Security

Contents

- Terminology . . . 3
  - Privilege . . . 3
  - Trust . . . 3
  - Integrity, confidentiality and availability . . . 3
- Security boundaries and threat model . . . 4
  - Security between applications . . . 4
  - Communication between applications . . . 5
  - Security between users . . . 6
  - Security between platform services . . . 6
  - Security between the device and the network . . . 7
  - Physical security . . . 7
- Solutions adopted by popular platforms . . . 7
  - Android . . . 8
  - Bada . . . 9
  - iOS . . . 10
- Mandatory Access Control . . . 11
  - Linux Security Modules (LSM) . . . 11
  - Comparison . . . 15
  - Performance impact . . . 17
  - Conclusion . . . 18
- polkit (PolicyKit) . . . 24
  - Motivation for polkit . . . 24
  - polkit's solution . . . 26
  - Recommendation . . . 27
- Resource Usage Control . . . 27
  - Imposing limits on I/O for block devices . . . 28
- Network filtering . . . 29
- Protecting the driver assistance system from attacks . . . 29
  - Protecting devices whose usage is restricted . . . 30
- Protecting the system from Internet threats . . . 30
  - Other sources of potential exploitation . . . 32
- Secure Software Distribution . . . 32
- Secure Boot . . . 33
- Data encryption and removal . . . 34
  - Data encryption . . . 34
  - Data removal . . . 35
- Stack Protection . . . 35
- Confining applications in containers . . . 35
  - LXC Containment . . . 35
  - The Flatpak framework . . . 37
- The IMA Linux Integrity Subsystem . . . 37
  - Conclusion regarding IMA and EVM . . . 38
- Seccomp . . . 39
- The role of the app store process for security . . . 40

How does security affect developer usage of a device? 41
Further discussion 41

This document discusses and details solutions for the security requirements of the Apertis system.

Security boundaries and threat model describes the various aspects of the security model, and the threat model for each.

Local attacks to obtain private data or damage the system, including those performed by malicious applications that get installed in the device somehow or through exploiting a vulnerable application are covered in Mandatory access control (MAC). It is also the main line of defense against malicious email attachments and web content, and for minimizing the damage that root is able to do are also mainly covered by the MAC infrastructure. This is the main security infrastructure of the system, and the depth of the discussion is proportional to its importance.

Denial of Service attacks through abuse of system resources such as CPU and memory are covered by Resource usage control. Attacks coming in through the device's network connections and possible strategies for firewall setup are covered in Network filtering

Attacks to the driver assistance system coming from the infotainment system are handled by many of these security components, so it is discussed in a separate section: Protecting the driver assistance system from attacks. Internet threats are the main subject of 10, Protecting the system from internet threats.

Secure software distribution discusses how to provide ways to make installing and upgrade software secure, by guaranteeing packages are unchanged, undamaged and coming from a trusted repository.

Secure boot for protecting the system against attacks done by having physical access to the device is discussed in Secure boot. Data encryption and removal, is concerned with features whose main focus is to protect the privacy of the user.

Stack protection, discusses simple but effective techniques that can be used to harden applications and prevent exploitation of vulnerabilities. Confining applications in containers, discusses the pros and cons of using the lightweight Linux Containers infrastructure for a system like Apertis.

The IMA Linux integrity subsystem, wraps up this document by discussing how the Integrity Measurement Architecture works and what features it brings to the table, and at what cost.

Terminology

Privilege

A component that is able to access data that other components cannot is said to be ***privileged***. If two components have different privileges – that is, at least one of them can do something that the other cannot – then there is said to be a ***privilege boundary*** between them.

Trust

A ***trusted*** component is a component that is technically able to violate the security model (i.e. it is relied on to enforce a privilege boundary), such that errors or malicious actions in that component could undermine the security model. The ***trusted computing base (TCB)*** is the set of trusted components. This is independent of its quality of implementation – it is a property of whether the component is relied on in practice, and not a property of whether the component is ***trustworthy***, i.e. safe to rely on. For a system to be secure, it is necessary that all of its trusted components be trustworthy.

One subtlety of Apertis' app-centric design[1] is that there is a privilege boundary between *application bundles* even within the context of one user. As a result, a multi-user design has two main layers in its security model: system-level security that protects users from each other, and user-level security that protects a user's apps from each other. Where we need to distinguish between those layers, we will refer to the ***TCB for security between users*** or the ***TCB for security between app bundles*** respectively.

Integrity, confidentiality and availability

Many documents discussing security policies divide the desired security properties into integrity, confidentiality and availability. The definitions used here are taken from the USA National Information Assurance Glossary.

> Committee on National Security Systems, CNSS Instruction No. 4009 National Information Assurance (IA) Glossary, April 2010. http://www.ncsc.gov/publications/policy/docs/CNSSI_4009.pdf

Integrity is the property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner. For example, if a malicious application altered the user's contact list, that would be an integrity failure.

Confidentiality is the property that information is not disclosed to system entities (users, processes, devices) unless they have been authorized to access the information. For example, if a malicious application sent the user's contact list to the Internet, that would be a confidentiality failure.

[1] https://jwd.pages.apertis.org/apertis-website/concepts/applications/

Availability is the property of being accessible and usable upon demand by an authorized entity. For example, if an application used so much CPU time, memory or disk space that the system became unusable (a denial of service attack), or if a security mechanism incorrectly denied access to an authorized entity, that would be an availability failure.

Security boundaries and threat model

This section discusses the security properties that we aim to provide.

Security between applications

The Apertis platform provides for installation of *application bundles*, which may come from the platform developer or third parties. These are described in the Applications design document.

Our model is that there is a trust boundary between these application bundles, providing confidentiality, integrity and availability. In other words, an application bundle should not normally be able to read data stored by another application bundle, alter or delete data stored by the other application bundle, or interfere with the operation of the other application bundle. As a necessary prerequisite for those properties, processes from an application bundle must not be able to gain the effective privileges of processes or programs from another application bundle (privilege escalation).

In addition to the application bundles, the Apertis *platform* (defined in the Applications design document, and including libraries, system services, and any user-level services that are independent of application bundles) has higher privilege than any particular application bundle. Similarly, an application bundle should not in general be able to read, alter or delete non-application data stored by the platform, except for where the application bundle has been granted permission to do so, such as a navigation application reading location data (a "least-privilege" approach); and the application bundle must not be able to gain the effective privileges of processes or programs from the platform.

The threat model here is to assume that a user installs a malicious application, or an application that has a security flaw leading to an attacker being able to gain control over it. The attacker is presumed to be able to execute arbitrary code in the context of the application.

Our requirement is that the damage that can be done by such applications is limited to: reading files that are non-sensitive (such as read-only OS resources) or are specifically shared between applications; editing or deleting files that are specifically shared between applications; reducing system performance, but to a sufficiently limited extent that the user is able to recover by terminating or uninstalling the malicious or flawed application; or taking actions that the application requires for its normal operation.

Some files, particularly large media files such as music, might be specifically shared between applications; such files do not have any integrity, confidentiality or availability guarantees against a malicious or subverted application. This is a trade-off for usability, similar to Android's Environment.getExternalStorageDirectory().

To apply this security model to new platform services, it is necessary for those platform services to have a coherent security model, which can be obtained by classifying any data stored by those platform services using questions similar to these:

- Can it be read by all applications, applications with a specific privilege flag, specific applications (for example the application that created it), or by some combination of those?
- Can it be written by all applications, applications with a specific privilege flag, specific applications, or some combination of those?

It is also necessary to consider whether data stored by different users using the same application must be separated (see Security between users).

For example, a platform service for downloads might have the policy that each application's download history can be read by the matching application, or by applications with a "Manage Downloads" privilege (which might for instance be granted to a platform Settings application).

As another example, a platform service for app-bundle installation might have a policy stating that the trusted "Application Installer" HMI is the only component permitted to install or remove app-bundles. Depending on the desired trade-off between privacy and flexibility, the policy might be that any application may read the list of installed app-bundles, that only trusted platform services may read the list of installed app-bundles, or that any application may obtain a subset of the list (bundles that are considered non-sensitive) but only trusted platform services may read the full list.

A service can be considered to be secure if it implements its security policy as designed, and that security policy is appropriate to the platform's requirements.

Communication between applications

In a system that supports capabilities such as data handover between applications, it is likely that pairs of application bundles can communicate with each other, either mediated by platform services or directly. The Interface Discovery[2] and Data Sharing[3] designs on the Apertis wiki have more information on this topic.

The mechanisms for communicating between application bundles, or between application bundle and the platform, are to be classified into *public* and *non-*

[2] https://jwd.pages.apertis.org/apertis-website/concepts/interface_discovery/

[3] https://jwd.pages.apertis.org/apertis-website/architecture/data_sharing/

public interfaces. Application bundles may enumerate all of the providers of *public* interfaces and may communicate with those providers, but it is not acceptable for application bundles to enumerate or communicate with the providers of *non-public* interfaces. The platform is considered to be trusted, and may communicate with any *public* or *non-public* interface.

The security policy described here is one of many possible policies that can be implemented via the same mechanisms, and could be replaced or extended with a finer-grained security policy at a later date, for example one where applications can be granted the capability to communicate with some but not all non-public interfaces.

Security between users

The Apertis platform is potentially a multi-user environment; see the Multiuser design document for full details. This results in a two-level hierarchy: users are protected from each other, and within the context of a user, apps are protected from other apps.

In at least some of the possible multi-user models described in the Multiuser design document, there is a trust boundary between users, again providing confidentiality, integrity and availability (see above). Once again, privilege escalation must be avoided.

As with security between applications, some files (perhaps the same files that are shared between applications) might be specifically shared between users. Such files do not have any integrity, confidentiality or availability guarantees against a malicious user. Android's Environment.getExternalStorageDirectory() is one example of a storage area shared by both applications and users.

Security between platform services

Within the platform, not all services and components require the same access to platform data.

Some platform components, notably the Linux kernel, are sufficiently highly-privileged that it does not make sense to attempt to restrict them, because carrying out their normal functionality requires sufficiently broad access that they can violate one of the layers of the security model. As noted in Terminology, these components are said to be part of the *trusted computing base* for that layer; the number and size of these components should be minimized, to reduce the exposure of the system as a whole.

The remaining platform components have considerations similar to those applied to applications: they should have "least privilege". Because platform components are part of the operating system image, they can be assumed not to be malicious; however, it is desirable to have "defence in depth" against design or implementation flaws that might allow an attacker to gain control of them. As such, the threat model for these components is that we assume an attacker gains control over the component (arbitrary code execution), and the desired property is that the integrity, confidentiality and availability impact is minimized, given the constraint that the component's privileges must be sufficient for it to carry out its normal operation.

Note that the concept of the trusted computing base applies to each of the two layers of the security policy. A system service that communicates with all users might be part of the TCB for isolation between users, but not part of the TCB for isolation between platform components or between applications. Conversely, a per-user service such as dconf might be part of the TCB for isolation between applications, but not part of the TCB for isolation between users. The Linux kernel is one example of a component that is part of the TCB for both layers.

Security between the device and the network

Apertis devices may be connected to the Internet, and should protect confidentiality and integrity of data stored on the Apertis device. The threat model here is that an attacker controls the network between the Apertis device and any Internet service of interest, and may eavesdrop on network traffic (passive attack) and/or substitute spoofed network traffic (active attack); we assume that the attacker does not initially control platform or application code running on the Apertis device. Our requirement is that normal operation of the Apertis device does not result in the attacker gaining the ability to read or change data on that device.

Physical security

An attack that could be considered is one where the attacker gains physical access to the Apertis system, for example by stealing the car in which it is installed. It is obviously impossible to guarantee availability in this particular threat model (the attacker could steal or destroy the Apertis system), but it is possible to provide confidentiality, via encryption "at rest".

A variation on this attack is to assume that the attacker has physical access to the system and then returns it to the user, perhaps repeatedly. This raises the question of whether integrity is provided (whether the user can be sure that they are not subsequently entering confidential data into an operating system that has been modified by the attacker).

This type of physical security can come with a significant performance and complexity overhead; as a trade-off, it could be declared to be out-of-scope.

Solutions adopted by popular platforms

As background for the discussions of this document, the following sections provide an overview of the approaches other mobile platforms have chosen for security, including an explanation of the trade-offs or assumptions where necessary.

Android

Android uses the Linux kernel, and as such relies on it being secure when it comes to the most basic security features of modern operating systems, such as process isolation and an access permissions model. On top of that, Android has a Java-based virtual machine environment which runs regular applications and provides them with APIs that have been designed specifically for Android. Regular applications can execute arbitrary native code within their application sandbox, for example by using the NDK interfaces.

> https://developer.android.com/training/articles/security-tips.html#Dalvik notes that "On Android, the Dalvik VM is not a security boundary".

However, some system functionality is not directly available within the application sandbox, but can be accessed by communicating with more-privileged components, typically using Android's Java APIs.

Early versions of Android worked under the assumption that the system will be used by a single user, and no attempt was made towards supporting any kind of multi-user use case. Based on this assumption, Android re-purposed the concept of UNIX user ID (uid), making each application run as a different user ID. This allows for very tight control over what files each application is able to access by simply using user-based permissions; this provides isolation between applications (Security between applications). In later Android versions, which do have multi-user support, user IDs are used to provide two separate security boundaries – isolating applications from each other, and isolating users from each other (Security between users) – with one user ID per (user, app) pair. This is discussed in more detail in the Multiuser design document[4].

The system's main file system is mounted read-only to protect against unauthorized tampering with system files (integrity for platform data, Security between platform services); however, this does not protect integrity against an attacker with physical access (Physical security). Encryption of the user data partition through the standard *dm-crypt* kernel facility (confidentiality despite physical access, Physical security) is supported if the user configures a password for their device. Users using gesture-based or other unlock mechanisms are unable to use this feature.

The root user on Android is all-powerful, and can do anything to the system. Android makes no attempt to limit the power of processes running as UID 0 (the root user ID); in other words, they are part of the TCB. All security of system services, and the core system and applications rely on the separation of users already discussed and in assuming nothing other than the essential (the kernel itself and a very small number of system services) runs with root privileges.

Older versions of Android did not use Mandatory Access Control, discussed in this document's chapter 5. More recent versions use SELinux to augment the

[4] https://jwd.pages.apertis.org/apertis-website/concepts/multiuser/

uid-based sandbox.

> Security-Enhanced Linux in Android, https://source.android.com/devices/tech/security/selinux/

The idea of restricting the services an application can use to those specified in the application's manifest also exists in Android. Before installation, Android shows a list of system services the application intends to access and installation only initiates if the user agrees. This differs slightly from the Applications design in Apertis[5], in which some permissions are subject to prompting similar to Android's, while other permissions are checked by the app store curator and unconditionally granted on installation.

Android provides APIs to verify a process has a given permission, but no central control is built into the API layer or the IPC mechanism as planned for Apertis – checking whether a caller has the required permissions to make that call is left to the service or application that provides the IPC interface or API, similar to how most GNOME services work by using PolicyKit[6](see section 6 for more on this topic).

> See, for instance, how the A2DP service verifies the caller has the required permission: https://github.com/android/platform_frameworks_base/blob/master/core/java/android/server/BluetoothA2dpService.java#L257

No effort is made specifically towards thwarting applications misbehaving and causing a Denial of Service on system services or the IPC mechanism. Android uses two very simple strategies to forcibly stop an application: 1) it kills applications when the device is out of memory; 2) it notifies the user of unresponsive applications[7] and allows them to force the application to close, similar to how GNOME does it.

An application is deemed to not be responding after about 5 seconds of not being able to handle user input. This feature is implemented by the Android window manager service, which is responsible for dispatching events read from the kernel input events interface (the files under **/dev/input**) to the application, in cooperation with the activity manager service, which shows the application not responding dialog and kills the application if the user decides to close it. After dispatching an event, the window manager service waits for an acknowledgement from the application with a timeout; if the timeout is hit, then the application is considered not responding.

Bada

Bada is not an Open Source platform, so closer inspection of the inner workings is not feasible. However, the documentation indicates that Bada also kills

[5] https://jwd.pages.apertis.org/apertis-website/concepts/applications/
[6] http://live.gnome.org/PolicyKit
[7] http://developer.android.com/guide/practices/design/responsiveness.html

applications when under memory pressure.

It also uses a simple *API privilege level* framework as the base of its security and reliability architecture. Applications running with the *Normal* API privilege level need to specify which *API privilege groups*[8] it needs to be able to access in their manifest file.

Some APIs are restricted under the *System* API level and can be used only by Samsung or its authorized partners. It's not possible to say whether those restrictions are applied in a general way or by having the modules that provide the APIs perform validation checks, but the latter seems more likely given these are C++ APIs that do not go through any kind of central service.

iOS

iOS is, like Bada, a closed platform, so details are sometimes difficult to obtain[9], but Apple does use some Open Source components (at the lower levels, in particular). iOS has an application sandbox[10] that is very similar in functionality to AppArmor, discussed bellow. The technology is based on Mandatory Access Control provided by the TrustedBSD[11] project and has been marketed under the *Seatbelt* name.

Like AppArmor, it uses configuration files that specify profiles, using path-based rules for file system access control. Also like AppArmor, other functionality such as network access can be controlled. The actual confinement is applied when the application uses system calls to request that the kernel carries out an action on the application's behalf (in other words, when the privilege boundary between user-space and the kernel is crossed).

Seatbelt is considered to be the single canonical solution to sandboxing applications on iOS; this is in contrast with Linux, in which AppArmor is one option among many (system calls can be mediated by seccomp, the Secure Computing API[12] described in section 17 of this document, in addition to up to one MAC layer such as AppArmor, SELinux or Smack).

None of this complexity is exposed to apps developed for iOS, though; they are merely implementation details.

Apparently, there are no central controls whatsoever protecting the system from applications that hang or try to DoS system services. The only real limitation imposed is the available system memory.

Applications are free to use any APIs available, there are no explicit declarative permissions system like the one used in Android. However, some functionality

[8] http://developer.bada.com/help/index.jsp?topic=/com.osp.documentation.help/html/bada_overview/using_privileged_api.htm
[9] http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf
[10] http://www.usefulsecurity.com/2007/11/apple-sandboxes-part-1/
[11] http://www.trustedbsd.org/mac.html
[12] http://lwn.net/Articles/475043/

are always mediated by the system, including through system-controlled UI.

For instance, an application can query the GPS for location; when that happens, the system will take over and present the user with a request for permission. If the user accepts the request will be successful and the application will be white-listed for future queries. The same goes for interacting with the camera: the application can request a picture be taken, but the UI that is presented for taking the picture is controlled by the system as is actual interaction with the camera.

This is analogous to the way in which Linux services can use PolicyKit to mediate privileged actions (see section 6), although on iOS the authorization step is specifically considered to be an implementation detail of the API used, whereas some Linux services do make the calling application aware of whether there was an interactive authorization step.

Mandatory Access Control

The goal of the Linux Discretionary Access Control (DAC) is a separation of multiple users and their data (Security between users, Security between platform services). The policies are based on the identity of a subject or their groups. Since in Apertis applications from the same user should not trust each other (Security between applications), the utilization of a Mandatory Access Control (MAC) system is recommended. MAC is implemented in Linux by one of the available Linux Security Modules (LSM).

Linux Security Modules (LSM)

Due to the different nature and objectives of various security models there is no real consensus about which security model is the best, thus support for loading different security models and solutions became available in Linux in 2001. This mechanism is called Linux Security Modules (LSM).

Although it is in theory possible to provide generic support for any LSM, in practice most distributions pick one and stick to it, since both policies and threat models are very specific to any particular LSM module.

The first implementation on top of LSM was SELinux developed by the US National Security Agency (NSA). In 2009 the TOMOYO Linux module was also included in the kernel followed by AppArmor in the same year. The subsections below gives a short introduction on the security models that are officially supported by the Linux Kernel.

SELinux

SELinux[13] is one of the most well-known LSMs. It is supported by default in Red Hat Enterprise Linux and Fedora. It is infamous for how difficult it

[13] http://selinuxproject.org/page/Main_Page

is to maintain the security policies; however, being the most flexible and not having any limitation regarding what it can label, it is the reference in terms of features. For every user or process, SELinux assigns a context which consists of a role, user name and domain/type. The circumstances under which the user is allowed to enter into a certain domain must be configured into the policies.

SELinux works by applying rules defined by a policy when kernel-mediated actions are taken. Any file-like object in the system, including files, directories, and network sockets can be labeled. Those labels are set on file system objects using extended file system attributes. That can be problematic if the file system that is being used in a given product or situation lacks support for extended attributes. While support has been built for storing labels in frequently used networking file systems like NFS, usage in newer file systems may be challenging. Note that BTRFS does support extended attributes.

Users and processes also have labels assigned to them. Labels can be of a more general kind like, for instance, the sysadm_t label, which is used to determine that a given resource should be accessible to system administrators, or of a more specific kind.

Locking down a specific application, for instance, may involve creating new labels specifically for its own usage. A label "browser_cache_t" may be created, for instance, to protect the browser cache storage. Only applications and users which have their label assigned to them will be able to access and manage those files. The policy will specify that any files created by the browser on that specific directory are assigned that label automatically.

Labels are automatically applied to any resources created by a process, based on the labels the process itself has, including sockets, files, devices represented as files and so on. SELinux, as other MAC systems, is not designed to impose performance-related limitations, such as specifying how much CPU time a process may consume, or how many times a process duplicates itself, but supports pretty much everything in the area it was designed to target.

The SELinux support built into D-Bus allows enhancement of the existing D-Bus security rules by associating names, methods and signals with SELinux labels, thus bringing similar policy-making capabilities to D-Bus.

TOMOYO Linux

TOMOYO Linux[14] focuses on the behavior of a system where every process is created with a certain purpose and allows each process to declare behaviors and resources needed to achieve their purposes. TOMOYO Linux is not officially supported by any popular Linux distribution.

SMACK

[14] http://tomoyo.sourceforge.jp/

Simplicity is the primary design goal of SMACK[15]. It was used by MeeGo before that project was cancelled; Tizen[16] appears to be the only general-purpose Linux distribution using SMACK as of 2015.

SMACK works by assigning labels to the same system objects and to processes as SELinux does; similar capabilities were proposed by Intel for D-Bus integration, but their originators did not follow up on reviews[17], and the changes were not merged. SMACK also relies on extended file system attributes for the labels, which means it suffers from the same shortcomings that come from that as SELinux.

There are a few special predefined labels, but the administrator can create and assign as many different labels as desired. The rules regarding what a process with a given label is able to perform on an object with another given label are specified in the system-wide policy file /etc/smack/accesses, or can be set in run-time using the smackfs virtual file system.

MeeGo used SMACK by assigning a separate label to each service in the system, such as "Cellular" and "Location". Every application would get their own labels and on installation the packaging system would read a manifest that listed the systems the application would require, and SMACK rules would then be created to allow those accesses.

AppArmor

Of all LSM modules that were reviewed, Application Armor (AppArmor[18]) can be seen as the most focused on application containment.

AppArmor allows the system administrator to associate an executable with a given profile in order to limit access to resources. These resource limitations can be applied to network and file system access and other system objects. Unlike SMACK and SELinux, AppArmor does not use extended file system attributes for storing labels, making it file system agnostic.

Also in contrast with SELinux and SMACK, AppArmor does not have a system-wide policy, but application profiles, associated with the application binaries. This makes it possible to disable enforcement for a single application, for instance. In the event of shipping a policy with an error that leads to users not being able to use an application it is possible to quickly restore functionality for that application without disabling the security for the system as a whole, while the incorrect profile is fixed.

Since AppArmor uses the path of the binary for profile selection, changing the path through manipulation of the file system name space (i.e. through links or mount points) is a potential way of working-around the limits that are put

[15] http://schaufler-ca.com/
[16] https://developer.tizen.org/sdk.html
[17] https://bugs.freedesktop.org/show_bug.cgi?id=47581
[18] https://gitlab.com/apparmor/apparmor/-/wikis/home

in place; while this is cited as a weakness, in practice it is not an issue, since restrictions exist to block anyone trying to do this. Creation of symbolic links is only allowed if the process doing so is allowed to access the original file, and links are followed to enforce any policy assigned to the binary they link to. Confined processes are also not allowed to mount file systems unless they are given explicit permission.

Here's an example of how restricting ping's ability to create raw sockets cannot be worked around through linking – lines beginning with $ represent commands executed by a normal user, and those starting with # have been executed by the root user:

```
$ ping debian.org
ping: icmp open socket: Operation not permitted
$ ln -s /bin/ping
$ ./ping debian.org
ping: icmp open socket: Operation not permitted
$ ln /bin/ping ping2
ln: failed to create hard link `ping2' => `/bin/ping': Operation not permitted
# ping debian.org
ping: icmp open socket: Operation not permitted
# ln -s /bin/ping /bin/ping2
# ping2 debian.org
ping: icmp open socket: Operation not permitted
#
```

AppArmor restriction applying to file system links

Copying the file would make it not trigger the containment. However, even if the user was able to symlink the binary or use mount points to work-around the path-based restrictions that should not mean privilege escalation, given the white-list approach that is being adopted. That approach means that any binary escaping its containment profile would in actuality be dropping privileges, not escalating them, since the restrictions imposed on binaries that do not have their own profile can be quite extensive.

Note that Collabora is proposing mounting partitions that should only contain data with the option that disallows execution of code contained in them, so even if the user manages to escape the strict containment of the user session and copied a binary to one of the directories they have write access to, they would not be able to run it. Refer to the System updates & rollback and Application designs for more details on file system and partition configuration.

Integration with D-Bus was developed by Canonical and shipped in Ubuntu for several years, before being merged upstream in dbus-daemon 1.9 and AppArmor 2.9. The implementation includes patches to AppArmor's user-space tools, to make the new D-Bus rules known to the profile parser, and to dbus-daemon, so that it will check with AppArmor before allowing a request.

AppArmor will be used by shipping profiles for all components of the platform, and by requiring that third-party applications ship with their own profiles that specify exactly what requests the application should be allowed.

Creating a new profile for AppArmor is a reasonably simple process: a new profile is generated automatically running the program under AppArmor's profile generator, aa-genprof[19], and exercising its features so that the profile generator can capture all of the accesses the application is expected to make. After the initial profile has been generated it must be reviewed and fine-tuned by manual editing to make sure the permissions that are granted are not beyond what is expected.

In AppArmor there is no default profile applied to all processes, but a process always inherits limitations imposed to its parent. Setting up a proper profile for components such as the session manager is a practical and effective way of implementing this requirement.

Comparison

Since all those Linux Security Modules rely on the same kernel API and have the same overall goals, the features and resources they are able to protect are very similar, thus not much time will be spent covering those. The policy format and how control over the system and its components is exerted varies from framework to framework, though, which leads to different limitations. The table below has a summary of features, simplicity and limitations:

	SELinux	AppArmor	SMACK
Maintainability	Complex	Simple	Simple
Profile creation	Manual/Tools	Manual/Tools	Manual
D-Bus integration	Yes	Yes	Not proposed upstream
File system agnostic	No	Yes	No
Enforcement scope	System-wide	Per application	System-wide

Comparison of LSM features

Historically LSM modules have focused on kernel-mediated accesses, such as access to file system objects and network resources. Modern systems, though, have several important features being managed by user-space daemons. D-Bus is one such daemon and is specially important since it is the IPC mechanism used by those daemons and applications for communication. There is clear benefit in allowing D-Bus to cooperate with the LSM to restrict what applications can talk to which services and how.

[19] https://gitlab.com/apparmor/apparmor/-/wikis/Profiling_with_tools

In that regard SELinux and AppArmor are in advantage since D-Bus is able to let these frameworks decide whether a given communication should be allowed or not, and whether a given process is allowed to acquire a particular name on the bus. Support for SMACK mediation was worked on by Intel for use in Tizen, but has not been proposed for upstream inclusion in D-Bus, and is believed to add considerable complexity to dbus-daemon. There is no work in progress to add TOMOYO support.

Like D-Bus' built-in support for applying "policy" to message delivery, AppArmor mediation of D-Bus messages has separate checks for whether the sender may send a message to the recipient, and whether the recipient may receive a message from the sender. Either or both of these can be used, and the message will only succeed if both sending and receiving were allowed. The sender's AppArmor profile determines whether it can send (usually conditional on the profile name of the recipient), and the recipient's AppArmor profile determines whether it can receive (either conditional on the profile name of the sender, or unconditionally), so some coordination between profiles is needed to express a particular high-level security policy.

The main difference between the SELinux and SMACK label-based mediation in terms of features is how granular you can get. With the D-Bus additions to the AppArmor profile language[20], for instance, in addition to specifying which services can be called upon by the constrained process it is also possible to specify which interfaces and paths are allowed or denied. This is unlike SELinux mediation[21], which only checks whether a given client can talk to a given service. One caveat regarding fine-grained (interface- and path-based) D-Bus access control is that it is often not directly useful, since the interface and path is not necessarily sufficient to determine whether an action should be allowed or denied (for example, Motivation for polkit describes why this is the case for the udisks service). As a result of considerations like this, the developers of kdbus oppose the addition of fine-grained access control within kdbus, and have indicated that kdbus' access-control will never go beyond allowing or rejecting a client communicating with a service.

> kdbus is a kernel module that has been proposed to take over the role of the user-space dbus-daemon in D-Bus on Linux systems. https://github.com/gregkh/kdbus

Software that is being used by large distributions is often more tested and tested in more diverse scenarios. For this reason Collabora believes that being used by one of the main distributions is a very important feature to look for in a LSM.

Flexibility is also good to have, since more complex requirements can be modeled more precisely. However, there is a trade-off between complexity and flexibility that should be taken into consideration.

[20] https://gitlab.com/apparmor/apparmor/-/wikis/AppArmor_Core_Policy_Reference#dbus-rules

[21] http://dbus.freedesktop.org/doc/dbus-daemon.1.html#lbAg

The recommendation on the selection of the framework is a combination of the adoption of the framework by existing distributions, features, maintainability, cost of deployment and experience of the developers involved. The table below contains a comparison of the adoption of the existing security models. Only major distributions that ship and enable the module by default are listed.

Name	Distributions	Merged to mainline	Maintainer
SELinux	Fedora, Red Hat Enterprise	08 Aug 2003	NSA, Network Associates, Secure Computi
AppArmor	SUSE, OpenSUSE, Ubuntu	20 Oct 2010	SUSE, Canonical
SMACK	Tizen	11 Aug 2007	Intel, Samsung2
TOMOYO		10 Jun 2009	NTT Data Corp.

Comparison of LSM adoption and maturity

Performance impact

The performance impact of MAC solutions depends heavily on the workload of the application, so it's hard to rely upon a single metric. It seems major adopters of these technologies are not too concerned about their real-world impact, even though they may be expressive in benchmarks, since there are no recent measurements of performance impact for the major MAC solutions.

That said, early tests indicate that SELinux has a performance impact floating around 7% to 10%[22], with tasks that are more CPU intensive having *less* impact, since they are not making many system calls that are checked. SELinux performs checks on every operation that touches a labeled resource, so when reading or writing a file all read/write operations would cause a check. That means making larger operations instead of several smaller ones would also make the overhead go down.

AppArmor generally does fewer checks than SELinux since only operations that open, map or execute a file are checked: the individual read/write operations that follow are not checked independently. Novell's documentation and FAQs state a 0.2% overhead is expected on best-case scenarios – writing a big file, for instance, with a 2% overhead in worst-case scenarios (an application touching lots of files once). Collabora's own testing on a 2012 x86-64 system puts the worst case scenario leaning towards the 5% range. The test measured reading 3000 small files with a hot disk cache, and ranged from ~89ms to ~94ms average duration.

SMACK's performance characteristics should be similar to that of SELinux, given their similar approach to the problem. SMACK has been tested for a TV embedded scenario[23] which has shown performance degradation from 0% all

[22] http://blog.larsstrand.no/2007/11/rhel5-selinux-benchmark.html
[23] http://www.embeddedalley.com/pdfs/Smack_for_DigitalTV.pdf

the way to 30% on a worst-case scenario of deleting a 0-length file. Degradation varied greatly depending on the benchmark used.

The only conclusion Collabora believes can be drawn from these numbers is that an approach which checks less often (as is the case for AppArmor) can be expected to have less impact on performance, in general. That said, these numbers should be taken with a grain of salt, since they haven't been measured in the exact same hardware and with the exact same methodology. They may also suffer from bias caused by benchmark tests which may not represent real-world usage scenarios.

No numbers exist measuring the impact on performance of the existing D-Bus SELinux and AppArmor mediation, nor with the in-development SMACK mediation. The overhead caused to each D-Bus call should be similar to that of opening a file, since the same procedure is involved: a check needs to be done each time a message is received from a client that is contained. It should be noted that D-Bus is not designed to be used for high-frequency communication due to its per-message overhead, so the additional overhead for AppArmor should not be problematic unless D-Bus is already being misused.

Where higher-frequency communication is required, D-Bus' file descriptor passing feature can be used to negotiate a private channel (a pipe or socket) between two processes. This negotiation can be as simple as a single D-Bus method call, and only incurs the cost of AppArmor checks once (when it is first set up). Subsequent messages through the private channel bypass D-Bus and are not checked individually by AppArmor, avoiding any per-message overhead in this case.

A more realistic and reliable assessment of the overhead imposed on a real-world system would only be feasible on the target hardware, with actual applications, where variables like storage device and file system would also be better controlled.

Conclusion

Collabora recommends the adoption of a MAC solution, specifically AppArmor. It solves the problem of restricting applications to the privileges they require to work, and is an effective solution to the problem of protecting applications from other applications running for the same user, which a DAC model is not able to provide.

SMACK and TOMOYO have essentially no adoption and support when compared to solutions like SELinux and AppArmor, without providing any clear advantages. MeeGo would have been a good testing ground for SMACK, but the fact that it was never really deployed in enforcing mode means that the potential was never realized.

SELinux offers the most flexible configuration of security policies, but it introduces a lot of complexity on the setup and maintenance of the policies, not only for distribution maintainers but also for application developers and packagers, which impacts on the costs of the solution. It is quite common to see Fedora users running into problems caused by SELinux configuration issues.

AppArmor stands out as a good middle-ground between flexibility and maintainability while at the same time having significant adoption: by the biggest end-user desktop distribution (Ubuntu) and by one of the two biggest enterprise distributors (SUSE). The fact that it is the security solution already supported and included in the Ubuntu distribution, which is the base of the Apertis platform, minimizes the initial effort to create a secure baseline and reduces the effort needed to maintain it. Since Ubuntu ships with AppArmor, some of the services and applications will already be covered by the profiles shipped with Ubuntu. Creation of additional profiles is made easy by the profile generator tool that comes with AppArmor. it records everything the application needs to do during normal operation, and allows for further refining after the recording session is done.

Collabora will integrate and validate the existing Ubuntu profiles that are relevant to the Apertis platform as well as modify or write any additional profiles required by the base platform. Collabora will also assist in the creation of profiles for higher level applications that ship with the final product and on the strategy for profile management for third party applications.

AppArmor Policy and management examples

Looking at a few examples might help better visualize how AppArmor works, and what creating new policies entices. Let's look at a simple policy file:

```
$ cat /etc/apparmor.d/bin.ping
...
/bin/ping {
  #include <abstractions/base>
  #include <abstractions/consoles>
  #include <abstractions/nameservice>

  capability net_raw,
  capability setuid,
  network inet raw,
  /bin/ping mixr,
  /etc/modules.conf r,
 ## Site-specific additions and overrides. See local/README for details.
  #include \<local/bin.ping\>
}
$
```

AppArmor policy shipped for ping in Ubuntu

This is the policy for the ping command. The binary is specified, then a few includes that have common rules for the kind of binary ping (console), and services it consumes (nameservice). Then we have two rules specifying capabilities that the program is allowed to use, and we state the fact that it is allowed to do perform raw network operations. Then it's specified that the process should be able to memory map (m) /bin/ping, inherit confinement from the parent (i), execute the binary /bin/ping (x) and read it (r). It's also specified that ping should be able to read /etc/modules.conf.

If an attack was able to execute arbitrary code by hijacking the ping process, then that is all it would be able to do. No reading of /etc/password would be allowed, for instance. If ping was a very core feature of the device and starts failing because of a bad policy, it is possible to disable security enforcement just for ping, leaving the rest of the system secured (something that would not be easily done with SMACK or SELinux), by running *aa-disable* with ping's path as the parameter, or by installing a symbolic link in /etc/apparmor.d/disable:

```
$ aa-disable /bin/ping
Disabling /bin/ping.
$ ls -l /etc/apparmor.d/disable/
total 0
lrwxrwxrwx 1 root root 24 Feb 20 19:38 bin.ping ->
/etc/apparmor.d/bin.ping
```

A symbolic link to disable the ping AppArmor policy

Note that *aa-disable* is only a convenience tool to unload a profile and link it to the **/etc/apparmor.d/disable** directory. Note that the convenience script is not currently shipped in the image intended for the target hardware. It is available in the repository though, and is available in the development and SDK images since it makes it more convenient to test and debug issues.

Note, also, that writing to the **/etc/apparmor.d/disable** directory is required for creating the symlink there, and the UNIX DAC permissions system already protects that directory for writing - only root is able to write to this directory. As discussed in A note about root, if an attacker becomes root the system is already compromised.

Also, as discussed in the System update & rollback, the system partition will be mounted read-only, so that is an additional protection layer already. And in addition to that, the white-list approach discussed in Implementing a white list approach will already deny writing to anywhere in the file system, so anything running under the application manager will have an additional layer of security imposed on them.

For these reasons, Collabora doesn't see any reason to add additional security such as AppArmor profiles specifically for protecting the system against unauthorized disabling of profiles.

Profiles for libraries

AppArmor profiles are always attached to a binary. That means there is no way to attach a profile to every program that uses a given library. However, developers can write files called *abstractions* with rules that can be included through the *#include* directive, similar to how libraries work for programming. Using this feature Collabora has written rules for the WebKit library, for instance, that can be included by the browser application as well as by any application that uses the library.

There is also concern with protecting internal, proprietary libraries, so that they cannot be used by applications. In the profiles and abstractions shipped with Apertis right now, all applications are allowed to use all libraries that are installed in the public library paths (such as **/usr/lib**).

The rationale for this is libraries are only pieces of code that could be included by the applications themselves, and it would be very time-consuming and error prone having to specify each and every library and module the application may need to use directly or that would be used indirectly by a library used by the application.

Collabora recommends that proprietary libraries that are used only by one or a few services should be installed in a private location, such as the application's directory. That would put those libraries outside of the paths covered by the existing rules, and they would this be out of reach for any other application already, given the white-list approach to session lockdown, as discussed in Implementing a white list approach.

If that is not possible, because the library hardcodes paths or some other issue, an explicit deny rule could be added to the **chaiwala-base** abstraction that implements the general rules that apply to most applications, including the one that allows access to all libraries. Collabora can help deciding what to do with specific libraries through support tickets opened in the bug tracking system.

> Chaiwala was a development codename for parts of the Apertis system. The name is retained here for compatibility reasons.

Application installation and upgrades

For installations and upgrades to be performed, no changes to the running system's security are necessary, since the processes that manage upgrade, including the creation of the required snapshots will have enough power given to them

An application's profile is read at startup time. That means an application that has been upgraded will only be contained with the new rules after it has been restarted. The D-Bus integration works by querying the kernel interface for the PID it is communicating with, not its own, so D-Bus itself does not need to be restarted when new profiles are installed.

When a *.deb* package is installed its AppArmor profile will be installed to the system AppArmor profile location (*/etc/apparmor.d/)*, but in the new snapshot created for the upgrade rather than on the running system.

The new version of the upgraded package and its new profile will only take effect after the system has been rebooted. For details about how *.deb* packages will be handled when the system is upgraded please see the *System Updates and Rollback* document.

For more details on how applications from the store will be handled, the *Applications* document produced by Collabora goes into details about how the permissions specified in the manifest will be transformed into AppArmor profiles and on how they will be installed and loaded.

A note about root

As has been demonstrated in listing *AppArmor restriction applying to file system links*, AppArmor can restrict even the powers of the root user. Most platforms do not try to limit that power in any way, since if an attacker has breached the system to get root privileges it's likely that all bets are already off. That said, it should be possible to limit the root user's ability to modify the AppArmor profiles, leaving that task solely for the package manager (see the Applications design for details).

Implementing a white-list approach

Collabora recommends the use of a white-list approach in which the app-launcher will be confined to a policy that denies almost everything, and specific permissions will be granted by the application profiles. This means all applications will only be able to access what is expressively allowed by their specific policies, providing Apertis with a very tight least-privilege implementation.

A simple example of how that can be achieve using AppArmor is provided in the following examples. The examples will emulate the proposed solution by locking down a shell, which represents the Apertis application launcher, and granting specific privileges to a couple applications so that they are able to access the files they require.

Listing *Sample profiles for implementing white-listing* shows a profile for the shell, essentially denying it access to everything by not allowing access to any files. It gives the shell permission to run both ls and cat. Note that flags *rix* are used for this, meaning the shell can read the binaries (r), and execute them (x); the *i* preceding the *x* tells AppArmor that these binaries should inherit the shell's confinement rules, even if they have rules of their own.

Then permission is given for the shell to run the *dconf* command. dconf is GNOME's settings storage. Notice that we have p as the prefix for x this time. This means we want this application to use its own rules; if no rules had been specified, then AppArmor would have fallen back to using the shell's confinement rules.

```
$ cat /etc/apparmor.d/bin.zsh4
## Last Modified: Fri May 11 11:43:44 2012

#include <tunables/global>
/bin/zsh4 {
  #include <abstractions/base>
  #include <abstractions/consoles>
  #include <abstractions/nameservice>
  /bin/ls rix,
  /bin/cat rix,
  /usr/bin/dconf rpx,
  /bin/zsh4 mr,
  /usr/lib/zsh/*/zsh/* mr,
}

$ cat /etc/apparmor.d/usr.bin.dconf
## Last Modified: Fri May 11 11:59:09 2012

#include <tunables/global>
/usr/bin/dconf {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  @{HOME}/.cache/dconf/user rw,
  @{HOME}/.config/dconf/user r,
  /usr/bin/dconf mr,
}
```

Sample profiles for implementing white-listing

The profile for *dconf* allows reading (and only reading) the user configuration for dconf itself, and allows reading and writing to the cache. By using these rules we have both guaranteed that no application executed from this shell will be able to look at or interfere with dconf's files, and that dconf itself is able to function when used. Here's the result:

```
% cat .config/dconf/user
cat: .config/dconf/user: Permission denied
% dconf read /apps/empathy/ui/show-offline
true
```

%

Effects of white-list approach profiles

As shown by this example, the application launcher itself and any applications which do not posses profiles can be restricted to the bare minimum permissions, and applications can be given the more specific privileges they require to do their job, using the *p* prefix to let AppArmor know that's what is desired.

polkit (PolicyKit)

polkit (formerly PolicyKit) is a service used by various upstream components in Apertis, as a way to centralize security policy for actions delegated by one process to another. The central problems addressed by polkit are that the desired security policies for various privileged actions are system-dependent and non-trivial to evaluate, and that generic components such as the kernel's DAC and MAC subsystems do not have enough context to understand whether a privileged action is acceptable.

Motivation for polkit

Broadly, there are two ways a process can carry out a desired action: it can do it directly, or it can use inter-process communication to ask a service to do that operation on its behalf. If the action is done directly, the components that say whether it can succeed are the Linux kernel's normal discretionary access control (DAC) permissions checks, and if configured, a mandatory access control module (MAC, section 5).

However, the kernel's relatively coarse-grained checks are not sufficient to express the desired policies for consumer-focused systems. A frequent example is mounting file systems on removable devices: if a user plugs in a USB stick with a FAT filesystem, it is reasonable to expect the user interface layer to either mount it automatically, or let the user choose to mount it. Similarly, to avoid data loss, the user should be able to unmount the removable device when they have finished with it.

Applying the desired policy using the kernel's permission checks is not possible, because mounting and unmounting a USB stick is fundamentally the same system call as mounting and unmounting any other file system, which is not desired: if ordinary users can make arbitrary mount system calls, they can mount a file system that contains setuid executables and achieve privilege escalation. As a result, the kernel disallows direct mount and unmount actions by unprivileged processes; instead, user processes may request that a privileged system process carries out the desired action. In the case of device mounting, Apertis uses the privileged udisks2 service to mount and unmount devices.

In environments that use a MAC framework like AppArmor, actions that would normally be allowed can also become privileged: for instance, in a framework for sandboxed applications, most apps should not be allowed to record audio. The resulting AppArmor adjustments prevent carrying out these actions directly. The result is that, again, the only way to achieve them is that a service with a suitable privilege carries out the action (perhaps with a mandatory user interface prompt first, as in certain iOS features).

These privileged requests are commonly sent via the D-Bus interprocess communication (IPC) system; indeed, this is one of the purposes for which D-Bus was designed. D-Bus has facilities for allowing or forbidding messages between particular processes in a somewhat fine-grained way, either directly or mediated by MAC frameworks. However, this has the same issue as the kernel's checks for direct mount operations: the generic D-Bus IPC framework does not understand the context of the messages. For example, it can allow or forbid messages that ask to mount a device, but cannot discriminate based on whether the device in question is a removable device or a system partition, because it does not have that domain-specific information.

This means that the security decision – having received this request, should the service obey it? – must be at least partly made by the service itself (for example udisks2), which does have the necessary domain-specific context to do so.

The kdbus subsystem proposed for inclusion in the Linux kernel, which aims to supersede the user-space implementation of D-Bus, has an additional restriction: to minimize the amount of code in the TCB, it only parses the parts of a message that are necessary for normal message-routing. As a result, it does not discriminate between messages by their interface, member name or object-path, only by attributes of the source and destination processes. This is another reason why permissions checking for services such as disk-mounting must be done at least partly by the domain-specific service such as udisks2.

The desired security policies for certain actions are also relatively complex. For example, udisks2 as deployed in a modern Linux desktop system such as Debian 8 would normally allow mounting devices if and only if:

- the requesting user is *root,* or
- the requesting user is in group *sudo*, or
- all of
    - the device is removable or external, and
    - the mount point is in /media, and
    - the mount options are reasonable, and
    - the device's *seat* (in multi-seat computing) matches one of the seats at which the user is logged-in, and
    - either
        - the user is in group *plugdev*, or
        - all of

- the user is logged-in locally, and
- the user is logged-in on the foreground virtual console

This is already complex, but it is merely a default, and is likely to be adjusted further for special purposes (such as a single-user development laptop, a locked-down corporate desktop, or an embedded system like Apertis). It is not reasonable to embed these rules, or a sufficiently powerful parser to read them from configuration, into every system service that must impose such a policy.

polkit's solution

polkit addresses this by dividing the authorization for actions into two phases.

In the first phase, the domain-specific service (such as udisks2 for disk-mounting) interprets the request and classifies it into one of several ***actions*** which encapsulate the type of request. The principle is that the *action* combines the verb and the object for the desired operation: if a security policy would commonly produce different results when performing the same verb on different objects, then they are represented by different actions. For example, udisks2 divides the high-level operation "mount a disk" into the actions org.freedesktop.udisks2.filesystem-mount, org.freedesktop.udisks2.filesystem-mount-system, org.freedesktop.udisks2.filesystem-mount-other-seat and org.freedesktop.udisks2.filesystem-fstab depending on attributes of the disk. It also gathers information about the process making the request, such as the user ID and process ID. polkit clients do not currently record the LSM context (AppArmor profile, etc.) used by MAC frameworks, but could be enhanced to do so.

In the second phase, the service sends a D-Bus request to polkit with the desired action, and the attributes of the process making the request. polkit processes this request according to its configuration, and returns whether the request should be obeyed.

In addition to "yes" or "no", polkit security policies can request that a user, or a user with administrative (root-equivalent) privileges, authenticates themselves interactively; if this is done, polkit will not respond to the request until the user has responded to the *polkit agent*, either by authenticating or by cancelling the operation.

We recommend that this facility is not used with a password prompt in Apertis, since that user experience would be highly distracting. For operations that are deemed to be allowed or rejected by the platform designer, either the policy should return "yes" or "no" instead of requesting authorization, or the platform-provided polkit agent should return that result in response to authorization requests without any visible prompting. However, a prompt for authorization, without requiring authentication, might be a desired UX in some cases.

Recommendation

We recommend that Apertis should continue to provide polkit as a system service. If this is not done, many system components will need to be modified to refrain from carrying out the polkit check.

If the desired security policy is merely that a subset of user-level components may carry out privileged actions via a given system service, and that all of those user-level components have equal access, we recommend that Apertis' polkit configuration should allow and forbid actions appropriately.

If it is required that certain user-level components can communicate with a given system service with different access levels, we recommend enhancing polkit so that it can query AppArmor, giving the *action* as a parameter, before carrying out its own checks; this parallels what dbus-daemon currently does for SELinux and AppArmor.

Alternative design: rely entirely on AppArmor checks

The majority of services that communicate with polkit do so through the libpolkit-gobject library. This suggests an alternative design: the polkit service and its D-Bus API could be removed entirely, and the AppArmor check described above could be carried out in-process by each service, by providing a "drop-in" compatible replacement for libpolkit-gobject that performed an AppArmor query itself instead of querying polkit.

We do not recommend this approach: it would be problematic for services such as systemd that do not use libpolkit-gobject, it would remove the ability for the policy to be influenced by facts that are not known to AppArmor (such as whether a user is logged-in and active), and it would be a large point of incompatibility with upstream software.

Resource Usage Control

Resource usage here refers to the limitation and prioritization of hardware resources usage. Common resources to limit usage of are CPU, memory, network, disk I/O and IPC.

The proposed solution is Control Groups (cgroup-v1[24], cgroup-v2[25]), which is a Linux kernel feature to limit, account, isolate and prioritize resource usage of process groups. It protects the platform from resource exhaustion and DoS attacks. The groups of processes can be dynamically created and modified. The groups are divided by certain criteria and each group inherits limits from its parent group.

The interface to configure a new group is via a pseudo file system that contains directories to label the groups and each directory can have sub-directories (sub-

[24] https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
[25] https://www.kernel.org/doc/Documentation/cgroup-v2.txt

groups). All those directories contain files that are used to set the parameters or provide information about the groups.

By default, when the system is booted, the init system Collabora recommends for this project, systemd, will assign separate control groups to each of the system services. Collabora will further customize the cgroups of the base platform to clearly separate system services, built-in applications and third-party applications. Support will be provided by Collabora for fine-tuning the cgroup profiles for the final product.

Imposing limits on I/O for block devices

The *blkio* subsystem is responsible for dealing with I/O operations concerning storage devices. It exports a number of controls that can be tuned by the *cgroups* subsystem. Those controls fall into one of two possible strategies: setting proportional weights for different cgroups or absolute upper bounds.

The main advantage of using proportional weights is that the it allows the I/O bandwidth to be saturated – if nothing else is running, an application always gets all of the available I/O bandwidth. If, however, two or more processes in different cgroups are competing for access to the I/O bandwidth, then they will get a share that is proportional to the weights of their cgroups.

For example, suppose a process A is on a cgroup with weight **10** (the minimum value possible) is working on mass-processing of photos, and process B is on a cgroup with weight **1000** (the maximum). If process A is the only one making I/O requests, it has the full available I/O bandwidth available for itself. As soon as process B starts doing its own I/O requests, however, it will get around **99%** of all the requests that get through, while process A will have only **1%** for its requests.

The second strategy is setting an absolute limit on the I/O bandwidth, often called *throttling*. This is done by writing how many bytes per second a cgroup should be able to transfer into a virtual file called **blkio.throttle.read_bps_device**, that lives inside the cgroup. This allows a great deal of control, but also means applications belonging to that cgroup are not able to take advantage of the full I/O bandwidth even if they are the only ones running at a given point in time.

Specifying a default weight to all applications, lower weights for mass-processing jobs, and higher weights for time-critical applications is a good first step in not only securing the system, but also improving the user experience. The hard-limit of an upper bound on I/O operations can also serve as a way to make sure no application monopolizes the system's I/O.

As is usual for tunables such as these, more specific details on what settings should be specified for which applications is something that needs to be developed in an empirical, iterative way, throughout the development of the platform, and with actual target hardware. More details on the *blkio* subsystem support for cgroups can be obtained from Linux documentation[26].

Network filtering

Collabora recommends the use of the Netfilter framework to filter network traffic. Netfilter provides a set of hooks inside the Linux kernel that allow kernel modules to register callback functions with the network stack. A registered callback function is then called back for every packet that traverses the respective hook within the network stack. Iptables is a generic table structure for the definition of rule sets. Each rule within an iptable consists of a number of classifiers (iptables matches) and one connected action (iptables target).

Netfilter, when used with iptables, creates a powerful network packet filtering system which can be used to apply policies to both IPv4 and IPv6 network traffic. A base rule set that blocks all incoming connections will be added to the platform by default, but port 80 access will be provided for devices connected to the Apertis hotspot, so they can access the web server hosted on the system. See the Connectivity document for more information on how this will work.

The best way to do that seems to be to add acceptance rules for the predefined private network address space the DHCP server will use for clients of the hotspot.

Collabora will offer support in refining the rules for the final product. Some network interactions may be handled by means of an AppArmor profile instead.

Protecting the driver assistance system from attacks

All communication with the driver assistance system will be done through a single service that can be talked to over D-Bus. This service will be the only process allowed to communicate with the driver assistance system. This means this service can belong to a separate user that will be the only one capable of executing the binary, which is Collabora's first recommendation.

The daemon will use an IP connection to the driver assistance system, through a simple serial connection. This means that the character device entry for this serial connection shall be protected both by an udev[27] rule that assigns permissions for only this particular user. Access to the device entry should also be denied by the AppArmor profile which covers all other applications, making sure the daemon's profile allows it.

Additionally, process namespace functionality can be used to make sure the driver assistance network interface is only seen and usable by the daemon that acts as gatekeeper. This is done by using a Linux-specific flag to the clone[28]

[26]https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt
[27]http://en.wikipedia.org/wiki/Udev
[28]https://man7.org/linux/man-pages/man2/clone.2.html

system call, CLONE_NEWNET, which creates a new process with its network namespace limited to viewing the loopback interface.

Having the process in its own cgroup also helps making it more robust, since Linux tries to be fair among cgroups, so is a good idea in general. Systemd already puts each service it starts in a separate cgroup, so making the daemon a system service is enough to take advantage of that fairness.

The driver assistance communication daemon shall be started with this flag on, and have the network interface for talking to the driver assistance system be assigned to its namespace. When a network interface is assigned to a namespace only processes in that namespace can see and interact with it. This approach has the advantage of both protecting the interface from processes other than the proxy daemon, and protecting the daemon from the other network interfaces.

Protecting devices whose usage is restricted

One or more cameras will be available for Apertis to control, but they should not be accessed by any applications other than the ones required to implement the driver assistance use cases. Cameras are made available as device files in the /dev file system and can thus be controlled by both DAC permissions and by making the default AppArmor policy deny access to it as well.

Protecting the system from Internet threats

The Internet is riddled with malicious or buggy code that present threats other than those that come from direct attacks to the device's IP connection. The user of a system such as the Apertis may face attacks such as emails that link to viruses, trojan horses and other kinds of malware, web sites that mislead the user or that try to cause the system to misbehave or become unresponsive.

There is no single answer to such threats, but care should be exercised to make each of the subsystems and applications involved in dealing with content from the Internet robust to such malicious and buggy content. The solutions that have been presented in the previous sections are essential for that.

The first line of defence is, of course, a good firewall setup that disallows incoming connections, protecting the IP interfaces of the device. The second line of defence is making sure that the applications that deal with those threats are well-written. Web browsers have also grown many techniques to protect the user from both direct attacks such as denial of service or private information disclosure and indirect forms of attack such as social engineering.

The basic rule of protecting the user from web content in a browser is essentially assuming all content is untrusted. There are fewer APIs that allow a web application to interact with local resources such as local files than there are for native applications. The ones that do exist are usually made possible only through express user interaction, such as when the user selects a file to upload.

Newer API that allows access to device capabilities such as the geolocation facilities only work after the user has granted permission.

Browsers also try to make sure users are not fooled into believing they are in a different site than the one they are really at, known as "phishing", which is one of the main social engineering attacks used on the web. The basic SSL certificate checks, along with proper UI to warn the user about possible problems can help prevent man-in-the-middle[29] attacks. The HTTP library used by the clutter port of WebKit is able to verify certificates using the system's trusted Certificate Authorities.

The *ca-certificates* package in Debian and Ubuntu carry those

In addition to those basic checks, WebKit includes a feature called *XSS Auditor* which implements a number of rules and checks to prevent cross-site scripting[30] attacks, sometimes used to mix elements from both a fake and a legitimate site.

The web browser can be locked down, like any other application, to limit the resources it can use up or get access to, and Collabora will be helping build an AppArmor profile for it. This is what protects the system from the browser in case it is exploited. By limiting the amount of damage the browser can do to the system itself, any exploits are also hindered from reaching the rest of the system.

It is also important that the UI of the browser behaves well in general. For instance, user interfaces that make it easy to run executables downloaded from the web make the system more vulnerable to attacks. A user interface that makes it easier to distinguish the domain from the rest of the URI is sometimes[31] employed to help careful users be sure they are where they wanted to go.

Automatically loading pages that were loaded or loading when the browser had to be terminated or crashed would make it hard for the user to regain control of the browser too. Existing browsers usually load an alternate page with a button the user can click to load the page, which is probably also a good idea for the Apertis browser.

Collabora evaluated taking the WebKit Clutter port to the new WebKit2 architecture as part of the Apertis project; as of 2012 it was deemed risky given the time and budget constraints.

As of 2015, it has been decided that Apertis will switch away from WebKit Clutter and onto the GTK+ port, which is already built upon the WebKit2 architecture. The main feature of that architecture is that it has several different classes of processes: the UI process deals with user interaction, the Web processes render page contents, the Network process mediates access to remote data, and the Plugin processes are responsible for running plugins.

[29] https://en.wikipedia.org/wiki/Man-in-the-middle_attack
[30] https://en.wikipedia.org/wiki/Cross-site_scripting
[31] https://chrome.googleblog.com/2010/10/understanding-omnibox-for-better.html

The fact that the processes are separate provides a great way of locking them down properly. The Web processes, which are the most likely to be exploited in case of successful attack are also the one that needs the least privileges when it comes to interfacing with the system, so the AppArmor policies that apply to it can be very strict. If a limited set of plugins is supported, the same can be applied to the Plugin processes. In fact, the WebKit codebase contains support for using seccomp filters (see Seccomp) to sandbox the WebKit2 processes. It may be a useful addition in the future.

Other sources of potential exploitation

Historically, document viewers and image loaders have had vulnerabilities exploited in various ways to execute arbitrary code. PDF and spreadsheet files, for instance, feature domain-specific scripting languages. These scripting facilities are often sandboxed and limited in what they can do, but have been a source of security issues nevertheless. Images do not usually feature scripting, but their loaders have historically been the source of many security issues, caused by programming errors, such as buffer overflows. These issues have been exploited to cause denial of service or run arbitrary code.

Although these cases do deserve mention specifically for the inherent risk they bring, there is no silver bullet for this problem. Keeping applications up-to-date with security fixes, using hardening techniques such as stack protection, discussed in Stack protection, and locking the application down to its minimum access requirements are the tools that can be employed to reduce the risks.

Launching applications based on MIME type

It is common in the desktop world to allow launching an application through the files that they are able to read. For instance, while reading email the user may want to view an attachment; by "opening" the attachment an application that is able to display that kind of file would be launched with the attachment as an argument.

Collabora is recommending that all kinds of application launching always go through the application manager. By doing that, there will be a centralized way of controlling and limiting the launching of applications through MIME or other types of content association, including being able to blacklist applications with known security issues, for instance.

Secure Software Distribution

Secure software updates are a very important topic in the security of the platform. Checking integrity and authenticity of the software packages installed in the system is crucial; an altered package might compromise the security of the whole platform.

This section is only related with security aspects, not the whole software distribution update mechanism, which will be covered in a separate document. The technology used for this is the same one used by Ubuntu. It's called Secure APT[32] and was introduced in Debian in 2005.

Every Debian or Ubuntu package that is made available through an APT repository is hashed and the hash is stored on the file that lists what packages are available, called the "Packages" file. That file is then hashed and the hash is stored in the Release file[33], which is signed using a PGP private key.

The public PGP key is shipped along with the product. When the package manager obtains updates or new packages it checks that the signature on the Release file is valid, and that all hashes match. The security of this approach relies on the fact that any tampering with the package or with the Packages file would make the hashes not match, and any changes done to the Release file would render the signature invalid.

Additional public keys can be distributed through upgrades to a package that ships installed; this is how Debian and Ubuntu distribute their public keys. This mechanism can be used to add new third-party providers, or to replace the keys used by the app store. Collabora will provide documentation and provide assistance on setting up the package repositories and signing infrastructure.

Secure Boot

The objective of secure boot[34] is to ensure that the system is booted using sanctioned components. The extent to which this is ultimately taken will vary between implementations, some may use secure boot avoid system kernel replacement, whilst others may also use it to ensure a Trusted Execution Environment[35] is loaded without interference.

The steps required to implement secure boot are vendor specific and thus the full specification for the solution depends on a definition from the specific silicon vendor, such as Freescale.

A solution that has been adopted by Freescale in the past is the High Assurance Boot (HAB), which ensures two basic attributes: authenticity and integrity. This is done by validating that the code image originated from a trusted source (authenticity), and verify that the code is in its original form (integrity). HAB uses digital signatures to validate the code images and thereby establishes the security level of the system.

To verify the signature the device uses the Super Root Key (SRK) which is stored on-chip in non-volatile memory. To enhance the robustness of HAB

[32] https://wiki.debian.org/SecureApt
[33] https://wiki.debian.org/SecureApt#Secure_apt_groundwork:_checksums
[34] https://jwd.pages.apertis.org/apertis-website/architecture/secure-boot/
[35] https://jwd.pages.apertis.org/apertis-website/concepts/op-tee/

security, multiple Super Root keys (RSA public keys) are stored in internal ROM. Collabora recommends the utilization of SRK with 2048-bit RSA keys.

In case a signature check fails because of incomplete or broken upgrade it should be possible to fall back to an earlier kernel automatically. Details of how that would be achieved are only possible after details about the hardware support for such a feature are provided by Freescale, and are probably best handled in the document about safely upgrading, system snapshots and rolling back updates.

More discussion of system integrity checking, its limitations and alternatives can be found later on, when the IMA system is investigated. See Conclusion regarding IMA and EVM in particular.

The signature and verification processes are described in the Freescale white paper "Security Features of the i.MX31 and i.MX31L".

Data encryption and removal

Data encryption

The objective of data encryption is to protect the user data for security and privacy reasons. In the event of the car being stolen, for instance, important user data such as passwords should not be easily readable. While providing full disk encryption is both not practical and harmful to overall system performance, encryption of a limited set of the data such as saved passwords is possible.

The Secrets D-Bus service[36] is a very practical way of storing passwords for applications. Its GNOME implementation[37] provides an easy to use API, uses locked down memory[38] when handling the passwords and encrypted storage for the passwords on disk. Collabora will provide these tools in the base platform and will support the implementation of secure password storage in the applications that will be developed.

One unresolved issue for data encryption, whether via the Secrets service, a full-disk encryption system (as optionally used in Android) or some other implementation, is that a secret token must be provided in order to decrypt the encrypted data. This is normally a password, but prompting for a password is likely to be undesired in an automotive environment. One possible implementation is to encode an unpredictable token in each car key, and use those tokens to decrypt stored secrets, with any of the keys for a particular car equally able to decrypt its data. In the simplest version of that implementation, loss of all of the car keys would result in loss of access to the encrypted data, but the car vendor could retain copies of the keys' tokens (and a record of which car is the relevant one) if desired

[36] https://specifications.freedesktop.org/secret-service/latest/re01.html
[37] https://wiki.gnome.org/Projects/GnomeKeyring
[38] https://wiki.gnome.org/Projects/GnomeKeyring/Memory

Data removal

A data removal feature is important to guarantee that personal user data that resides on the device can be removed before the car changes hands, for instance. Returning the device configuration to factory is also important because it allows resetting of any customization and preferences.

Collabora recommends these features be implemented by making sure user data and settings are stored in a separate storage area. By removing this area both user data and configuration are removed.

Proper data wiping is only necessary to defeat forensic analysis of the hardware and would not pose a privacy risk for the simpler cases of the car changing hands. Such procedures rely on hardware support, so would only be possible if that is in place, and even in that case they may be very time consuming. It's also worth noting that flash storage will usually perform wear levelling, which defeats software techniques such as writing over a block multiple times. Collabora recommends not supporting this feature.

Stack Protection

It is recommended to enable stack protection, which provides protection against stack-based attacks such as a stack buffer overflow. Ubuntu, the distribution used as a base for Apertis has enabled a stack protection mechanism offered by GCC called SSP[39]. Modern processors have the capability to mark memory segments (like stack) executable or not, which can be used by applications to make themselves safer. Some initial tests with the Freescale kernel 2.6.38 provided on imx6 board shows correct enforcement behaviour.

Memory protection techniques like disabling execution of stack or heap memory are not possible with some applications, in particular execution engines such as programming language interpreters that include a just in time compiler, including the ones for JavaScript currently present in most web engines. Cases such as this and also cases in which the limitations should apply but are not being respected will be documented.

Collabora will also document best practices for building software with this feature so that others can take advantage of stack protection for higher level libraries and applications.

Confining applications in containers

LXC Containment

LXC[40] is a solution that was developed to be a lightweight alternative to virtualization, built on top of cgroups and namespaces, mainly. Its main focus is on

[39] https://wiki.ubuntu.com/GccSsp
[40] https://linuxcontainers.org/

servers, though. The goal is to separate processes completely, including using a different file system and a different network. This means the applications running inside an LXC container are effectively running in a different system, for all practical purposes. While this does have the potential of helping protect the main system, it also brings with it huge problems with the integration of the application with the system.

For graphical applications the X server will have to run with a TCP port open, so that applications running in a container are able to connect, 3D acceleration will be impossible or very difficult to achieve for applications running in a container. D-Bus setup will be significantly more complex.

Besides increasing the complexity of the system, LXC essentially duplicates functionality offered by cgroups, AppArmor, and the Netfilter firewall. When LXC was originally suggested it was to be used only for system services. By using systemd the Apertis system will already have every service on the system running on their own cgroup, and properly locked down by AppArmor profiles. This means adding LXC would only add redundancy and no additional value.

Protection for the driver assistance and limiting the damage root can do to the system can both be achieved by AppArmor policies, which can be applied to both system services and applications, as opposed to LXC, which would only be safely applicable to services. There are no advantages at all in using LXC for these cases. Limiting resources can also be easily done through cgroups, which will not be limited to system services, too. For these reasons Collabora recommends against using LXC.

Making X11, D-Bus and 3D work with LXC

For the sake of completeness, this section provides a description of possible solutions for LXC shortcomings.

LXC creates what, for all practical purposes, is a separate system. X supports TCP socket connections, so it could be made to work, but that would require opening the TCP port and that would be another interface that needs protection.

D-Bus has the same pros and cons of X11 – it can be connected to over a TCP port[41], but that again increases the surface area that needs to be protected, and adds complexity for managing the connection. It is also not a popular use case so it does not get a lot of testing.

3D over network has not yet been made to work on networked X. All solutions available, such as Virtual GL[42] involve a lot of copying back and forth, which would make performance suffer substantially, which is something that needs to be avoided given the high importance of performance on Apertis requirements.

[41] https://www.freedesktop.org/wiki/Software/DBusRemote/
[42] https://virtualgl.org/

Collabora's perspective is that using LXC for applications running on the user session adds nothing that cannot be achieved with the means described in this document, while at the same time adding complexity and indirection.

The Flatpak framework

Flatpak[43] is a framework for "sandboxed" desktop applications, under development by several GNOME developers. Like LXC, it makes use of existing Linux infrastructure such as cgroups (see Resource usage control) and namespaces.

Unlike LXC, Flatpak's design goals are focused on confining individual applications within a system, which makes it an interesting technology for Apertis. We recommend researching Flatpak further, and evaluating its adoption as a way to reduce the development effort for our sandboxed applications.

One secondary benefit of Flatpak is that by altering the application bundle's view of the filesystem, it can provide a way to manage major-version upgrades without app-visible compatibility breaks, by continuing to run app bundles that were designed for the old "runtime" in an environment more closely resembling that old version, while using the new "runtime" for app bundles that have been tested in that environment.

The IMA Linux Integrity Subsystem

The goal of the Integrity Measurement Architecture (IMA[44]) subsystem is to make sure that a given set of files have not been altered and are authentic – in other words, provided by a trusted source. The mechanism used to provide these two features are essentially keeping a database of file hashes and RSA signatures. IMA does not protect the system from changes, it is simply a way of knowing that changes have been made so that measures to fix the problem can be taken as quickly as possible. The authenticity module of IMA is still not available, so we won't be discussing it.

In its simpler mode of operation, with the default policy IMA will intercept calls that cause memory mapping and execution of a file or any access done by root and perform a hash of the file before the access goes through. This means execution of all binaries and loading of all libraries are intercepted. To hash a file, IMA needs to read the whole file and calculate a cryptographic sum of its contents. That hash is then kept in kernel memory and extended attributes of the file system, for further verification after system reboots.

This means that running any program will cause its file and any libraries it uses to be fully read and cryptographically processed before anything can be done with it, which causes a significant impact in the performance of the system. A 10% impact has been reported[45] by the IMA authors in boot time on a default

[43] https://flatpak.org/
[44] https://sourceforge.net/p/linux-ima/wiki/Home/
[45] https://blog.linuxplumbersconf.org/2009/slides/David-Stafford-IMA_LPC.pdf

Fedora. There are no detailed information on how the test was performed, but the performance impact of IMA is mainly caused by increased I/O required to read the whole of all executable and library files used during the boot for hash verification. All executables will take longer to start up after a system boot up because they need to be fully read and hashed to verify they match what's recorded (if any recording exists).

The fact that the hashes are maintained in the file system extended attributes, and are otherwise created from scratch when the file is first mapped or executed means that in this mode IMA does not protect the system from modification while offline: an attacker with physical access to the device can boot using a different operating system modify files and reset the extended attributes. Those changes will not be seen by IMA.

To overcome this problem IMA is able to work with the hardware's trusted platform module through the extended verification module (EVM[46]), added[47] to Linux in version 3.2: hashes of the extended attributes are signed by the trusted platform module (TPM) hardware, and written to the file system as another extended attribute. For this to work, though, TPM hardware is required. The fact that TPM modules are currently only widely available and supported for Intel-based platforms is also a problem.

Conclusion regarding IMA and EVM

IMA and EVM both are only useful for detecting that the system has been modified. They do so using a method that incurs significant impact on the performance, particularly application startup and system boot up. Considering the strict boot up requirements for the Apertis system, this fact alone indicates that IMA and EVM are suboptimal solutions. However, EVM and IMA also suffer from being very new technologies as far as Linux mainline is concerned, and have not been integrated and used by any major distributions. This means implementing them in Apertis means incurring into significant development costs.

In addition to that, Collabora believes that the goals of detecting breaches, protecting the base system and validating the authenticity of system files are attained in much better ways through other means, such as keeping the system files separate and read-only during normal operation, and using secure methods for installing and updating software, such as those described in Protecting the driver assistance system from attacks.

For these reasons Collabora advises against the usage of IMA and EVM for this project. An option to provide some security for the system in this case is making it hard to disconnect and remove the actual storage device from the system, to minimize the risk of tampering.

[46] https://sourceforge.net/p/linux-ima/wiki/Home/#linux-extended-verification-module-evm

[47] https://kernelnewbies.org/Linux_3.2#head-03576b924303bb0fad19cabb35efcbd33eeed084

Seccomp

Seccomp[48] is a sandboxing mechanism in the Linux kernel. In essence, it is a way of specifying which system calls a process or thread should be able to make. As such, it is very useful to isolate processes that have strict responsibilities. For instance, a process that should not be able to write or read from the disk should not be able to make an *open* system call.

Most security tools that were discussed in this document provide a system-wide infrastructure and protect the system in a general way from outside the application's process. As opposed to those, seccomp is something that is very granular and very application-specific: it needs to be built into the application source code.

In other words, applications need to be written with an architecture which allows a separation of concerns, isolating the work that deals with untrusted processes or data to a separate process or thread that will then use seccomp filters to limit the amount of damage it is able to do through system calls.

For use by applications, seccomp needs to be enabled in the kernel that is shipped with the middleware. There is a library called **libseccomp**[49], which provides a more convenient way of specifying filters. Should feature be used and made it available through the SDK, the seccomp support can be enabled in the kernel and libseccomp can be shipped in the middleware image provided by Collabora.

The seccomp filter should be used on system services designed for Apertis whose architecture and intended functionality allow dropping privileges. Suppose, for instance, that Apertis has a health management daemon which needs to be able to kill applications that misbehave but has no need whatsoever of writing data to a file descriptor. It might be possible to design that daemon to use seccomp to filter out system calls such as **open** and **write**. The **open** system call might need to be allowed to go through for opening files for reading, depending on how the health daemon monitors processes – it might need to read information from files in the **/proc** file system, for instance. For that reason, filtering for **open** would need to be more granular, just disallowing it being called with certain arguments.

Depending on how the health management daemon works it would also not need to fork new processes itself, so filtering out system calls such as **fork**, and **clone** is a possibility. As explained before, to take advantage of these opportunities, the architecture of such a daemon needs to be thought through from the onset with these limitations in mind. Opportunities, such as the ones discussed here, should be evaluated on a case-by-case basis, for each service intended for deployment on Apertis.

[48] https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
[49] https://lwn.net/Articles/494252/

AppArmor and seccomp are complementary technologies, and can be used together. Some of their purposes overlap (for example, denying filesystem write access altogether could be achieved equally well with either technology), and they are both part of the kernel and hence in the TCB.

The main advantage of seccomp over AppArmor is that it inhibits all system calls, however obscure: all system calls that were not considered when writing a policy are normally denied. Its in-kernel implementation is also simpler, and hence potentially more robust, than AppArmor. This makes it suitable for containing a module whose functionality has been designed to be strongly focused on computation with minimal I/O requirements, for example the rendering modules of browser engines such as WebKit2. However, its applicability to code that was not designed to be suitable for seccomp is limited. For example, if the confined module has a legitimate need to open files, then its seccomp filter will need to allow broad categories of file to be opened.

The main advantage of AppArmor over seccomp is that it can perform finer-grained checking on the arguments and context of a system call, for example allowing filesystem reads from files owned by the process's uid, but denying reads from other uids' files. This makes it possible to confine existing general-purpose components using AppArmor, with little or no change to the confined component. Conversely, it groups together closely-related system calls with similar security implications into an abstract operation such as "read" or "write", making it considerably easier to write correct profiles.

The role of the app store process for security

The model which is used for the application stores should precludes automated publishing of software to the store by developers. All software, including new versions of existing applications will have to go through an audit before publishing.

The app store vetting process will generate the final package that will reach the store front. That means only signatures made by the app store curator's cryptographic keys will be valid, for instance. Another consequence of this approach is that the curator will have not only the final say on what goes in, but will also be able to change pieces of the package to, say, disallow a given permission the application's author specified in the application's manifest.

This also presents a good opportunity to convert high level descriptions such as the permissions in the manifest and an overall description of files used into concrete configuration files such as AppArmor profiles in a centralized fashion, and provides the curator with the ability to fine tune said configurations for specific devices or even to rework how a given resource is protected itself, with no need for intervention from third-parties.

Most importantly, from the perspective of this document, is the fact that the app store vetting process provides an opportunity for final screening of submissions for security issues or bad practices both in terms of code and user interface, so that should be taken into consideration.

How does security affect developer usage of a device?

How security impacts a developer mode depends heavily on how that developer mode of work is specified. This chapter considers that the two main use cases for such a mode would be installing an application directly to the target through the Eclipse *install to target* plugin and running a remote debugging session for the application, both of which are topics discussed in the SDK design.

The *install to target* functionality that was made available through an Eclipse plugin uses an **sftp** connection with an arbitrary user and password pair to connect to the device. This means that putting the device in developer mode should ensure the **ssh** server is running and add an exception to the firewall rules discussed in Network filtering, to allow an inbound connection to port 22.

Upon login, the SSH server will start user sessions that are not constrained by the AppArmor infrastructure. In particular the white-list policy discussed in section Implementing a white list approach, will not apply to ssh user sessions. This means the user the IDE will connect with needs file system access to the directory where the application needs to be installed or be able to tell the application installer to install it.

The procedure for installing an application using an **sftp** connection is not too different from the *install app from USB stick* use case described in the Applications document, that similarity could be exploited to share code for these features.

The main difference is the developer mode would need to either ignore signature checking or accept a special "developer" signature for the packages. Decision on how to implement this piece of the feature needs a more complete assessment of proposed solutions on how the app store and system DRM could work, and how open (or openable) the end user devices will be.

Running the application for remote debugging also requires that the **gdb-server**'s default port, 2345, be open. Other than that, the main security constraint that will need to be tweaked when the system is put in developer mode is AppArmor. While under developer mode AppArmor should probably be put in complain mode, since the application's own profile will not yet exist.

Further discussion

This chapter lists topics that require further thinking and/or discussion, or a more detailed design. These may be better written as Wiki pages rather than formal designs, given they require and benefit from iterating on an implementation.

- Define which cgroups (Resource usage control) to have, how they will be created and managed
- Define exactly what Netfilter rules (Network filtering) should be installed and how they will be made effective at boot time
- Evaluate Flatpak (The Flatpak framework)