Policy Specialization to Support Domain Isolation

Simone Mutti Enrico Bacis Stefano Paraboschi

{ simone.mutti, enrico.bacis, parabosc } @ unibg.it
DIGIP — Università degli Studi di Bergamo, Italy

ABSTRACT

The exponential growth of modern information systems has introduced several new challenges in the management of security requirements. Nowadays, the technological scenario has evolved and the introduction of MAC models provides a better isolation among software components and reduces the damages that the malicious or defective ones can cause to the systems. On one hand it is important to confine applications and limit the privileges that they can request. On the other hand we want to let applications benefit from the flexibility given by MAC models, such as SELinux.

In this paper we show how the constructs already available in SELinux and the specialization of security domains can be leveraged to define boundaries where the applications are confined but still able to introduce sophisticated security patterns, such as application isolation and the least privilege principle. After defining the proposed model, we describe how it can be integrated into real systems through the use of examples on Android and Apache Web Server.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access control

Keywords

Mandatory Access Control; SELinux; typebounds; Android; App containerization; Policy Modularity.

1. INTRODUCTION

The widespread diffusion of information systems in the last decade has led to a considerable growth in the width of offered services. Applications become more sophisticated, extensive and operate in an increasingly open and integrated environment. This growth was followed by an increase both in terms of complexity and management of the systems itself. This increase in the systems' complexity has had an effect also on the management of the security requirements that the system has to fulfill. Firstly, the complexity of the system and the number of interconnections between elements of the system has increased the attack surface of the systems themselves. Secondly, it has increased the complexity of the security management process by system administrators.

The management of security requirements is, thus, a critical task that has the goal of avoiding possible information disclosures. A particular area of security requirement management is access control management, which focuses on defining a set of rules, called policies. To ease the definition and management of access control policies the concept of *containerization* was introduced. Containerization occurs when the operating system allows for multiple isolated user-space instances, instead of just one. The security isolation is provided by means of different mechanisms. The most prominent is represented by the use of Mandatory Access Control models. MAC models are commonly perceived as offering a significant contribution to the security of systems. MAC policies regulate accesses to data on the basis of predefined classification of subjects and objects in the system.

In this paper we define a model to describe the relationships between SELinux subjects, their permissions and their boundaries. SELinux permits to limit the privileges that a subject can obtain over the system objects using the *typebounds* bounding mechanism that will be discussed in Section 4. This can be used by system administrators to define safe policy areas where applications can be confined[1].

Using our model, we show how this mechanism can also be used to specialize the policy and generate different isolated areas where the applications are confined but still able to define their SELinux domains in order to adopt sophisticated security patterns. Our approach shows how it is possible to combine confinement [4] and specialization in SELinux.

Outline

Section 2 illustrates the related work of this paper. Section 3 summarizes how SELinux works and defines the model that will be used in the subsequent sections. Section 4 shows how the proposed model can describe the bounding mechanism offered by the SELinux *typebounds* rule and its implications. Section 5 shows how this idea can be used to improve real world systems, with references to Android and to Apache Web Server. Section 6 draws some concluding remarks.

[1]Since threads can only be assigned domains that are bounded by the parent process domain, this mechanism is also used to limit the privileges of threads. The case of Apache Plus, which limits the privileges of its workers will be discussed in Section 5.2

SafeConfig'15, October 12, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2809826.2809832.

2. RELATED WORK

Different models have been proposed to formalize SELinux policies [13, 6]. Hicks et al. [5] analyze the SELinux Multi-Level/Multi-Category Security, the standard way used to create multiple isolated instances of the same policy domains, which, however, does not permit their specialization.

Flaskdroid by Bugiel et al. [3], permits to specify Android app policies, relying on other software components besides SELinux. Our solution is based solely on SELinux, so it can be directly used in any system that adopts it, without imposing any additional performance overhead.

To the best of our knowledge, this is the first attempt to formally define the SELinux *typebounds* rule and illustrate how it can be used not only to confine the policy domains but also to specialize them.

3. THE SELINUX MODEL

SELinux [8, 12] is one of the most widely adopted implementations of Mandatory Access Control. SELinux policies are expressed at the level of *security context* (also known as *security label* or just *label*). SELinux requires a security context to be associated with every process (or subject) and resource (or object). The label is used to decide whether access is allowed or not as defined by the policy. Every request that a process generates to access a resource will be accepted only if it is authorized by both the classical DAC access control service and by the SELinux policy. The advantages of SELinux compared to the DAC model are its flexibility (the design of Linux assumes a *root* user that has full access to DAC-protected resources) and the fact that process and resource labels can be assigned and updated in a way that is specified at system level by the SELinux policy.

In this paper we propose a model that can be used to represent an SELinux policy and allows to better characterize our approach. Its basic elements are:

type : represents an identifier that can be used to describe both the subject and the target of an authorization; a *type* denotes a security domain or the profile of a process or resource in the system; the *type* is used to build labels for processes and resources.

class : represents the kind of resource (e.g., file, process) that will be the target of an authorization; an implementation of SELinux in a system will have to provide in its setup a set of classes consistent with the variety of resources that the system is able to manage.

permission : represents the possible actions a source can apply on a target of a specific *class*, specified in the setup of SELinux; every *class* has its own set of permissions, represented by *class.permissions* (e.g., *file.permissions = read, write, execute, . . .*).

A set of *types* $\mathbb{T}$ can be grouped into an *attribute*. Formally, an attribute statement declares an identifier that can then be used to refer to a group of type identifiers.

Definition 1. Given the definition of types, classes, permissions and attributes, we introduce the concept of *authorization*. Considering a set $\mathbb{L} = \mathbb{T} \cup \mathbb{A}$ of labels (where $\mathbb{T}$ represents the set of types and $\mathbb{A}$ represents the set of attributes), a set $\mathbb{C}$ of classes, and a set $\mathbb{P}$ of permissions, an *authorization* is a quadruple $\langle \sigma, \tau, \gamma, \alpha \rangle$, where:

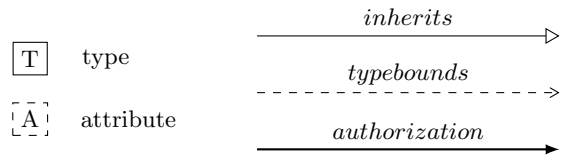
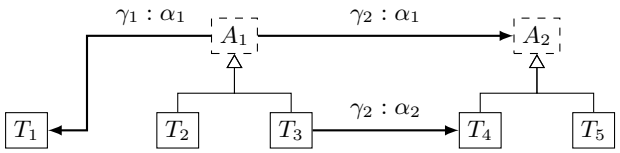
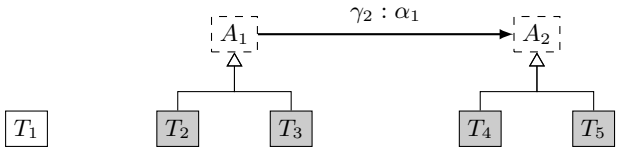


Figure 1: Graphical syntax used in the SELinux model.



(a) Graphical representation of three authorizations.



(b) Policy Region related to $\langle A_1, A_2, \gamma_2, \alpha_1 \rangle$.

Figure 2: Graphical representation of a policy.

$\sigma \in \mathbb{L}$ is the label used as source of the authorization (the security principal of the authorization);

$\tau \in \mathbb{L}$ is the label used as target (associated with the object that is accessed by σ);

$\gamma \in \mathbb{C}$ is the class of resource accessed in the operation;

$\alpha \in \{\mathbb{P} \cap \gamma.\alpha\}$ is the specific *permission*, which has to be compatible with the *class*.

Authorizations can be grouped into four types of AV rules: *allow*, *dontaudit*, *auditallow*, and *neverallow*.

Definition 2. A *policy* is a collection of tuples $\langle$*ruletype*, *authorization*$\rangle$ where *ruletype* is one of the AV rule types.

Without loss of generality, we consider the case of an SELinux policy containing only *allow* AV rules.[2]

Definition 3. An attribute A may contain one or more types and this is represented by the $containedTypes : \mathbb{A} \rightarrow 2^{\mathbb{T}}$ function, where $\mathbb{A}$ and $\mathbb{T}$, respectively, represent the set of attributes and the set of types.

Example 1. *Figure 2a shows a graphical representation (see Figure 1 for the graphical syntax) of the three authorizations* $\langle A_1, T_1, \gamma_1, \alpha_1 \rangle$, $\langle A_1, A_2, \gamma_2, \alpha_1 \rangle$ *and* $\langle T_3, T_4, \gamma_2, \alpha_2 \rangle$.

Definition 4. A label is said *primitive* iff the output of the *containedTypes* function is the label itself (i.e., it is a type). Furthermore, an authorization $auth = \langle \sigma, \tau, \gamma, \alpha \rangle$ is said *primitive* iff σ and τ are primitive elements.

Without loss of generality, we can express access control decisions only in terms of primitive authorizations.

[2]It is to note that (i) *neverallow*, (ii) *dontaudit* and (iii) *auditallow* rules can be left out, because (i) may only cause the rejection of the policy by the compiler, (ii) and (iii) describe the configuration of the auditing services.

Definition 5. The function $primitives(auth)$, given an authorization $auth$, can be used in order to obtain the set of equivalent primitive authorizations:

$$primitives(\ \langle \sigma, \tau, \gamma, \alpha \rangle\ ) = \{\ \langle \tilde{\sigma}, \tilde{\tau}, \gamma, \alpha \rangle\ |\ \tilde{\sigma} \in containedTypes(\sigma),\ \tilde{\tau} \in containedTypes(\tau)\ \}$$

A *primitive policy* $\tilde{P}$, containing only primitive authorizations, can be obtained from a policy P as:

$$\tilde{P} = \bigcup_{auth\ \in\ P} primitives(auth)$$

Definition 6. Regions: given a primitive policy $\tilde{P}$ and a source σ_i, we define the *sourceRegion* $R_\sigma(\sigma_i)$ as the set of all the primitive authorizations involving σ_i as source. Respectively, *targetRegion* $R_\tau(\tau_i)$, represents the set of all the primitive authorizations involving τ_i as target. Furthermore, we can compute the *Region* $R(\sigma_i, \tau_i)$ containing all the primitive authorizations that have σ_i as source and τ_i as target.

EXAMPLE 2. *Given the authorization* $\langle A_1, A_2, \gamma_2, \alpha_1 \rangle$ *(see Figure 2b) the output of* $primitives(\langle A_1, A_2, \gamma_2, \alpha_1 \rangle)$ *returns the following primitive authorizations:*

source	***target***	***class***	***permission***
T_2	T_4	γ_2	α_1
T_2	T_5	γ_2	α_1
T_3	T_4	γ_2	α_1
T_4	T_5	γ_2	α_1

An alternative way to represent the primitive policy $\tilde{P}$ *derived from the policy in Figure 2, is the following matrix* M*:*

σ \ τ	T_1	T_2	T_3	T_4	T_5
T_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
T_2	$\gamma_1 : \alpha_1$	$\emptyset$	$\emptyset$	$\gamma_2 : \alpha_1$	$\gamma_2 : \alpha_1$
T_3	$\gamma_1 : \alpha_1$	$\emptyset$	$\emptyset$	$\gamma_2 : \alpha_1$ $\gamma_2 : \alpha_2$	$\gamma_2 : \alpha_1$
T_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
T_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

It is to note that $M_{[T_3][...]}$ *represents the* sourceRegion $R_\sigma(T_3)$*,* $M_{[...][T_4]}$ *represents the* targetRegion $R_\tau(T_4)$ *and* $M_{[T_3][T_4]}$ *represents the* Region $R(T_3, T_4)$*.*

4. THE TYPEBOUNDS RULE

The *typebounds* rule is used to define that a bounded type must have a subset of the privileges of its bounding type. The official SELinux Documentation [10] defines the rule as:

The typebounds rule was added in version 24 of the policy. This defines a hierarchical relationship between domains where the bounded domain cannot have more permissions than its bounding domain (the parent).

This definition does not provide a full formalization. For example it is not immediately clear what should be checked when both the source type σ and the target type τ are typebounded. In this section we extend the model presented in Section 3 to include the *typebounds* rule. After analyzing the implications of this extension, we propose a schema to separate and isolate the domains of different applications while keeping the confinement in place.

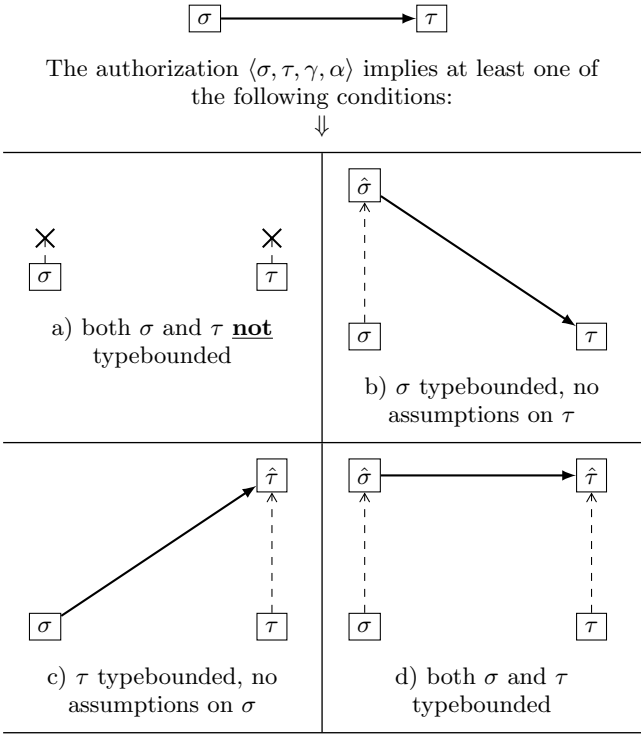


Table 1: The four conditions that can support the authorization $\langle \sigma, \tau, \gamma, \alpha \rangle$. In order to simplify the schema presented in Figure 2 we consider that the class γ and the action α are fixed for all the authorizations in the schema. Making no assumption on a type means that the condition is valid both when the type is typebounded and when it is not.

It is useful to emphasize that:

- a type can have at least zero and at most one *bounding* (parent) type;
- the *typebounds* rule does not automatically assign any authorization to the bounded type; it just sets the upper bound of the bounded type privileges to those held by the bounding type;
- an exception is raised at compile time if an allow rule tries to assign a privilege to a bounded type not given to its bounding type (if present). Additional checks are performed when the policy changes at runtime.

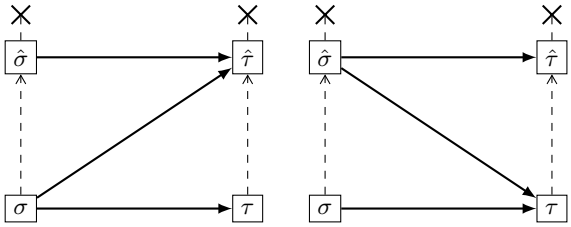
Before introducing Theorem 1, we need some additional definitions in order to deal with the hierarchy imposed by the *typebounds* rule.

Definition 7. Let $\sigma \in \mathbb{T}$, then $\pi(\sigma)$ is the parent of σ, if it exists (i.e., if the policy defines *typebounds* $\hat{\sigma}$ σ)[3].

Definition 8. Let σ and τ be types and γ a class. Then $\varphi(\sigma, \tau, \gamma)$ is the set $\{\alpha_i, \alpha_j, \ldots\}$ of actions included in $R(\sigma, \tau)$, associated with class γ.

It is to note that the use of the primitive policy permits to compute regions that are immediately usable in the evaluation of the respect of the typebounds rules.

[3]The same considerations can be applied to τ.



(a) $\sigma \to \tau$ can be satisfied by $\sigma \to \hat{\tau}$ (Equation 1c). In turn $\sigma \to \hat{\tau}$ requires $\hat{\sigma} \to \hat{\tau}$ (Equation 1b).

(b) $\sigma \to \tau$ can be satisfied by $\hat{\sigma} \to \tau$ (Equation 1b). In turn $\hat{\sigma} \to \hat{\tau}$ requires $\hat{\sigma} \to \hat{\tau}$ (Equation 1c).

Figure 3: $\hat{\sigma} \to \hat{\tau}$ is a necessary condition for $\sigma \to \tau$ when σ and τ are both typebounded while $\hat{\sigma}$ and $\hat{\tau}$ are not.

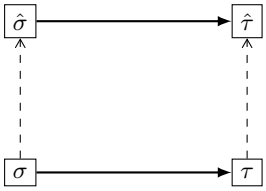


Figure 4: $\hat{\sigma} \to \hat{\tau}$ is a sufficient condition for $\sigma \to \tau$ when σ and τ are both typebounded. The domains σ and τ do not need to share any authorization with $\hat{\sigma}$ and $\hat{\tau}$, so these domains are isolated from each other and no information leakage can happen.

We now have all the elements to express the theorem that indicates whether a policy satisfies the hierarchical checks.

THEOREM 1. *An SELinux policy P satisfies the* typebounds *rules iff* $\forall \sigma, \tau \in \mathbb{T}, \forall (\gamma, \alpha) \in R(\sigma, \tau)$*:*

$$(\pi(\sigma) \textit{ is undefined} \wedge \pi(\tau) \textit{ is undefined}) \vee \quad (1a)$$

$$(\hat{\sigma} = \pi(\sigma) \wedge \alpha \in \varphi(\hat{\sigma}, \tau, \gamma)) \vee \quad (1b)$$

$$(\hat{\tau} = \pi(\tau) \wedge \alpha \in \varphi(\sigma, \hat{\tau}, \gamma)) \vee \quad (1c)$$

$$(\hat{\sigma} = \pi(\sigma) \wedge \hat{\tau} = \pi(\tau) \wedge \alpha \in \varphi(\hat{\sigma}, \hat{\tau}, \gamma)) \quad (1d)$$

Table 1 shows the four conditions that can support the authorization $\langle \sigma, \tau, \gamma, \alpha \rangle$ whether σ and τ are typebounded or not. We analyze these conditions in order to show that they satisfy Theorem 1:

- In Table 1.a, neither σ nor τ are typebounded, thus any authorization, including $\langle \sigma, \tau, \gamma, \alpha \rangle$, would satisfy the check imposed by Theorem 1 (Equation 1a).
- In Table 1.b, $\pi(\sigma) = \hat{\sigma}$, thus $\langle \hat{\sigma}, \tau, \gamma, \alpha \rangle$ satisfies Theorem 1 for $\langle \sigma, \tau, \gamma, \alpha \rangle$ (Equation 1b).
- In Table 1.c, $\pi(\tau) = \hat{\tau}$, thus $\langle \sigma, \hat{\tau}, \gamma, \alpha \rangle$ satisfies Theorem 1 for $\langle \sigma, \tau, \gamma, \alpha \rangle$ (Equation 1c).
- In Table 1.d, $\pi(\tau) = \hat{\tau}$ and $\pi(\sigma) = \hat{\sigma}$, thus $\langle \hat{\sigma}, \hat{\tau}, \gamma, \alpha \rangle$ satisfies Theorem 1 for $\langle \sigma, \tau, \gamma, \alpha \rangle$ (Equation 1d). It is to note that Theorem 1 can also be satisfied by $\langle \hat{\sigma}, \tau, \gamma, \alpha \rangle$ (Equation 1b) or $\langle \sigma, \hat{\tau}, \gamma, \alpha \rangle$ (Equation 1c).

An application can create new typebounded types in order to specialize the ones provided by the policy and introduce sophisticated security patterns among them. Some of the most common patterns are shown in Section 5.

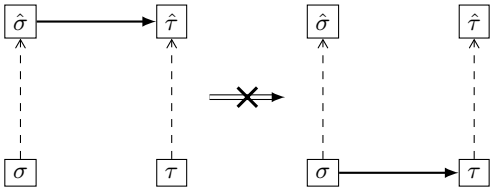


Figure 5: The typebounds rule does not imply inheritance from the parent types to the children types.

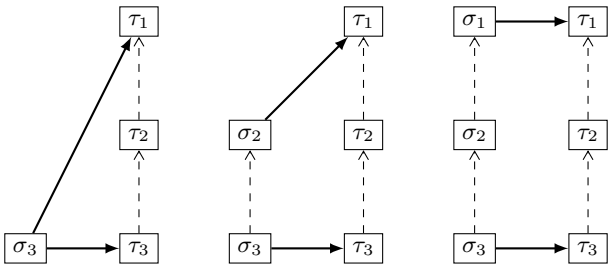


Figure 6: The typebounds rule is not transitive, so these schemas **do not** satisfy Theorem 1.

The most comprehensive schema is the one where $\hat{\sigma}$ and $\hat{\tau}$ are domains provided by the base policy, while σ and τ are the specialized ones created by the application.

As explained in Figure 3, when σ and τ are both typebounded (respectively by $\hat{\sigma}$ and $\hat{\tau}$), and $\hat{\sigma}$ and $\hat{\tau}$ are not typebounded, then $\langle \hat{\sigma}, \hat{\tau}, \gamma, \alpha \rangle$ is a necessary condition for the authorization $\langle \sigma, \tau, \gamma, \alpha \rangle$. The authorization $\langle \hat{\sigma}, \hat{\tau}, \gamma, \alpha \rangle$ is also a sufficient condition for $\langle \sigma, \tau, \gamma, \alpha \rangle$ as shown in Table 1.d and Figure 4.

THEOREM 2. *When $\pi(\sigma) = \hat{\sigma}$, $\pi(\tau) = \hat{\tau}$, $\pi(\hat{\sigma})$ is undefined and $\pi(\hat{\tau})$ is undefined, the authorization $\langle \hat{\sigma}, \hat{\tau}, \gamma, \alpha \rangle$ is a necessary and sufficient condition for satisfying the hierarchy checks of the authorization $\langle \sigma, \tau, \gamma, \alpha \rangle$.*

Theorem 2 and Figure 4 indicate how it is possible to specialize the domains offered by the system policy without the need of granting the system domains privileges to the newly defined ones. The result is that the new domains can be isolated and protected from the system and from other applications. This example shows how it is possible to combine confinement and specialization using the typebounds rule.

4.1 Typebounds $\not\Rightarrow$ inheritance

As already stated in Section 4, the typebounds rule does not assign any authorization, it just sets a confinement for the children types (as shown in Figure 5), so the typebounds rule is also *not transitive*. This is important to note because it may appear counterintuitive. In fact, similar layered models imply inheritance and thus transitiveness.

EXAMPLE 3. *Let σ_3, τ_1, τ_2 and τ_3 be four types such that $\pi(\tau_3) = \tau_2$ and $\pi(\tau_2) = \tau_1$. The authorization $\langle \sigma_3, \tau_1, \gamma, \alpha \rangle$ is not sufficient to satisfy any condition of Theorem 1 for authorization $\langle \sigma_1, \tau_1, \gamma, \alpha \rangle$. In order to comply with the hierarchy check, $\langle \sigma_1, \tau_2, \gamma, \alpha \rangle$ must also be assigned.*

Figure 6 shows the schema in Example 3 and some other schemas that do not satisfy Theorem 1.

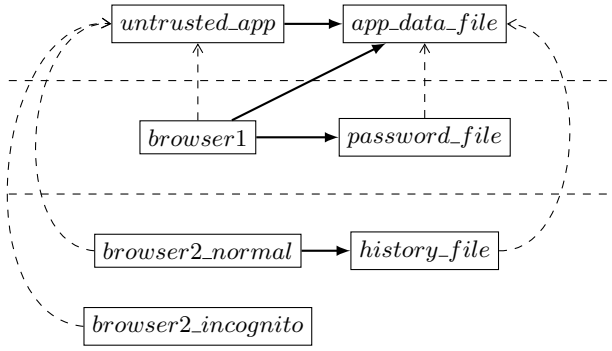


Figure 7: Isolation of two applications using typebounds.

5. USE CASES

In this Section we present scenarios that can benefit from the isolated domains described in the previous section. Furthermore, an example on how the approach proposed in this paper can be used in emerging scenario such as *Docker* is presented in [1].

5.1 Android M

The current design of SELinux in Android (i.e., SEAndroid [11]) aims at protecting system components and trusted apps from abuses by third-party apps. All the third-party apps fall within a single *untrusted_app* MAC domain and an app interested in getting protection from other apps or from internal vulnerabilities can only rely on Android Permissions Framework (APF) and the Linux DAC support.

A step ahead in the direction of per app policy customization is represented by Android M. In the M release each user is assigned a specific SELinux category, building a Multi Category Security (MCS) model. MCS works like the DAC extended attributes: users are assigned to categories and they can apply these categories at their discretion to the content that they own. This means that Android can create different *worlds* (i.e., use different categories) in order to separate apps belonging to different users. The introduction of categories brings several advantages in terms of flexibility. The categories, in fact, permit to have multiple instances of the same type. A process is allowed to perform an action only when the type enforcement *allow* rules are satisfied and the target has a category that is compatible with the categories assigned to the source. Although the categories permit to define a more fine-grained policy, they do not provide a mechanism to drop privileges when not needed. This is a significant limitation, since apps can get a concrete benefit from the specification of their own policy. The *appPolicyModules* [2] proposal implements in Android the model detailed in this paper. It improves the definition and enforcement of the security requirements associated with each app through the use of specialized types.

5.1.1 AppPolicyModules in Android

In general, with the availability of *appPolicyModules*, the system could evolve from a scenario where each app is given at installation time access to the whole *untrusted_app* domain at the SELinux layer, to a scenario where each app is associated with the portion of *untrusted_app* domain that is really needed for its execution, with a better support of the classical "least-privilege" security principle. It is to note that Android M provides the feature to drop Android permissions at runtime. This confirms that *appPolicyModules* and more generally domain specialization identify a concrete need and that Android is evolving in this direction.

EXAMPLE 4. *Most mobile browsers (e.g., Chrome, Firefox) store confidential information such as* usernames *and* passwords *in a SQLite database. Following Google's best practices for developing secure apps, the password database is saved in the app data folder, which should be accessible only to the app itself. However, this is not enough to protect the password database by other apps with root privileges.*

The use of MAC support offers protection even against threats coming from the system itself, like a malicious app that abuses root privileges. The app can protect its resources from other apps, specifying its own types and defining in a flexible way which system components may or may not access the domains introduced by the APM.

Figure 7 shows an example where the untrusted_app *domain does not hold any permission on the file labeled as* password_file*, which is accessible only by the* browser1 *domain.*

It is to note that both *browser1* and *password_file* are typebounded, thus *browser1* is not violating any restriction defined on the parent domain (i.e., *untrusted_app*) according to Section 4. Greater flexibility derives from the possibility to freely manage privileges for internal types over internal resources, building a MAC model that remains completely under the control of the app.

EXAMPLE 5. *Most mobile browsers provide the* incognito mode *(i.e., anonymous surfing), which allows the user to surf the web without storing permanently the history and the cookies, and in general aiming at leaving no traces of the browsing session in persistent memory.*

In order to enhance its security and protect the user even from possible app flaws, the appPolicyModule could specify a switch of context (i.e., it may change the SELinux domain associated with its process) when the user enters the incognito mode. In Figure 7, lower part, the domain browser2_normal *can read and write all the files labeled as* history_file*, while the domain* browser2_incognito*, used during anonymous surfing, drops the privilege of writing the files, preventing the leakage of resources that may leave a trace of the navigation session.*

As for the example before, the policy is correct due to the *typebound* statement between *browser2_normal*, *browser2_incognito* and *untrusted_app*.

5.2 Apache web server and SELinux

Another scenario where the model presented in this paper could be adopted is the Apache Web Server. Current web servers with three-layer architectures (i.e., with a browser working as a front-end to a Web Server accessing data stored on a DBMS) assume that user privileges on accessing the data are described in the Web application, which has continuous privileged access to the DBMS, with full control on all the data accessible by the user. In emerging scenarios such as cloud services the developer controls only a part of the entire infrastructure thus a mechanism providing isolation among web applications and also among different services offered by the same web application is desirable.

To meet these new security requirements, modern Web Servers can rely on the use of Mandatory Access Control provided directly by the OS. This design provides a characteristic feature called system wide consistency in access control, because all the decisions are made by the SELinux security server based on a single declarative policy.

Apache is a multi-processed program structured as a pool of *worker* threads monitored and controlled by a *master* process. The workers are responsible for handling the communications with the clients. A worker handles at most one connection at a time until the connection is terminated. Because the worker performs on behalf of the user, it is necessary to grant the correct set of privileges, i.e., assign the right security context. The Apache Web Server is supported by SELinux using the additional module *mod_selinux*. It enables to assign a restrictive security context based on *HTTP authentication* prior to the invocation of content handlers that include the entry points of web applications.

When the web server process receives an HTTP request, its request headers are parsed and analyzed, then HTTP authentication is applied if necessary. It is to note that the authentication is done under the web server's context. Next, the *mod_selinux* module spawns a one-time worker thread, while the parent waits for its completion. The worker modifies its own security context prior to the invocation of content handlers, including entry points of web applications.

The typebounds rule confines the scope of privileges to be allowed on the bounded domain (i.e., the worker). When a domain is bounded by another one, any privilege can never be assigned to the bounded domain as far as the bounding one is allowed. We can consider the bounded domain as a special state of the original domain that lacks a part of privileges. In fact, SELinux allows to change the security context of a thread as far as the new security context is bounded by the older one.

Apache/SELinux Plus [7, 9] uses the bounded domain to assign an individual security context on a certain thread, without unnecessary privileges for the authenticated users. The current schema can enforce the least privilege principle, while the model discussed in Section 4 allows the use of more sophisticated security patterns. The workers could enter different specialized domains based on some connection details, while still being typebounded by the Apache master process domain.

6. CONCLUSIONS

Information systems are becoming more difficult to manage, with the integration of resources of different owners, and access offered to a larger variety of users. Service oriented architectures facilitate this evolution. Access control policies follow this trend. The use of solutions like SELinux further increases the role that access control policies have in the security of modern operating systems. The efficient management of the policies, with the ability to containerize different services promises to lead to significant benefits in a large number of application scenarios. The paper highlights the benefit of the use of isolated domains by means of the *typebound* statement. The extensive reuse of SELinux constructs demonstrates the flexibility of SELinux and facilitates the deployment of the proposed solution.

7. ACKNOWLEDGEMENTS

This work was partially supported by a Google Research Award (winter 2014), by the Italian Ministry of Research within the PRIN project "GenData 2020" and by the EC within the 7FP and H2020 program, respectively, under projects PoSecCo (257129) and EscudoCloud (644579).

8. REFERENCES

[1] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi. DockerPolicyModules: mandatory access control for docker containers. In *IEEE CNS 2015 Poster Session (to appear)*, Florence, Italy, 2015.

[2] E. Bacis, S. Mutti, and S. Paraboschi. AppPolicyModules: Mandatory Access Control for Third-Party Apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 309–320. ACM, 2015.

[3] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 131–146, Berkeley, CA, USA, 2013. USENIX Association.

[4] A. Crowell, B. H. Ng, E. Fernandes, and A. Prakash. The Confinement Problem: 40 Years Later. *JIPS*, 9(2), 2013.

[5] B. Hicks, S. Rueda, L. St Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. *ACM Transactions on Information and System Security (TISSEC)*, 2010.

[6] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, pages 5–5. USENIX Association, 2003.

[7] K. Kohei. Introduction of the Apache/SELinux plus. `https://code.google.com/p/sepgsql/wiki/Apache_SELinux_plus` . Accessed: 2015-07-30.

[8] F. Mayer, K. MacMillan, and D. Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, NJ, USA, 2006.

[9] SELinux Project. Apache SELinux Support. `http://www.selinuxproject.org/page/NB_Apache` . Accessed: 2015-07-30.

[10] SELinux Project. Bounds Rules. `http://www.selinuxproject.org/page/Bounds_Rules` . Accessed: 2015-07-30.

[11] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Network and Distributed System Security Symposium (NDSS 13)*, 2013.

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

[13] G. Zanin and L. V. Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 136–145. ACM, 2004.