2). Even in the single process mode (depicted in Fig. 2(b)), the application's memory is serialised upon termination and can be resumed by deserialisation upon relaunch. This means that although the malware may be halted temporarily during application switching, it will resume its operations when the context of execution changes back to the host application. The processing models are demonstrated in Fig. 2.

Table 2. Details of execution state testing with result, during prior to each test the evaluation application is launched

Test performed	Triggering event	Observed result
Examine process list	Launch malicious app	Evaluation app not running
Examine process list	Launch malicious service	Evaluation app not running
Attempt IPC call	Activate IPC from malicious app to evaluation app	IPC call succeeds (failed test)
Attempt activity call	Activate activity from malicious app to evaluation app	activity call succeeds (failed test)
Attempt broadcast call	Activate broadcast from malicious app to evaluation app	Broadcast received (failed test)



(a) Android's default multi-process model

(b) User-configurable single process model

Fig. 2. An abstract model demonstrating the differences between the operations of the Android OS' default multi-processing model, and the developer option enabled single process model.

6 Conclusions

This research set out with the aim of determining whether an Android device could be configured to prevent malware from executing in parallel with trusted applications, and whether the system image could be protected from unauthorised modifications. As Android devices are increasingly being used as controllers in IoT infrastructure, the mitigation and prevention of malware on these platforms is critical. We presented a prototype scheme for meeting the above requirements in the proposed *Controlled Access Prototype*. This prototype succeeded in preventing unauthorised modification to the system image through the implementation of the dm-verity kernel feature. However, the prototype failed to prevent unauthorised IPC calls when in single process mode. This research provides an opportunity for future work. We suggest modification of the Android sandbox to prevent IPC from running blacklisted or non-whitelisted applications. This would effectively enable an OS-level firewall for Android devices.

Acknowledgments. This work has been partially funded by the European Commission via grant agreement no. 611659 for the AU2EU FP7 project.

References

1. Barroso, D.: 21sec Security Blog: ZeuS Mitmo: Man-in-the-mobile (III) (2015). http://securityblog.s21sec.com/2010/09/zeus-mitmo-man-in-mobile-iii.html
2. Valli, C., Martinus, I., Johnstone, M.: Small to medium enterprise cyber security awareness: an initial survey of Western Australian business. In: Proceedings of the 2014 International Conference on Security and Management, pp. 71–75 (2014)
3. Lohr, H., Sadeghi, A., Winandy, M.: Patterns for secure boot and secure storage in computer systems. In: Proceedings of the 10th International Conference on Availability, Reliability, and Security, pp. 569–573 (2010)
4. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 65–71. IEEE Press, New York (1997)
5. Bulygin, Y., Furtak, A., Bazhaniuk, O.: A Tale of one software bypass of Windows 8 secure boot. In: Proceedings of Black Hat, USA (2013)
6. Kostiainen, K., Reshetova, E., Ekberg, J., Asokan, N.: Old, new, borrowed, blue: a perspective on the evolution of mobile platform security architectures. In: Proceedings of the First ACM Conference on Data and Application Security and Privacy (CODASPY 2011), pp. 13–24. ACM, New York (2011)
7. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google Android: a comprehensive security assessment. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 35–44. IEEE Press, New York (2010)
8. Gostev, A.: Mobile malware evolution: an overview (2001). http://www.viruslist.com/en/analysis?pubid=200119916
9. King, S., Chen, P., Wang, Y., Verbowski, C., Wang, H., Lorch, J.: SubVirt: implementing malware with virtual machines. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 314–327. IEEE Press, New York (2006)
10. Vidas, T., Votipka, D., Christin, N.: All your droid are belong to us: a survey of current Android attacks. In: Proceedings of the 5th USENIX Conference on Offensive Technologies, p. 10. USENIX Association, Berkeley, CA, USA (2011)

11. Dietrich, K., Winter, J.: Secure boot revisited. In: Proceedings of the International Conference for Young Computer Scientists, pp. 2360–2365 (2008)
12. Hendricks, J., van Doorn, L.: Secure bootstrap is not enough: shoring up the trusted computing base. In: Proceedings of the 11th Workshop on ACM SIGOPS European Workshop. ACM, New York (2004). Article 11
13. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, Maryland, USA (2011)
14. Agrawal, A.: Android application security part 3-Android application fundamentals (2015). https://manifestsecurity.com/android-application-security-part-3/
15. Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., Shastry, B.: Practical and lightweight domain isolation on Android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Chicago, Illinois, USA (2011)
16. Chen, Q.A., Qian, Z., Mao, Z.M.: Peeking into your app without actually seeing it: UI state inference and novel Android attacks. In: Proceedings of the 23rd USENIX Conference on Security Symposium, San Diego, CA (2014)
17. Elenkov, N.: Android explorations: using KitKat verified boot (2014). http://nelenkov.blogspot.com.au/2014/05/using-kitkat-verified-boot.html. Accessed 22 Sept. 2016