Characterizing SEAndroid Policies in the Wild

Keywords: Security, Access Control, SELinux, SEAndroid

Abstract: Starting from the 5.0 Lollipop release all Android processes must be run inside confined SEAndroid access control domains. As a result, Android device manufacturers were compelled to develop SEAndroid expertise in order to create policies for their device-specific components. In this paper we analyse SEAndroid policies from a number of 5.0 Lollipop devices on the market, and identify patterns of common problems we found. We also suggest some practical tools that can improve policy design and analysis. We implemented the fist of such tools, SEAL.

1 INTRODUCTION

During the past decade Android has gained a considerable share of the mobile device market. However, at the same time the number of malware and various exploits available for Android has also been increasing (Zhou and Jiang, 2012; Smalley and Craig, 2013). Many classical Android exploits, such as GingerBreak and Exploid (Smalley and Craig, 2013), attempt to target system daemons that run with elevated, often unlimited, privileges. Once such a daemon is compromised, the whole Android OS usually becomes compromised and the attacker is able to get permanent root privileges on the device. Since the Android permission system, which relies on Linux Discretionary Access Control (DAC), cannot protect from such attacks, a new Mandatory Access Control (MAC) mechanism has been introduced. SEAndroid (Smalley and Craig, 2013) is an Android port of the well-established SELinux MAC mechanism (Smalley et al., 2001) with some Android-specific additions and modifications. In SELinux, security decisions are taken according to a policy: the reference policy for SEAndroid was created from scratch and is maintained as part of the Android Open Source Project (AOSP) [1].

Starting from the Android 5.0 Lollipop release, the Android compliance requirements have mandated that every process must be run inside a confined SEAndroid domain with a proper set of access control rules defined. This has put many Android Original Equipment Manufacturers (OEMs) in the difficult position of enabling SEAndroid in enforcing mode on their devices with a set of fully configured access control domains. While the reference SEAndroid policy is provided by AOSP, any OEM customization to the reference AOSP device design results in a need for SEAndroid policy modifications. Writing well-designed SELinux policies requires expertise; this difficulty, together with high time-to-market pressure, can possibly lead to the introduction of mistakes and outright vulnerabilities in modified SEAndroid policies deployed in OEM Android devices.

In this paper, we conduct a systematic manual analysis of several available SEAndroid 5.0 Lollipop OEM policies and identify common patterns and mistakes. We find that OEM modifications can render policies less strict, resulting in a wider attack surface for potential vulnerabilities. Based on these findings, we identify a number of practical tools that can assist SEAndroid policy designers and researchers to analyze and improve SEAndroid policies. We also provide an initial implementation of one such tool, *SEAL*. To the best of our knowledge, this is the *first comparative study* of SEAndroid policies from real-world devices.

2 BACKGROUND

2.1 SELinux

SELinux (Smalley et al., 2001) is a well-established MAC mechanism available for Linux-based distributions. It was the first MAC for mainstream Linux with its initial release in 1998. SELinux has been implemented in the Linux kernel following the Flask architecture (Spencer et al., 1999), where the policy enforcement code, known as Linux Security Module (LSM) Framework, and the policy decision-making code are separated: this allows other MAC modules, such as AppArmor (Bauer, 2006) or Smack (Schau-

[1] `source.android.com`

fler, 2008), to utilize the same policy enforcement code.

The main MAC mechanism in SELinux is Domain/Type Enforcement (Badger et al., 1995), which assigns a `type` to each subject or object in the system; a subject's type is also known as `domain`. A subject running in `domain` can only access an object belonging to `type` if there is an `allow` rule in the policy of the following form:

```
allow domain type : class permissions
```

where `class` represents the nature of an object such as file, socket or property, and `permissions` represent the types of operation on this object that are being controlled, like `open`, `write`, `set` etc. Subjects can change their `domain` if a corresponding type transition rule is defined. For example, if a process executes a new binary, it is possible for the resulting process to run in a different `domain`. Such a transition rule will be represented as:

```
type_transition olddomain type:process
                newdomain
```

where `type` denotes the type of a binary that should be executed in order for the transition to happen. We will use the term *process transition* in the future to refer to such a rule. In practice there are a number of additional rules that are needed in order to make the transition happen, but we leave them out here for the sake of simplicity. Another type of transition can occur if an object is created and its `type` should differ from the `type` of the object's parent. For file creation inside an existing directory such a rule can be represented as:

```
type_transition domain oldtype:dir newtype
```

where `domain` denotes the domain of the subject creating the file, `oldtype` represent the `type` of the directory where the file is being created and `newtype` denotes the `type` that the new file should be assigned.

In addition, the SELinux policy language has the following notions:

- An `attribute` is a way to refer to sets of types and domains. It is used to express type hierarchies and rule inheritance. For example, SELinux policies on Android define an `app` attribute consisting of common rules for all platform applications such as the ability access the device display.
- *Initial* `Security Identifier (SID)` are types that should be assigned by default to subjects and objects during system initialization, such as for example `kernel` and `init`.
- `genfs` contexts define types that should be assigned to objects residing in special filesystems, such as `proc`, `debugfs` and `ecryptfs`.

2.2 SEAndroid

The SELinux port to Android, SEAndroid (Smalley and Craig, 2013), was mostly based on SELinux code with some additional LSM hooks to support Android-specific mechanisms, such as Binder Inter Process Communication (IPC). However, the SEAndroid reference policy was written from scratch due to 1) a desire for a simpler and a smaller policy and 2) the big difference between the userspace layers of Android and a standard Linux distribution.

SEAndroid classes and permissions are mostly the same as on SELinux, with some Android-specific additions like the `property` class for the Android init-based property service and the `keystore_key` class for the Android keystore key object.

Native services and daemons are assigned SEAndroid domains based on filesystem labeling or direct domain declaration in the service definition in the `init.rc` file. In turn, applications are assigned domains based on the signature of the Android application package file (`.apk`). There are a number of predefined application domains, like `system_app`, `platform_app` and `untrusted_app`. OEMs are able to create additional domains if needed.

One new notion that SEAndroid has is the presence of `neverallow` rules in the source policy. A `neverallow` rule specifies that certain accesses should never be allowed by the policy. For example, the following `neverallow` rule asserts that only processes running in the `init` domain should be able to modify security-sensitive files in the `proc` filesystem:

```
neverallow {domain -init}
proc_security:file {append write}
```

If one tries to add a rule that conflicts with a `neverallow` rule, the policy compilation fails.

SEAndroid was initially added to the AOSP codebase for Android 4.3 back in 2012; at that point, it was configured in permissive mode. In Android 4.4, SEAndroid was switched to enforcing mode: however, most domains were left in permissive mode, apart from a number of core AOSP domains such as `init` and `vold`. Android 5.0 Lollipop eventually required every single process to be put in an enforcing domain, effectively extending the enforcement to the whole system.

2.3 OEM Modifications to the AOSP SEAndroid Reference Policy

Default AOSP services, processes and applications are already covered by the AOSP SEAndroid reference policy. Typically, OEM Android devices are

highly customized with their own specific drivers, new services, processes and filesystem mounts. In order for these custom components to work, appropriate additions must be made to the SEAndroid reference policy. OEMs were allowed to make additions to the SEAndroid reference policy right from the start, but very few of them actually did in Android 4.3 and 4.4: and in fact, the resulting policy was stricter than the AOSP reference policy. The stringent requirements of Android 5.0 Lollipop, however, forced all OEMs to deploy comprehensive policies defining complete rules for their own custom services: this turned out to be a challenging task for most. The inherent difficulty of incorporating SEAndroid in the development process, combined with high time-to-market pressure, has resulted in the introduction of anti-patterns, mistakes and potential vulnerabilities in OEM policies.

3 SEANDROID POLICIES IN DEPLOYED OEM DEVICES

3.1 Statistical Analysis

We collected 8 policy files from non-rooted, off-the-shelf commercial Android 5.0 Lollipop devices by different manufacturers. Table 1 shows the comparison of all basic policy attributes and characteristics with regards to the Android 5.0 Lollipop AOSP SEAndroid reference policy in the following categories:

- **Policy size.** All OEMs increased the policy size, by factors ranging from 1.1 up to 3.2.
- **Types, domains, type transitions and domain transitions.** The overall ratio of newly added domains to newly added types ranges between 4.7 and 6.5, which is very close to the ratio in AOSP itself (6.3). We conjecture that OEMs tend to add slightly simpler domains, with fewer types per each domain defined. The ratio of newly added process transitions to newly added domains is close to 1; this indicates that OEMs add simple domains, with only one process transition to these domains either from the `init` domain (upon system startup) or from the parent process domain (usually upon execution of processes such as `shell` or `toolbox`). This ratio is 1.4 for LG G3, due to a number of newly defined domains and having two or more transitions to the `shell`, `toolbox`, `dumpstate` and `logcat` domains. New type transitions are mostly used by OEMs for `tmpfs` types, as in the following example:

```
type_transition aal tmpfs : file
                aal_tmpfs
```

- **Allow rules.** The ratio of total number of allow rules to total number of types varies between 10.9 and 13.1, with the exception of 14.8 for Motorola G and 18.7 for LG G3; the ratio for AOSP is 12.0. The numbers for Motorola and LG are comparatively excessive, and may indicate overly permissive policies; this may be due to the use of tools to automatically generate policies from system logs.
- **Attributes.** Only Samsung and Sony define new attributes. Sony adds only one, probably related to the system update process. In contrast, Samsung adds many new attributes, which seem to be auto-generated and most probably used for policy optimization. The rest of the OEMs add separate domains for their services and applications, and do not introduce any new domain hierarchies: this may imply unfamiliarity with the use of policy hierarchies.
- **Classes, permissions and initial SIDs.** OEMs do not modify the default set of SEAndroid `classes` (86), `permissions` (267) or initial `SIDs` (27): this is to be expected, since they represent interfaces and objects recognized and supported by SEAndroid. The only change we observed was for Samsung S6, that had 4 more permissions defined: `delete_as_user`, `get_by_uid`, `insert_as_user` and `set_max_retry_count`, all granted on the `keystore_key` class. The most probable reason for such additions is Samsung's implementation of the keystore and its API, which requires specific permissions.
- `genfs` **contexts.** The primary reason for the addition of `genfs` contexts is that most OEMs have additional mount points and filesystems on their devices, which by default would be labeled as `unlabeled` unless a proper `genfs` context for it is specified. An AOSP `neverallow` rule prohibits any OEM domain from creating files with this type: this restriction has forced OEMs to define proper types for their new mount points.

3.2 Systematic Manual Analysis

We manually searched each policy for OEM misconfigurations: we used existing tools for SELinux policy analysis, which we found to be cumbersome. Our primary tool was `apol` (Tresys, 2014), a GUI tool that allows the user to load a binary policy and examine it by specifying various filters. However, `apol` was not suitable for comparing two policies: it was necessary to run two instances of `apol` simultaneously,

Table 1: Policy comparison and complexity.

	size (KB)	*types*	*domains*	*type trans*	*process trans*	*allow rules*	*attributes*	*genfs contexts*	*untrusted_app rules*
AOSP	117	341	54	95	41	4096	21	30	33
LG Nexus 5	134	416, +75	65, +11	158, +63	51, +10	4972, +876	21	32, +2	33
Intel	127	393, +52	65, +11	115, +20	51, +10	4748, +652	21	32, +2	38, +5
HTC M7	181	621, +280	106, +52	213, +118	95, +54	7587, +3491	21	34, +4	46, +13
Motorola G	193	590, +249	92, +38	199, +104	83, +42	8753, +4657	21	33, +3	33
LG G3	302	851, +510	149, +95	340, +245	180, +139	15921, +11825	21	45, +15	168, +135
Intex Aqua	230	900, +559	142, +88	266, +171	128, +87	9824, +5728	21	37, +7	44, +11
Samsung S6	370	1102, +761	215, +161	430, +335	180, +139	14412, +10316	158, +137	43, +13	81, +48
Sony Xperia	218	793, +452	139, +85	265, +170	113, +72	9308, +5212	22, +1	37, +7	42, +9

+ denotes the number of additions compared to AOSP

manually insert the same queries into both and examine the differences between the outputs. Another tool was `sediff` (Tresys, 2014), which can do basic policy comparison but does not allow filtering based on specific types or domains.

To make our manual analysis tractable, we identified three sets of types that we consider important to check. The first set comprises core Android and security-sensitive domains, such as `init`, `vold`, `keystore`, `tee`, as well as types that protect access to security-sensitive areas of the filesystem, such as `proc_security`, `kmem_device` and `security_file`. The second is the set of default types that would be assigned to an object upon its creation unless a concrete type is specified in one of the policy files. The third is the set of types that would be assigned to untrusted code and its data, primarily the `untrusted_app` domain.

Analyzing these sets of types and the associated rules, we discovered the following patterns across many devices from different OEMs.

3.2.1 Overuse of Default Types

As mentioned above, a default type is one that is assigned to an object upon creation unless a dedicated type for it is specified in the policy files: examples include `unlabeled`, `device`, `socket_device`, `default_prop` and `system_data_file`. Table 2 shows that in many cases OEMs overuse the default SEAndroid object types. For example, compared to AOSP, HTC M7 has 10 new rules allowing various system daemons, such as `healthd`, `netd`, `vold`, `mediaserver`, `wpa`, `system_server`, to set system properties with the default type `default_prop`. In practice, this means that some of the system properties belonging to these components end up labeled as `default_prop`. Similarly, HTC M7 has 13 more rules granting various system daemons (`rild`, `mediaserver`, `thermal-engine`, `sensors`, `thermald`, `system-server`, ...) `write` access to the default `socket_device` object type. The only exceptions when OEM actually reduced the number of rules with regards to default type `device` is LG Nexus 5 and Motorola G policies, where they removed a rule belonging to `logd`. Below are concrete example rules from different OEMs to show the usage of default types:

```
allow thermald socket_device : sock_file
    {write create setattr unlink}

allow mediaserver default_prop :
    property_service set

allow untrusted_app unlabeled : dir
    {ioctl read getattr search open}

allow untrusted_app unlabeled :
    filesystem getattr
```

Plausible reasons for OEMs to use default types include the fact that objects are automatically assigned default types, and the common practice of using tools like `audit2allow` (SELinux Project, 2014) which parse audit logs and automatically create new allow rules to permit denied accesses.

There are two main consequences of such mistakes. Foregoing distinct, dedicated types in favor of default types means that different, unrelated resources are collected under a common label: domains with access to said label thus get wider access rights than actually needed. This is undesirable, as it violates the principle of least privilege. The second, more severe, consequence is that some untrusted domains might be given access to default labeled sensitive objects.

Fortunately, we did not find examples of such cases in the policies we examined, apart from the example above where `untrusted_app` is given some access to `unlabeled` filesystem objects; however, the possibility of such mistakes remains.

Google is actively trying to address this problem by fine-tuning the set of `neverallow` rules in the AOSP reference policy. Starting from Android 5.1, OEMs are not allowed to modify this set. For example, it is not possible anymore for an OEM domain to set default properties or access block devices.

Table 2: Usage of default types by OEMs.

	unlabeled	***socket_device***	***device***	***default_prop***	***system_data_file***
AOSP	25	4	18	0	42
LG Nexus 5	25	7, +3	17, –1	0	45, +3
Intel	27, +2	5, +1	24, +6	3, +3	54, +12
HTC M7	31, +6	17, +13	26, +8	10, +10	68, +26
Motorola G	31, +6	7, +3	17, –1	2, +2	62, +20
LG G3	42, +17	21, +17	28, +10	3, +3	120, +78
Intex Aqua	33, +8	8, +14	23, +5	0	108, +66
Samsung S6	24, –1	22, +18	46, +28	1, +1	204, +162
Sony Xperia	25	15, +11	21, +3	1, +1	57, +15

± *denotes the number of additions/removals compared to AOSP*

3.2.2 Overuse of Predefined Domains

Another observed trend is that typically OEMs do not define separate domains for specific system applications, but tend to place them either in `system_app` or in `platform_app` domains. Consequently, these domains accumulate a lot of `allow` rules that are shared by all system or platform applications. Moreover, if many applications are pre-installed in the same domain, SEAndroid cannot prevent privilege escalation attacks or unauthorized data sharing by such apps (Smalley and Craig, 2013). As an example, let us consider the pre-installed McAfee anti-virus application on LG devices. It runs in the `system_app` domain, which contains more than 900 associated allow rules in the LG G3 policy compared to 46 in the AOSP one. It is quite difficult to identify specific rules that were added to the `system_app` domain because of the McAfee application, given that many other applications run in this domain. However, by analyzing the permissions of the same McAfee application in Google Play Store, we observed corresponding SEAndroid policy rules in the `system_app` domain, such as access to the telephony functionality, camera and several types related to the filesystem, including `tmpfs` types and sockets. This might indicate that these rules were added for the McAfee application.

A solution to this problem would be to place certain powerful system applications in their own SEAndroid domains; this can be done by signing these applications with different keys and creating a mapping between these keys and target application domains.

3.2.3 Forgotten or Seemingly Useless Rules

Another common trend is the presence of rules that seem to have no effect. One example is rules of the following type present in one device:

```
allow untrusted_app <xyz>_exec : file
                <file op>
```

For example

```
allow untrusted_app tee_exec : file {read
        getattr execute open}
```

Since no corresponding process transition rule from the `untrusted_app` domain to the `tee` domain via `tee_exec` file is defined, and no `execute_no_trans` access type is granted, a process running in the `untrusted_app` domain cannot execute a file labeled as `tee_exec`. There are two plausible explanations. One is the use of tools like `audit2allow` (SELinux Project, 2014) to automatically generate rules, as discussed above. The other is the failure to clean up rules that were tested at some point but are no longer required. Below is an example of a vestigial rule that allows access to the debug interface of the Qualcomm KGSL GPU driver, which is itself disabled in production builds:

```
allow untrusted_app sysfs_kernel_debug_kgsl
        : file {read getattr}
```

3.2.4 Potentially Dangerous Rules

When working under tight time-to-market requirements, OEMs might decide to ship less strict security policies rather than make invasive changes to their codebase. This leads to a number of potentially dangerous rules appearing in OEM policies, like access to the `procfs` security-related filesystem objects. The rules below give `read/write` permissions on such objects to a trusted `hal` domain, a `release_app` domain and an `untrusted_app` domain.

```
allow hal proc_security : file {write
        getattr open}
```

```
allow release_app proc_security : file
    {ioctl read getattr lock open}
```

```
allow untrusted_app proc_security : file
        {read getattr open}
```

While processes running in the hal system domain can be considered trusted, the first rule is undesirable because it increases the attack surface of certain interfaces (like sensitive procfs settings) if the trusted process is compromised. The same applies to applications put in the release_app domain, as in the second rule. The third rule is even more dangerous, because it allows malicious applications running in the untrusted_app domain to get sensitive information, such as mmap_min_addr, memory randomization parameters and kernel pointer exposure settings, that can be used for further exploits.

Another example of a potentially dangerous rule is allowing processes running in the untrusted_app domain to read/write application data belonging to the system_app domain:

```
allow untrusted_app system_app_data_file :
        file {read write getattr}
```

However, since processes from untrusted_app and system_app domains will be run with different UIDS, the Linux DAC layer would guard against such arbitrary accesses unless a system application erroneously made its own files world-accessible.

In general, OEMs should have no additions to the set of rules for the untrusted_app domain, because any new allow rule increases the possible attack surface for malicious untrusted_app applications. However, Table 1 shows that almost all OEMs do add new rules for the untrusted_app domain.

On a positive note, OEMs are aware of such mistakes; some of them have been already fixed in the subsequent Android 5.1 update. The major reason behind these fixes was the release of the Android Compatibility Test Suite (CTS)[2] version 5.1, that added tests to ensure that AOSP neverallow rules are not violated by any process running on a device.

3.2.5 Discussion

We found several problematic patterns in the Android 5.0 OEM SEAndroid policies we examined. We conjecture that the reason for their presence is the relative unfamiliarity with SEAndroid. Google utilizes the set of neverallow rules in order to try to prevent OEMs from making security mistakes. However, while this approach might prevent some mistakes, it can also create difficulties for OEMs. For example any Global Platform-enabled TEE design [3] will likely end up with untrusted_app applications needing to access a kernel driver for their memory referencing. In Android 5.1 this conflicts with the existing neverallow rules, and as a result OEMs are forced to come up with a workaround. In the next section we propose a set of tools that can further help OEMs to avoid security mistakes and at the same time do not imply any restrictions on OEMs.

[2] source.android.com/compatibility/cts/index.html

[3] globalplatform.org

4 NEW TOOLS FOR SEANDROID

Although our systematic manual analysis unearthed some problem areas in the policies we analyzed, the process was cumbersome and time-consuming using the currently available tools. Based on our experience, we argue that new tools or new functionality in existing tools are necessary to aid both OEMs and security researchers to create and analyze SEAndroid policies effectively. We identify several such desirable tools below. We have implemented the first on the list (live policy analyzer) and are working on the rest.

4.1 Live Policy Analyzer

Existing SELinux policy analysis tools focus solely on the policy itself, and do not address the question of how the policy rules apply to a specific target device. A tool that can answer questions like "what files can a specified process on a device access?" or "what processes on a device can access a specified file?" would be very useful for the analyst. We developed SEAndroid Live device analysis tool (SEAL)[4] for this purpose. SEAL allows different queries that take into account not only the SEAndroid policy loaded on the device, but also the actual device state, i.e. running processes and filesystem objects. SEAL offers command line and GUI interfaces, and queries the device over the adb interface. In order to obtain results about the entire device filesystem, the target device has to be either rooted or running an engineering build. Figure 1 shows the architecture of SEAL.

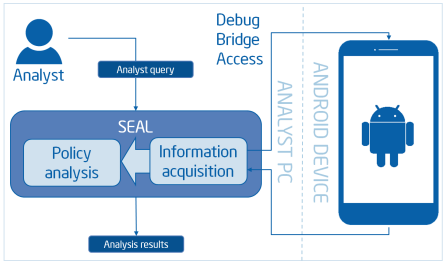

Figure 1: SEAL tool architecture

[4] github.com/seandroid-analytics/seal

4.2 Policy Decompilation Tool

One of the main problems during our manual analysis was the lack of a tool to easily identify and analyze changes that an OEM made to the default AOSP policy. Current tools like `apol` and `sediff` are not directly suited for this task, as described in Section 3.2. It would be very beneficial to have a tool able to transform a binary policy into a set of source files organized similarly to the AOSP SEAndroid source policy. In this case, it would not only be possible to perform manual analysis in a more organized and convenient manner, but also to employ standard text manipulation tools to compare or filter needed information.

4.3 Policy Visualization Tool

SELinux has the notion of `attributes`, that allow organizing policy types and domains in a hierarchical manner. This is a very powerful mechanism that can easily be misconfigured by mistake. The data collected in section 3.1 showed that most OEMs do not create new policy attributes, perhaps due to the perceived complexity of the attribute mechanism. A tool to visualize hierarchies induced by attributes may help an analyst better understand and make use of attributes effectively.

4.4 Policy Analyzer

The hardest part of our manual analysis was identifying rules that are either potentially dangerous or possibly unnecessary. The analysis can be automated with a set of heuristic checks. The tool can also utilize SEAL in order to make policy queries with regards to the device state.

Let us consider the following rule from section 3.2.4 and explain how it can be automatically detected as suspicious:

```
allow untrusted_app tee_exec : file {read
        getattr execute open}
```

The tool first uses SEAL to fetch from a device all files with the `tee_exec` label. Then it queries SEAL for all labels and DAC permissions of all higher-level directories on the path to each `tee_exec` file, and tries to determine if the `untrusted_app` domain can even reach the target file to perform the requested operations. If the first check passes, the analyzer can further check that each requested access control type makes sense. For example, in order for `execute` to succeed given that `execute_no_trans` access is not granted, there has to be a `type_transition` rule defined; furthermore, in order to be able to `write` or `read` a file, one would also normally need to have `open` permission. The policy analyzer can mark the rule as not functional if the checks fail.

In order to identify potentially dangerous rules, the policy analyzer can scan rules for possible additional usages of default types, mentioned in Section 3.2.1, and analyze new rules associated with sensitive types, such as `tee` or `proc_security`, or untrusted domains, such as the `untrusted_app` domain.

One use of the policy analyzer is its integration into OEMs' automatic build systems, in order to consistently verify that the policy does not contain unreachable or potentially dangerous rules, and that it is optimized with regards to the usage of attributes and types. This would provide value for OEMs, since policy additions might be made by different development teams, possibly without detailed knowledge of SEAndroid. The output of the tool can be further analyzed manually by a person with detailed knowledge of SEAndroid, in order to reject or accept suggested modifications.

5 RELATED WORK

Several tools and methods originally designed for SELinux are relevant to the new mobile environment.

The *de facto* standard for handling SELinux policies in text and binary format is the SETools library (Tresys, 2014): this contains the aforementioned `apol` and `sediff` tools, which can be used interchangeably on SELinux and SEAndroid.

Formal methods have been used for SELinux policy analysis. Gokyo (Jaeger et al., 2003) is a policy analysis tool designed to identify and resolve conflicting policy specifications. Usage of the HRU security model (Harrison et al., 1976) has been proposed as an approach to SELinux policy analysis (Amthor et al., 2011). Information flow analysis has been applied to SELinux policies (Guttman et al., 2005). These analysis methods are not SELinux-specific, and can be easily adapted to SEAndroid.

Some researchers have applied information visualization techniques to SELinux policy analysis (Clemente et al., 2012), also in combination with clustering (Marouf and Shehab, 2011). These techniques are also system-agnostic, and we may use them in future SEAndroid tools.

SELinux policy generation and refining tools are rare. Polgen, a tool for semi-automated SELinux policy generation based on system call tracing (Sniffen et al., 2006), appears to be no longer in active development. The SELinux userspace tools (SELinux Project, 2014) can generate SELinux policies. One

of these tools, `audit2allow`, is widely used to automatically generate and refine SELinux policies by converting SELinux audit messages into rules; these policies, however, are not necessarily correct, complete or secure, since the rules depend on code paths taken during execution, and there is no way to distinguish intended and possibly malicious application behavior. These tools are used both in SELinux and SEAndroid.

There has been some research in applying Domain Specific Languages (DSL) (Fowler, 2010) to SELinux policy development and verification (Hurd et al., 2009). The authors proposed a tool (`shrimp`) to analyze and find errors in the SELinux Reference Policy, similar to the `Lint` tool for C. This is similar to a tool we propose, but different in scope as it is limited to analysis of the SELinux reference policy.

The only SEAndroid-specific analysis method is based on audit log analysis with machine learning (Wang et al., 2015). This approach is completely different from what we propose, since it relies on significant volumes of data to classify rules.

6 CONCLUSIONS

In this paper we presented a number of common mistakes made by OEMs in their SEAndroid policies, suggesting potential reasons behind them. As a result of this study, we identified a number of practical tools that should help OEMs and security researchers to improve SEAndroid policies. We provided the implementation of a first tool, SEAL, and we are currently working on the rest.

REFERENCES

Amthor, P., Kuhnhauser, W., and Polck, A. (2011). Model-based safety analysis of selinux security policies. In *NSS*, pages 208–215. IEEE.

Badger, L., Sterne, D., Sherman, D., Walker, K., Haghighat, S., et al. (1995). Practical domain and type enforcement for UNIX. In *Security and Privacy*, pages 66–77. IEEE.

Bauer, M. (2006). Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, (148):13.

Clemente, P., Kaba, B., Rouzaud-Cornabas, J., Alexandre, M., and Aujay, G. (2012). Sptrack: Visual analysis of information flows within selinux policies and attack logs. In *AMT*, pages 596–605. Springer.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. (2005). Verifying information flow goals in security-enhanced Linux. *Journal of Computer Security*, 13(1):115–134.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Commun. ACM*, 19(8):461–471.

Hurd, J., Carlsson, M., Finne, S., Letner, B., Stanley, J., and White, P. (2009). Policy DSL: High-level Specifications of Information Flows for Security Policies.

Jaeger, T., Sailer, R., and Zhang, X. (2003). Analyzing integrity protection in the selinux example policy. In *USENIX Security*, page 5.

Marouf, S. and Shehab, M. (2011). SEGrapher: Visualization-based SELinux policy analysis. In *SAFECONFIG*, pages 1–8. IEEE.

Schaufler, C. (2008). Smack in embedded computing. In *Ottawa Linux Symposium*.

SELinux Project (2014). Userspace tools. `https://github.com/SELinuxProject/selinux/wiki`. Accessed: 2015-09-29.

Smalley, S. and Craig, R. (2013). Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *NDSS*, volume 310, pages 20–38.

Smalley, S., Vance, C., and Salamon, W. (2001). Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139.

Sniffen, B. T., Harris, D. R., and Ramsdell, J. D. (2006). Guided policy generation for application authors. In *SELinux Symposium*.

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., and Lepreau, J. (1999). The Flask security architecture: System support for diverse policies. In *USENIX Security*.

Tresys (2014). SETools project page. `https://github.com/TresysTechnology/setools3/wiki`. Accessed: 2015-09-29.

Wang, R., Enck, W., Reeves, D., Zhang, X., Ning, P., Xu, D., Zhou, W., and Azab, A. (2015). EASE-Android: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *USENIX Security*.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Security and Privacy*, pages 95–109. IEEE.