WILL ANDROID TROJANS, WORMS OR ROOTKITS SURVIVE IN SEANDROID AND CONTAINERIZATION?

Rowland Yu & William Lee
Sophos, Australia

Email {rowland.yu, william.lee}@sophos.com.au

ABSTRACT

SEAndroid and containerization have become buzz-words in mobile security over the last year. Both of them supply an isolated working environment for *Android* devices. Moreover, the main goal of each is to try to minimize the damage that can be caused by malicious applications, intruders, exploits and vulnerabilities.

SEAndroid stands for Security Enhancements for *Android*, which defines and enforces a system-wide security policy over all processes, objects and operations. It blocks extra privileges escalated by applications, separates applications from each other and the system, and prevents the bypass of security features. Meanwhile, containerization refers to the ability to separate an encrypted zone on a device and manage access to the zone. In other words, it not only secures data on a device, but also controls how applications can access, share and use the data.

Android 5.0 is trying to set itself up as a safe corporate mobile operating system by touting SEAndroid and containerization. The enforcement of SEAndroid and containerization has been changing the way OEMs and security vendors respond to security issues. However, this paper will prove that, even with these security enhancements, you can still be infected, still have data stolen, still have corporate data leaked, and experience exploration of kernel vulnerabilities.

1. INTRODUCTION

Android is a *Linux* kernel-based mobile operating system [1]. The *Linux* kernel provides a multi-user nature and discretionary access control (DAC) enforcement module on top of which all *Android* layers sit. *Android* utilizes the kernel-level sandboxing and isolation mechanism to separate apps from one another, and to control the communication between apps or resource accesses.

In the *Linux* DAC mechanism, all users have a user ID and a group ID, with a unique numerical identification number associated to the user ID (UID) and group ID (GID) respectively. The use of groups allows additional privileges and access rights. When an *Android* app is installed, it is assigned a unique UID and GID. The same UID and GID are assigned to the app's home directory and data to allow the app full access. *Android* maintains special groups for Internet, Bluetooth, external storage and more. For instance, when an app is granted the INTERNET permission, it is automatically added to the 'inet' group by the *Android* system.

However, there are some inherent weaknesses associated with DAC in the *Android* security model [2], which can cause vulnerabilities in the system's security. The vulnerabilities can be divided into two primary aspects [3]: first, the inconsistency between subject (whereby an active entity, such as a process, performs an action) and object (a passive entity, such as file or socket, which has a set of privileges) makes it possible for unauthorized parties to access certain resources. Secondly, flawed or malicious applications can bypass the permission system, escalate their privileges and directly access resources in the kernel.

Security Enhancements for *Android* (SEAndroid) has been introduced to mitigate the above shortcomings. It deploys a mandatory access control (MAC) mechanism, which piggybacks on the existing *Android* DAC model. In addition, a centralized security policy configuration is set up for all processes, objects and operations as per the defined security context. In general, SEAndroid intends to control access to applications' data and resources, protect and confine system services, protect users from potential flaws and reduce the effects of malicious apps [4].

Over the last decade, there has been a tremendous growth in the number of mobile users worldwide. More and more employees are bringing their own mobile devices to their workplace. This so-called BYOD (bring your own device) phenomenon poses a threat to corporate data. Therefore, containerization has been adopted by many organizations in mobile device management (MDM) to provide employees with secure access to corporate data and to prevent malware, intruders or other apps from accessing sensitive data.

The rest of this paper is laid out as follows. In Section 2, we provide a description of the fundamentals of SEAndroid and containerization. In Section 3, we provide an overview of how existing malware will survive with the above security enhancements. An evaluation of existing vulnerabilities and bootkits is also included. We will then try to predict the evolution of *Android* malware and vulnerabilities with respect to these enhancements in the future. Finally, we offer a conclusion.

2. THE FUNDAMENTALS OF SEANDROID AND CONTAINERIZATION WITH RELATED IMPACTS

This section gives an overview of the enhancements made to *Android* with the addition of SEAndroid and containerization. The main objective of this section is to provide background information on SEAndroid and containerization as well as look at their impact on OEMs and security vendors.

2.1 Security Enhancements for Android (SEAndroid)

Starting with *Android 4.3*, SELinux has played a significant part in the *Android* security architecture, enforcing mandatory access control (MAC) over all processes, objects and operations in a system-wide security policy [5]. SELinux is capable of confining privileged process access to files and network resources and even processes running with root or superuser identity. Moreover, it provides a centralized security configuration and automation policy against potential harm from a confined daemon that becomes compromised.

SEAndroid refers to the Security Enhancements for *Android* project [6], which has been implanted in the *Android* Open Source Project (AOSP). SEAndroid widened the scope of

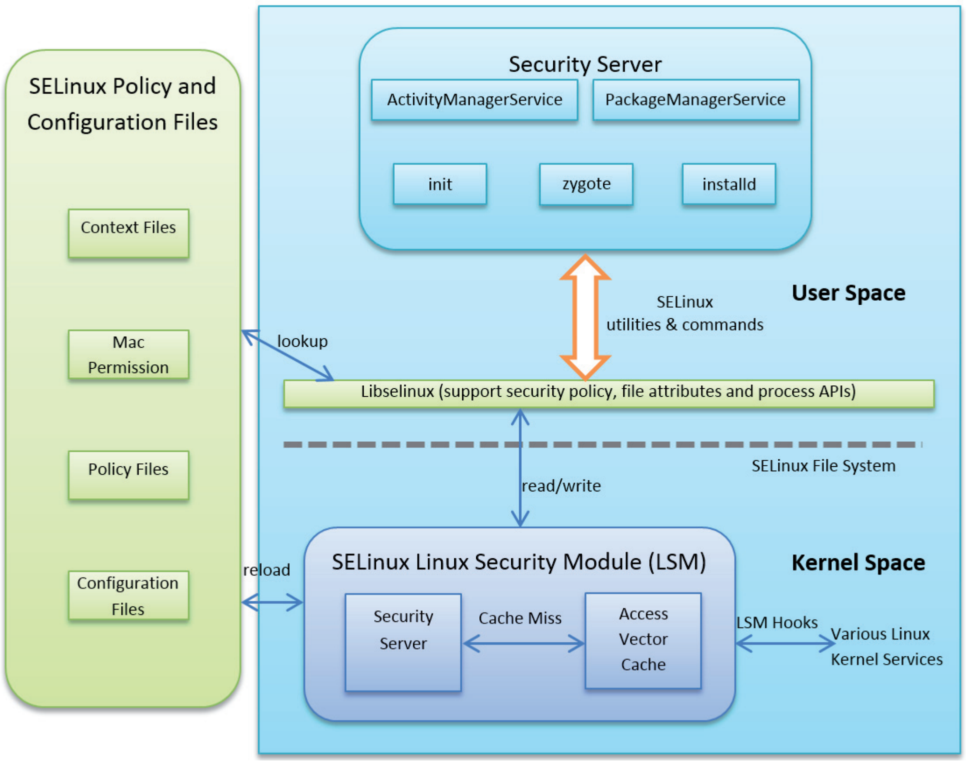

Figure 1: Overview of the SEAndroid framework.

SELinux and enabled the integration of SELinux and run-time middleware MAC (MMAC) into *Android* in a comprehensive and coherent manner. Figure 1 gives a high-level view of the SEAndroid framework [7]. The framework can be broken into three parts: kernel space, user space, and the security policy and configuration files.

2.1.1 Kernel space

In the kernel, SELinux depends on the *Linux* Security Module (LSM), audit subsystem and file system. The LSM contains an Access Vector Cache and a security server. The security server is capable of permitting or denying the use of an object by a subject. The SELinux decisions, including permitting or denying access, are cached in order to improve performance. The cache is known as the Access Vector Cache (AVC).

As a variant of *Linux*, *Android* introduces a number of new kernel subsystems and drivers such as Binder, ashmem, logger and wakelocks. As a result, major changes have been made to define new hooks in the LSM security module and to insert calls and corresponding SELinux permission checks of these hooks into relevant subsystems such as Binder. Furthermore, SELinux in the *Android* file system has gained additional support for extended attributes for security labelling.

2.1.2 Policy and configuration files

All operating system access control is based on certain types of access control attributes associated with objects and subjects [8]. The access control attribute in SELinux is described as a security context. The security context of SEAndroid is based on the concepts of Type Enforcement (TE), role-based access control and multilevel security. Under Type Enforcement, subjects accessing objects are governed through a set of rules. Subjects, also known as domains, are processes, while objects include things like files, sockets and network hosts. Subjects and objects are labelled with a single security context, which has four elements: user, role, type, and mls_level with the following format:

```
user:role:type:mls_level
```

SEAndroid includes several unique configuration files and policy definition files for computer security contexts. Table 1 shows the configuration files that can be found on devices.

`file_contexts`	Declares default security contexts for labelling all files on the system.
`seapp_contexts`	Contains information to label security contexts of application processes (domains).
`property_contexts`	The file is unique to *Android*, and declares default security contexts for *Android* properties and services.
`service_contexts`	Declares default security contexts for the *Android* subsystem, e.g. the *Android* servicemanager.
`mac_permissions.xml`	Contains certificate info used by the seinfo attribute during the installation of an app. The seinfo attribute is also part of the middleware MAC for inter-process communication (IPC).

Table 1: Configuration files that can be found on devices.

As described in Section 2.1.1, Access Vectors are the rules that determine a domain's access rights to an object. An Access Vector rule contains the subject, object, class, and the permissions granted to the subject [9]. The policy rule has the following format:

<av_action> <domain> <type>:<classes> {<permissions>}

- *av_action* defines the permitted actions including allow, auditallow, dontaudit and neverallow.
- *domain* is a label for the subject (e.g. process) or set of processes.
- *type* is a label for the object (e.g. file, socket) or set of objects.
- *class* is the type of object (e.g. file, socket) being accessed.
- *permissions* are the operation (e.g. read, write) being performed.

The *.te (type enforcement) policy files under the external/sepolicy/ directory of the Android Open Source Project contain all the required allow, type_transition and other rules for each domain. These .te files also determine access to objects by calling macros. Macros are common groupings of classes, permissions and rules to simplify the writing of rules. All policy files are built into a non-humanly readable kernel policy matrix file which is named 'sepolicy' and installed by default in the root directory.

The configuration files listed in Table 1 are also installed as parts of the SEAndroid policy files. These policy files are compiled as part of the *Android* build and added into the ramdisk image so that they can be interpreted by the init process at a very early stage in the boot process. Once the partitions have been mounted, the kernel policy as well as other policy configuration files such as file_contexts and property_contexts will be reloaded by setting the trigger selinux.reload_policy property in the init.rc configuration.

Since this paper mainly focuses on *Android* malware in the SEAndroid environment, the MAC policy configuration file, mac_permissions.xml, is the primary concern that will be explained in Section 3. The file is used for a check of application permissions against the MAC policy at install time. It utilizes the values of signature and seinfo tags to assign policy stanzas for a given app or for all apps from either the platform or third parties. Denied *Android* permissions can be specified via a blacklist (deny-permission), while a whitelist is used for allow-permissions. As a result, an application cannot be installed if the permission is not allowed, and it can't be run after an updated policy.

Listing 1 (an example of mac_permissions.xml) shows that apps with the platform certificate can have allow-all

```
<?xml version="1.0" encoding="utf-8"?>
<policy>
<!--
    Sample signer stanza for install policy
    Rules:
    Sample stanzas are given below based on the AOSP developer keys.
-->
    <!-- Platform dev key with AOSP -->
    <signer signature="....b357" >
      <allow-all />
      <seinfo value="platform" />
    </signer>
    <!-- shared dev key in AOSP -->
    <signer signature="...6f84" >
      <allow-permission name="android.permission.ACCESS_COARSE_LOCATION" />
      <allow-permission name="android.permission.ACCESS_FINE_LOCATION" />
      <allow-permission name="android.permission.ACCESS_NETWORK_STATE" />
      <allow-permission name="android.permission.ALLOW_ANY_CODEC_FOR_PLAYBACK" />
      <allow-permission name="android.permission.BIND_APPWIDGET" />
      <allow-permission name="android.permission.BIND_WALLPAPER" />
      <allow-permission name="android.permission.CALL_PHONE" />
      ....
      <seinfo value="shared" />
    </signer>
    <!-- All other keys -->
    <default>
      <seinfo value="default" />
      <deny-permission name="android.permission.ACCESS_COARSE_LOCATION" />
      <deny-permission name="android.permission.ACCESS_FINE_LOCATION" />
      <deny-permission name="android.permission.AUTHENTICATE_ACCOUNTS" />
      <deny-permission name="android.permission.CALL_PHONE" />
      <deny-permission name="android.permission.CAMERA" />
      <deny-permission name="android.permission.READ_LOGS" />
      <deny-permission name="android.permission.WRITE_EXTERNAL_STORAGE" />
    </default>
</policy>
```

Listing 1: Example of mac_permissions.xmk showing that apps with the platform certificate can have allow-all permissions.

permissions. Meanwhile, third-party apps fall into the default 'seinfo' tag and cannot access permissions such as location, camera, and calling home. However, as shown in Appendix A, there is a different story in the real world: that is, there is no any restriction on permissions for third-party apps.

The seinfo tag information will be used in the seapp_contexts configuration file shown below. Applications signed by the platform signatures run in the processes with the domain named 'platform_app', and the type of its data file is 'app_data_file'. Meanwhile, the 'untrusted_app' domain is the default assignment in seapp_contexts for all non-system apps as well as for any system apps that are not signed by the platform key.

```
# Input selectors:
#       isSystemServer (boolean)
#       user (string)
#       seinfo (string)
#       name (string)
#       path (string)
#       sebool (string)
#...
isSystemServer=true domain=system_server
user=system domain=system_app type=system_app_data_file
user=bluetooth domain=bluetooth type=bluetooth_data_file
user=nfc domain=nfc type=nfc_data_file
user=radio domain=radio type=radio_data_file
user=shared_relro domain=shared_relro
user=shell domain=shell type=shell_data_file
user=_isolated domain=isolated_app
user=_app seinfo=platform domain=platform_app type=app_
data_file
user=_app domain=untrusted_app type=app_data_file
```

The type enforcement policy rules of untrusted_app can be found in untrusted_app.te (shown in Appendix B). There, we find that untrusted applications are never allowed to send/receive uevent and netlink messages, read information in debugfs, register services, or access *Android* properties. In general, these rules attempt to isolate untrusted apps from the kernel and address privilege escalation from an unprivileged app via exploits.

2.1.3 Userspace

In the SEAndroid userspace, two aspects, the SELinux API library and app security labelling, will be explained, since security contexts for processes and app data directories may affect behaviours of *Android* malware.

A minimal port of the SELinux API library named 'libselinux' is created for *Android*. Libselinux positions itself between the kernel and userspace. The library hides the low-level functionalities of the kernel while providing interfaces and functions for apps to get and set process and file security contexts, and to obtain security policy decisions.

The zygote process, typically via the request of the ActivityManagerService (AMS), initiates the startup of all *Android* apps. Each process is assigned the DAC credentials (UID, GID and supplementary GIDs) when it is forked from the zygote. In addition to DAC, a security context is required for app processes in SEAndroid. The seinfo argument described in Section 2.1.2 is used to generate the security context via the AMS for the particular app being started.

Furthermore, each app data directory needs to be assigned the corresponding security label when created. The *Android* installd daemon is responsible for the creation of the app data directories. The installd daemon receives the seinfo value from the PackageManagerService then labels the security contexts for data directories based on the configuration file seapp_contexts.

2.2 Containerization

Containerization, also known as secure container, offers the ability to set up a separate and encrypted zone on a device. Within this secure protected zone, enterprise applications are able to be run while isolated from personal applications. Moreover, containerization can protect sensitive resources including business email, documents, contacts, calendars and intranet browsing alongside personal data. Furthermore, containerization allows a company to deploy its applications over hundreds of devices without the headache of app wrapping and data pushing.

However, containerization has some limitations. First of all, it cannot protect everything [10]: it only works on a handful of apps and data identified by the container vendor. Secondly, containerization may cause compatibility problems and break other functionalities; it may prevent other processes accessing the device's contact list and message conversations. Thirdly, the potential vulnerabilities and exploits of *Android* or other personal apps may subvert the system of containerization. An attacker can embed malicious code into alternative keyboard apps, turning these apps into keylogging spyware [12].

2.3 Impacts of SEAndroid and containerization on OEMs and security vendors

Starting from the *Android 5.0* release, *Android* has shifted from partial enforcement to full enforcement and made several significant improvements listed as follows. This means that OEMs and security vendors must fully understand and scale their implementations and apps on SEAndroid in order to provide compatible devices and comprehensive security respectively [4]:

- Everything has been in full enforcement since the 5.0 release.
- More than 60 domains, including crucial domains (installd, netd, vold, and zygote), are enforced.
- Only the init process is able to run in the init domain.
- Any generic denial prevents any special domain from starting.

Meanwhile, several enterprise mobile management and security providers have implemented containerization, since the container approach can offer strong values in securing enterprise data. In the next five years, 65 per cent of enterprises are expected to adopt a mobile device management (MDM) system with containerization functionality embedded [11]. However, companies must be aware that container solutions cannot offer absolute security, and infection cannot be guaranteed not to happen.

3. THE SURVIVAL OF EXISTING ANDROID MALWARE BASED ON ENFORCEMENTS

This section analyses a set of *Android* malware discovered by *SophosLabs* in the last 12 months. We divide these malware into different classifications including SMS senders, trojan backdoors, spyware, FakeApps, ransomware, and potentially unwanted apps (PUA). We provide a general description of their functionalities and corresponding permissions. Subsequently, analysis and tests of these families are performed on the basis of SEAndroid and containerization.

3.1 Classifications of Android malware

During the last 12 months, *SophosLabs* has recorded close to 1.5 million unique pieces of *Android* malware. Figure 1 shows the classification of these malware. Premium SMS senders form the largest malware type and accounts for 55.6% of the total. Backdoor trojans hold the second position, accounting for nearly 21%. The third largest classification is spyware. The rest of the malware types, such as rootkits, banker trojans and ransomware, account for only 7.5%.

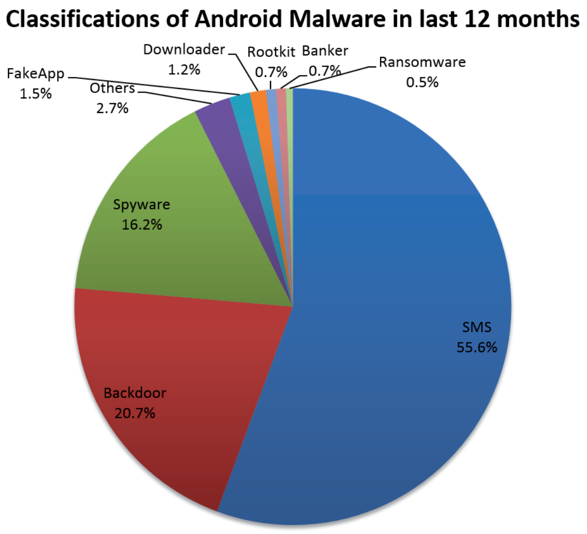

Figure 2: Android malware classification from May 2014 to April 2015.

3.1.1 Premium SMS senders

Over the last couple of years, premium SMS sender trojans have become the largest threat plaguing *Android* because this is the easiest way for cybercriminals to make money fast. SMS trojans usually take advantage of social engineering techniques and masquerade as popular apps, games, pornographic attractions and more.

Cybercriminals can register short codes via premium SMS providers, and deploy them in their malicious apps. In *Android* these apps require only one necessary permission, which is 'android.permission.SEND_SMS'; they then use the sendTextMessage () method to silently send SMS messages to the premium SMS short code. We have demonstrated that this is still an achievable method under the enforcement of SEAndroid and containerization.

3.1.2 Backdoors

Android backdoors are advanced mobile trojans and usually have the ability to perform multiple tasks as follows:

- Set up or distribute via a mobile botnet
- Send or intercept SMS messages
- Download, install, or activate any *Android* app without the user's knowledge
- Make arbitrary phone calls
- Clear user data, uninstall existing applications, or disable system applications
- Upload sensitive information including device ID, locations, application usage, call log and SMS history to remote websites
- Execute command and control services.

Backdoors are one of the most complicated types of *Android* malware. Malware writers use a variety of methods and techniques in their malicious apps. Therefore, it is hard to measure the effectiveness of SEAndroid and containerization in addressing the threat of *Android* backdoors. If SEAndroid is enabled, a normal backdoor trojan will have trouble carrying out its functionalities such as installing itself into the system directory, disabling system apps, or gaining access to apps' data – but it is still able to steal and upload sensitive info, download and ask to install applications, and set up mobile botnets when setting proper *Android* permissions.

However, sophisticated backdoors like CoolReaper [13] are hidden in ROM images either by legitimate OEMs or third-party distributors. As a result, such backdoors have the superuser privilege and can accomplish any task without the worry of SEAndroid or containerization because the security setting can easily be customized or disabled by the OEMs or distributors.

3.1.3 Spyware and banker trojans

Android spyware and banker trojans have key functionalities in common, which aim to gather sensitive information about a person, organization or device and send such information to another entity without the user's consent [14]. In *Android*, the information contains the device ID, app info, call and SMS log, contacts, locations, and login and password details. The information could be sent out via Internet or SMS messages. Moreover, a banker trojan is able to scan for legitimate banking apps, replace them with bogus apps, and attempt to disable any mobile security software.

In SEAndroid, most functionalities of spyware and banker trojans can be achieved via declaring suitable permissions such as 'android.permission.READ_SMS', 'android.permission.RECEIVE_SMS', 'android.permission.READ_PHONE_STATE', and 'android.permission.READ_CONTACTS'. Additionally, malware writers can steal bank login information by launching a pop-up activity like a phishing website. However, with SEAndroid enabled, this kind of malware is not able to delete legitimate banking apps and disable security software silently. Also, containerization is able to prevent the leakage of enterprise data and even access to SMS or contact information since enterprise data is locked inside the container and only authorized apps can access it.

3.1.4 FakeApps and ransomware

FakeApps, especially FakeAV, report fake security alerts to scare victims into paying money for simulated removal of malware. The payment can be made online via credit card or via premium SMS. The writers of FakeApps may perform additional tasks of downloading and installing malware, and stealing sensitive information. Some FakeApps and *Android* ransomware also lock up your device with a pop-up that freezes out all other apps. Moreover, crypto ransomware such as SimpleLocker has the ability to encrypt documents, pictures, audios and videos.

In general, both FakeApps and ransomware can survive with SEAndroid and containerization enforcement and can cause damage to personal information. However, FakeApps and ransomware are not able to access or steal sensitive information in enterprise security containers. Nevertheless, the malicious behaviour of lock-out and encryption will seriously decrease the usability of your device. And these malware can only be removed either by rebooting into safe mode or by performing a factory reset.

3.1.5 Rootkits and bootkits

In summary, SEAndroid is designed to minimize the risks from exploits and vulnerabilities for *Android*. It can effectively block existing exploits and vulnerabilities during their execution [5]. However, it is difficult for SEAndroid to prevent the operating system from being compromised. The *Samsung Galaxy S4*, built on SEAndroid, was rooted just a few months after being released [15]. Also, the latest *Samsung Galaxy S6* and *S6 Edge* were rooted by the PingPong exploit [16] less than 30 days after their first release. The *S6* and *S6 Edge* are running on *Samsung Knox*, extending SEAndroid.

In the real world, the number of rooted devices is far greater than the expectation. A study [17] from Chinese Internet portal *Tencent* shows that 27.44% of mobile users actually rooted their devices in order to remove built-in apps or customize the devices. Considering that China now has 386 million active *Android* users [18], the count of rooted devices is huge. As users usually download rootkits from third-party stores or buy rooted devices from grey markets, it comes as no surprise that malicious rootkits are so popular, and bootkit malware like Oldboot can infect hundreds of thousands of devices [19].

4. THE EVOLUTION OF ANDROID MALWARE AND VULNERABILITIES IN THE NEAR FUTURE

SEAndroid introduces security enhancements with SELinux-based mandatory access control (MAC) to provide centralized policy management for every device. Meanwhile, containerization creates a separate encrypted zone on a device for enterprise resources and supplies another layer for secure access to these resources in the container. The goal of both enhancements is to limit or block the damage or information leakage caused by malware [20, 21].

In fact, none of the security enhancements address one of the core issues related to the *Android* permission model. Permissions are the key to control if an *Android* app can access sensitive information. However, it is hard to distinguish malicious apps from clean apps only with requested permission information. Appendix C shows the permission information from 8,000 popular clean apps, the majority of which contain potentially dangerous permissions that are also widely used in *Android* malware. So far, it seems that neither SEAndroid nor containerization has been able to tackle the critical problems as a significant increase in malware has been recorded by *SophosLabs*.

As a result, *Google I/O* 2015 introduces a new app permissions model that allows the user to pick and choose what permissions an app can access at runtime [22]. The proposal tries to address leftover security issues from SEAndroid and containerization. That would be a good move, but the majority of users have little knowledge of the scope and implications of permissions and may not make the correct security decisions. So it will rely heavily on OEMs and security service providers to continue their effort in delivering better solutions.

In summary, the rising trends will continue to dominate *Android* malware attacks in 2015. *Android* malware has been getting smarter and aiming to generate more profit. Since SMS mobile payments are still a popular and common method of payment in some regions [23], the incentive of premium SMS senders for mobile malware developers is simple and clear. Moreover, where *Windows* has gone, *Android* has followed: the *Android* ransomware – Koler and SimplerLocker – borrow their method of operation from similar malware on *Windows*. Last but not least, rootkits and bootkits will emerge time and time again because of the open nature of *Android* and the habit of rooting devices.

5. CONCLUSION

This paper explains the need for the SEAndroid enhancement due to the inherent weaknesses associated with *Android*'s DAC security model. It also describes the management control of containerization for content and apps. Additionally, we show their impact on OEMs and security vendors. However, both enhancements have failed to meet their objectives against the damage or information leakage caused by *Android* malware. The failures have been demonstrated through the significant increase in *Android* malware. Furthermore, we expect the continuing increase of smarter *Android* malware in the future, which will effectively take advantage of social engineering and bypass these security enhancements.

REFERENCES

[1] Android (operating system). http://en.wikipedia.org/wiki/Android_%28operating_system%29.

[2] Android and Security. http://viewpoint.sasken.com/?p=422.

[3] Thompson, D. J.; Wang, X.; Zhou, X. A Study of the Consistency of Android Permissions and Linux DAC. Indiana University School of Informatics.

[4] Security-Enhanced Linux in Android. http://source.android.com/devices/tech/security/selinux/.

[5] Smalley, S.; Craig, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. Trusted Systems Research National Security Agency https://www.internetsociety.org/sites/default/files/02_4.pdf.

[6] Security Enhancements (SE) for Android. http://seandroid.bitbucket.org/index.html.

[7] SEforAndroid. http://morris--notes.blogspot.com.au/2013/05/seforandroid.html.

[8] SELinux Concepts. http://www.informit.com/articles/article.aspx?p=606586.

[9] SELinux concepts from Google. https://source.android.com/devices/tech/security/selinux/concepts.html.

[10] Pos and cons of using secure containers for mobile device security. http://searchconsumerization.techtarget.com/feature/Pros-and-cons-of-using-secure-containers-for-mobile-device-security.

[11] Mobile enterprise management tools are targeted by spyphones, researchers warn. http://www.cio.com.au/article/456368/mobile_enterprise_management_tools_targeted_by_spyphones_researchers_warn/.

[12] How Mobile Malware Bypasses Secure Container Solutions. https://www.lacoon.com/wp-content/uploads/2013/07/BypassingTheContainer-180613.pdf.

[13] CoolReaper Revealed: A Backdoor in Coolpad Android Devices. http://researchcenter.paloaltonetworks.com/2014/12/coolreaper-revealed-backdoor-coolpad-android-devices/.

[14] Spyware. http://en.wikipedia.org/wiki/Spyware.

[15] Mobile Security Myth 2: Deploying Containers Will Lock Down Corporate Data. https://bluebox.com/business/mobile-security-myth-2-deploying-containers-will-lock-down-corporate-data/.

[16] PingPongRoot ***S6 & S6 Edge Root Tool***. http://forum.xda-developers.com/galaxy-s6/general/root-pingpongroot-s6-root-tool-t3103016.

[17] Over 27.44% Users Root Their Phone(s) In Order To Remove Built-In Apps, Are You One Of Them?. http://www.androidheadlines.com/2014/11/50-users-root-phones-order-remove-built-apps-one.html.

[18] China now has 386 million active Android users. https://www.techinasia.com/china-386-million-active-android-users-q2-2014/.

[19] Most Sophisticated Android Bootkit Malware ever Detected; Infected Millions of Devices. http://thehackernews.com/2014/04/most-sophisticated-android-bootkit.html.

[20] SELinux for Android and Samsung KNOX. http://www.all-things-android.com/content/selinux-android-and-samsung-knox.

[21] Google Slashes Android Malware in Half. http://blogs.csc.com/2015/04/06/google-slashes-android-malware-in-half/.

[22] Google Locks Down Excessive Android App Permissions. https://threatpost.com/google-locks-down-excessive-android-app-permissions/113051.

[23] Report: Huge spike in mobile malware targets Android, especially mobile payments. http://www.pcworld.com/article/2691668/report-huge-spike-in-mobile-malware-targets-android-especially-mobile-payments.html.

APPENDIX A: SAMPLE MAC_PERMISSIONS.XML

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- AUTOGENERATED FILE DO NOT MODIFY -->
<policy>
<signer signature="...e26a">
    <seinfo value="platform"/>
</signer>
<default>
    <seinfo value="default"/>
</default>
</policy>
```

Sample mac_permissions.xml from a Nexus 5 running on Android 5.1.

APPENDIX B: SAMPLE UNTRUSTED_APP.TE POLICY

(Please see next page.)

APPENDIX C: HIGH-RISK PERMISSIONS IN TOP 8000 ANDROID APPS FROM GOOGLE PLAY

Permission	Percentage
INTERNET	98.1%
WRITE_EXTERNAL_STORAGE	75.3%
READ_PHONE_STATE	43.4%
ACCESS_FINE_LOCATION/ACCESS_COARSE_LOCATION	25.3%
READ_EXTERNAL_STORAGE	16.3%
CAMERA	13.8%
READ_CONTACTS	12.6%
SYSTEM_ALERT_WINDOW	8.8%
CALL_PHONE	6.9%
RECEIVE_SMS	5.0%
SEND_SMS	4.7%
READ_SMS	3.8%
READ_CALENDAR	3.1%
MOUNT_UNMOUNT_FILESYSTEMS	2.6%
READ_CALL_LOG	1.9%
PROCESS_OUTGOING_CALLS	1.9%

High-risk permissions in top 8000 Android apps from Google Play.

APPENDIX B: SAMPLE UNTRUSTED_APP.TE POLICY

```
### Untrusted apps
###
### untrusted_app includes all the appdomain rules, plus the
### additional following rules:
###

type untrusted_app, domain;
app_domain(untrusted_app)
net_domain(untrusted_app)
bluetooth_domain(untrusted_app)

# Some apps ship with shared libraries and binaries that they write out
# to their sandbox directory and then execute.
allow untrusted_app app_data_file:file { rx_file_perms execmod };

allow untrusted_app tun_device:chr_file rw_file_perms;

# ASEC
allow untrusted_app asec_apk_file:file r_file_perms;
# Execute libs in asec containers.
allow untrusted_app asec_public_file:file { execute execmod };

# Allow the allocation and use of ptys
# Used by: https://play.google.com/store/apps/details?id=jackpal.androidterm
create_pty(untrusted_app)

# Used by Finsky / Android "Verify Apps" functionality when
# running "adb install foo.apk".
# TODO: Long term, we don't want apps probing into shell data files.
# Figure out a way to remove these rules.
allow untrusted_app shell_data_file:file r_file_perms;
allow untrusted_app shell_data_file:dir r_dir_perms;

# b/18504118: Allow reads from /data/anr/traces.txt
# TODO: We shouldn't be allowing all untrusted_apps to read
# this file. This is only needed for the GMS feedback agent.
# See also b/18340553. GMS runs as untrusted_app, and
# it's too late to change the domain it runs in.
# This line needs to be deleted.
allow untrusted_app anr_data_file:file r_file_perms;

#
# Rules migrated from old app domains coalesced into untrusted_app.
# This includes what used to be media_app, shared_app, and release_app.
#

# Access /dev/mtp_usb.
allow untrusted_app mtp_device:chr_file rw_file_perms;

# Access to /data/media.
allow untrusted_app media_rw_data_file:dir create_dir_perms;
allow untrusted_app media_rw_data_file:file create_file_perms;

# Write to /cache.
allow untrusted_app cache_file:dir create_dir_perms;
allow untrusted_app cache_file:file create_file_perms;

###
### neverallow rules
###

# Receive or send uevent messages.
neverallow untrusted_app domain:netlink_kobject_uevent_socket *;

# Receive or send generic netlink messages
neverallow untrusted_app domain:netlink_socket *;
```

```
# Too much leaky information in debugfs. It's a security
# best practice to ensure these files aren't readable.
neverallow untrusted_app debugfs:file read;

# Do not allow untrusted apps to register services.
# Only trusted components of Android should be registering
# services.
neverallow untrusted_app service_manager_type:service_manager add;

# Do not allow untrusted_apps to connect to the property service
# or set properties. b/10243159
neverallow untrusted_app property_socket:sock_file write;
neverallow untrusted_app init:unix_stream_socket connectto;
neverallow untrusted_app property_type:property_service set;

# Allow verifier to access staged apks.
allow untrusted_app { apk_tmp_file apk_private_tmp_file }:dir r_dir_perms;
allow untrusted_app { apk_tmp_file apk_private_tmp_file }:file r_file_perms;
```

Sample untrusted_app.te policy.