POLITECNICO
MILANO 1863

School of Industrial and Information Engineering
Master of Science in Computer Science and Engineering

SELinux policies for fine-grained protection of Android apps

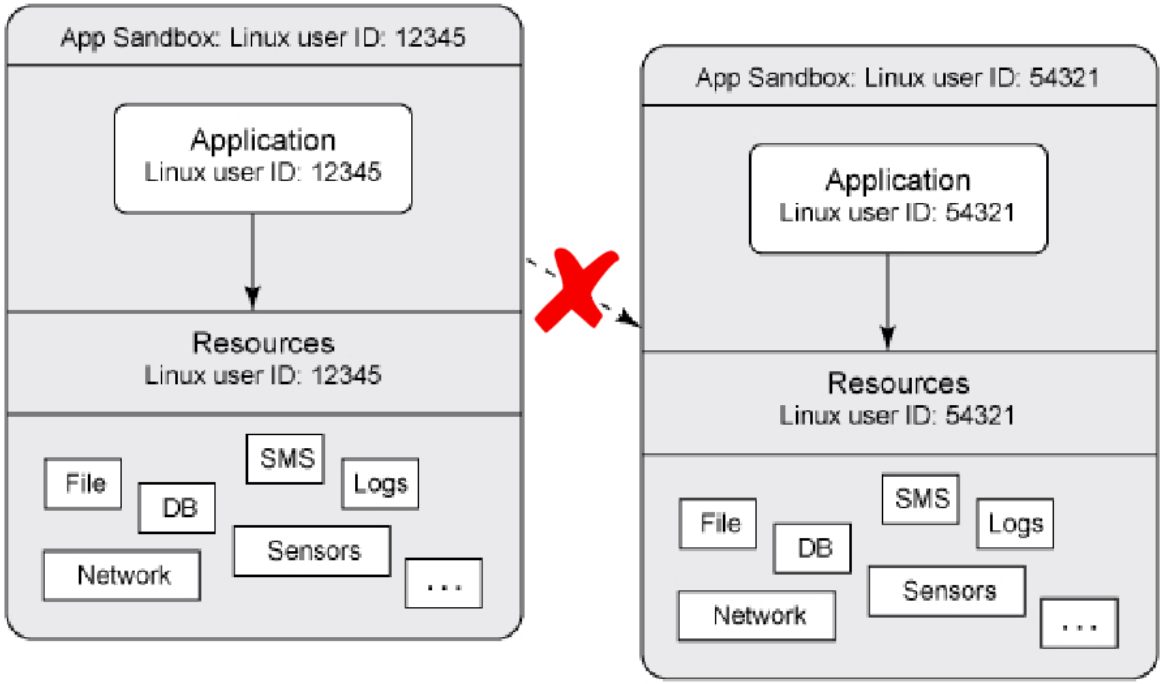
Master's thesis by: Matthew Rossi
Supervisor: Prof. Stefano Paraboschi

Academic Year 2017-2018

DAC in Android

In Android each app receives a dedicated User IDentifier (UID) and Group Identifier (GID) at install-time.

These identifiers are used to set the user and the group owner of the resources installed by the app in the default data directory. This confinement of the data folders permits to enforce a strict **isolation** from other applications.



Mandatory Access Control and SELinux

An access control model by which the opearting system constraints the ability of a subject to perform some sort of operation on an object.

In practice:

- a **subject** is usually a process or thread
- an **object** is a file, directory, TCP/UDP port, shared memory segment, IO device, etc.

Security-Enhanced Linux (**SELinux**) is a MAC implementation that, whenever a subject attempts an operation on an object, examines subject and object security attributes and operation to decide whether the operation can take place according to the set of authorization rules (aka **policy**).

SELinux operates on the principle of **default denial**, which means that every single operation must be explicitly allowed by the policy, with:

```
allow subject_types object_types:classes permissions;
```

SELinux in Android

Android uses a variation of SELinux to constrain **privilege escalation** and make it harder for attacker to successfully exploit vulnerabilities.

The current design aims at protecting the **system components** and **trusted apps** from abuses by third-party apps.

All the third-party apps fall within a single **untrusted_app** domain and an app interested in getting protection from other apps or from internal vulnerabilities can only rely on the Linux DAC support.

```
matt@Home:~$ adb shell ls -lZ data/data
total 688
drwx------ 4 system    system    u:object_r:system_app_data_file:s0       android
drwx------ 4 u0_a11    u0_a11    u:object_r:app_data_file:s0:c512,c768    android.ext.services
drwx------ 4 u0_a43    u0_a43    u:object_r:app_data_file:s0:c512,c768    android.ext.shared
drwx------ 4 u0_a24    u0_a24    u:object_r:app_data_file:s0:c512,c768    com.android.apps.tag
drwx------ 4 u0_a1     u0_a1     u:object_r:app_data_file:s0:c512,c768    com.android.backupconfirm
drwx------ 4 u0_a34    u0_a34    u:object_r:app_data_file:s0:c512,c768    com.android.bips
drwx------ 4 bluetooth bluetooth u:object_r:bluetooth_data_file:s0        com.android.bluetooth
drwx------ 4 u0_a32    u0_a32    u:object_r:app_data_file:s0:c512,c768    com.android.bluetoothmidiservice
```

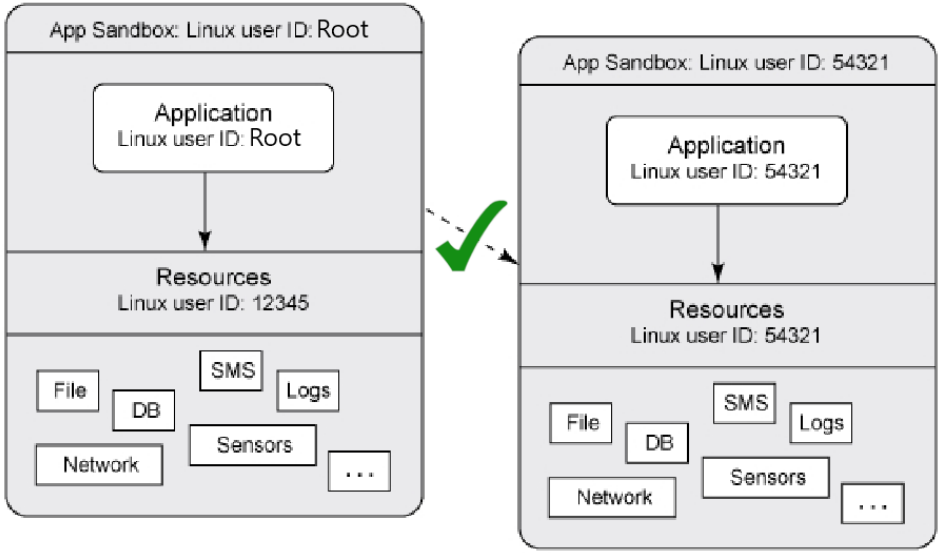
This is a significant limitation, since apps can get a concrete benefit from the specification of their own policy.

Threat model - Bypassing DAC

In Android app isolation is only enforced at DAC level, but this is not enough to protect the app and its own resources by other apps with **root** privileges.

Apps can gain root privileges in two ways:

- with the **user intervention**: the user is aware of the fact that some apps act as root, however she does not know how these privileges are used and she has to trust the app.
- **exploiting a bug** in a system component: the user is unaware of the fact that an app acts as root



Solution

To provide a solution to both scenarios the **appPolicyModules** methodology has been implemented.

AppPolicyModules allows developers to define app **specific** policy modules to be installed alongside the app.

This methodology offers three important benefits to apps:

- **protection from external threats** with app isolation enforced with MAC
- **reduction of the attack surface** since developers, knowing their specific app functionalities, can further tighten SELinux policy to the **least privileges** needed by the app to correctly run
- **policy flexibility** as multiple domains with different privileges can be defined to match app modes of operations

Solution - Requirements

Analyzing the introduction of appPolicyModules, we need to consider the different cases that emerge from the combination of the system policy and an appPolicyModule.

Each **allow rule**, has two types involved: a **source** and a **target** type.

These types may be defined in either the **system policy** or the **appPolicyModule**.

We then have four types of allow rule, depending on the origin of the source and target domains.

Two of these configurations are problematic and are not allowed to happen in appPolicyModules.

Solution - No impact on system policy

allow unstrusted_app system_data_file:file rw_file_pe

Since **third-party apps** can **not** be **trusted** a priori, it is imperative that the provided appPolicyModule must not be able to have an impact on privileges where source and target are system types.

To enforce this all the allow rules listed in the appPolicyModule must specify **as source a new type**, guaranteed to be outside of the system policy.

Solution - No escalation

allow my_domain system_data_file:file rw_file_pe 

New domains declared in an appPolicyModule must always operate **within** the **boundaries** defined by the system policy as acceptable for the execution of apps.

When a new application is installed, its domain has to be created under the **untrusted_app** domain, so the system policy can flexibly define the maximum allowed privileges for third-party apps.

To enforce this a **typebounds statements** must be used on all the types that appear as source in an allow statement of the appPolicyModule.

```
typebounds untrusted_app my_domain;
```

The policy compiler will then verify the satisfaction of the bounding relation.

Implementation - device specific

On Android the SEPolicy comes in two variants:

- **monolithic** (legacy non-treble devices): a binary policy file located at `/sepolicy`
- **split** (treble devices):
  - platform portion of the source policy in the system partition
  - non-platform portion of the source policy in the vendor partition

Our testing device, the Nexus 6P, does not support treble, so we made some changes to allow on device policy compilation.

1. Modify the build system to put the split policy and policy compiler in the system image
2. Add the non-platform portion of the source policy in the precompiled vendor image
3. Mount system and vendor at `init`'s first stage so it can load SELinux policy fragments (add `fstab` entries to the Device Tree Blob attached to the kernel)

Implementation - install/uninstall app

To successfully install and uninstall apps with their specific policy files some **extra steps** need to be performed by the PackageManagerService.

Install

1. Extract the policy files from the APK
2. Validate the policy against the appPolicyModule requirements (`PolicyLexer`, `PolicyParser`)
3. Compile all source policy files on the device with the one provided by the new app and load the produced binary policy into the kernel (JNI call to `installd`)
4. Store the policy files to `/data/selinux/<package_name>` to make them available to the underlying SELinux library (`libselinux`) and future policy compilation

Uninstall

5. Remove the `/data/selinux/<package_name>` directory
6. Compile all source policy files remaining on the device and load the outcome into the kernel

Implementation - API

We built a new Android SDK to extend app's API with some SELinux specific methods to guarantee apps control over their own domain and files security contexts. Here the most important ones will be described.

setcon mimics the libselinux function Zygote calls to set the domain of newly spawn app.

With this the app can transition to a domain among the ones specified inside its policy.

```
public static native boolean setContext (String);
```

restorecon restores a file security context to the one specified in the app's file_contexts.

With this the app can correctly assign security contexts to its files and protect them according to the specified policy.

```
public static native boolean restoreContext ();
```

Implementation - SEPolicy

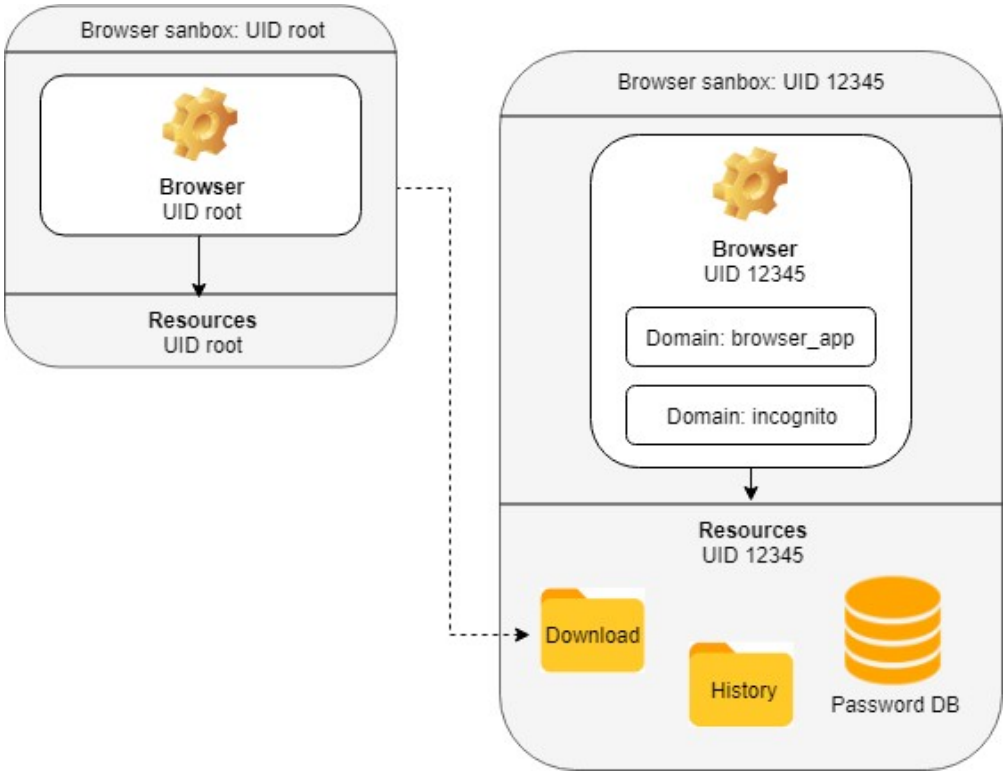
The previously described modifications change how some processes function, as a consequence adaptations to the **platform policy** are needed in order for the solution to correctly work. Here the most important ones will be described.

Installd

- transition `installd` to `secilc` on execution of `secilc_exec` to grant policy access only to secilc execution
- allow `installd` to load the policy into the kernel

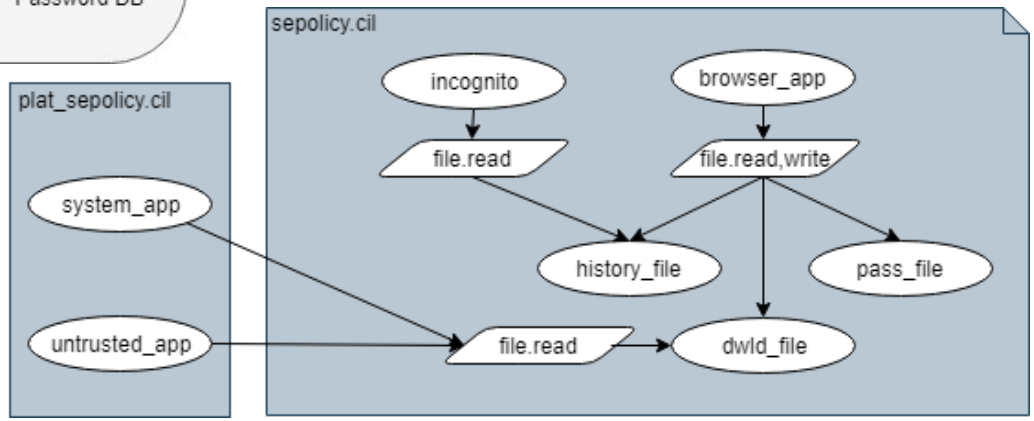
Untrusted app requires some more privilegies to be able to perform `restorecon` and `setcon` functions. On a production release this should be done with a system service to avoid granting all this permission to third party apps.

Example - browser incognito mode and password manager



To enforce the incognito mode at MAC level, the app **transitions** into the incognito domain dropping the privilege of writing the files, preventing the leakage of resources that may leave a trace of the navigation session.

Browser can **grant** apps the privilege to read the downloaded files, while **preventing** the access to the usernames and passwords database.



Thank you for your attention!

Any questions?