Security Architecture Overview

19 October, 2016 | Access Control, DAC, MAC, LSM and more...

Gunnar Andersson

Lead Architect, GENIVI Alliance

This work is licensed under a Creative Commons Attribution-Share Alike 4.0 (CC BY-SA 4.0)
GENIVI is a registered trademark of the GENIVI Alliance in the USA and other countries.
Copyright © GENIVI Alliance 2016.

Abstract and introduction

Abstract

Main topic:

A secure system is more than **access control**, but this is where we build our foundation (after a trusted boot process).

It has been the primary topic of interest in the *open-source* systems we have compared in the past (Tizen IVI, AGL, Apertis, Android...) which makes it relevant to discuss from a GENIVI standards point of view.

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Purpose

- S*et the stage* for the following case-studies in this presentation track, and make a few concrete proposals
- Warning: Some very basic principles will be covered. For the security experts in the room, realize that the purpose is not always to teach something new to you
- For efficient collaboration, getting everyone "on the same page" is worth covering some basics. This should facilitate further collaboration.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Questions (as promised in Abstract)

- What does the GENIVI architecture say so far about different patterns and options for controlling access to data and resources?

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 |

Questions

- What are the basic expectations of system design inherent in some GENIVI components today?

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Questions

- What are the differences between Linux's Discretionary and Mandatory Access Control?

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Questions

- Does using user IDs to identify individual applications (Android style) make it impossible to properly protect user confidential data?

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 |

Questions

- Beginner's intro into available options for Linux Security Modules options.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0

Questions

- What are the limits of using the vanilla UNIX users-and-groups system to model fine-grained access control?

GENIVI®

Questions

- How do we as an alliance recommend to apply multiple security techniques?

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Questions

- Finally, from a GENIVI standards point of view, is it possible to speak of an access control architecture that could be realized using different technical solutions?

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Not covered today

Kernel development... a lot happening.

- Kernel hardening
  - Additional memory protection, etc.
    - Long term projects - Grsecurity, RSBAC.
  - Stacked LSMs.
- Networking
  - Lots to consider, and some new development
- Seccomp and new proposals (Secomp+BPF)
- POSIX Capabilities (read: $ man 7 capabilities)
- New IPCs: kdbus → bus1?

GENIVI®

Principle of least privilege

Principle of least privilege

This is what we are aiming for, for every individual part in the system:

Every Actor (process) shall have access to only the resources (files, system resources, ...) that is necessary for its function, and none other.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 |

Discretionary vs Mandatory Access Control

Dictionary definition

Discretionary

adjective dis·cre·tion·ary \-ˈkre-shə-ˌner-ē\

: available to be used when and how you decide

: done or used when necessary

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Dictionary definition

Mandatory

adjective man·da·to·ry \ˈman-də-ˌtȯr-ē\

: required by a law or rule

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Discretionary Access Control (DAC)

Standard file permissions. Built into all UNIX/Linux

1. File owner can (within rules) modify the permissions
2. Checked on file open only
3. A valid file handle stays valid
4. Open file handle can be passed to another process (within rules)

GENIVI®

Mandatory Access Control (MAC)

...as implemented by Linux Security Modules(LSM)

1. Users cannot modify policy
2. Checked on every new access
3. (Potentially) rich policy language / flexible rules

GENIVI®

Access control setup

Introduction

Before talking about solutions let's bring this up a level to the very basics.

We have a set of **Actors**

to access a set of **Resources**

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Introduction

Before talking about solutions let’s bring this up a level to the very basics.

Translation:

Linux Processes access
Files (and IPC*, and Networking*, Hardware*
and Kernel syscalls, and ...)

(* much of which is modelled as files in UNIX)

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

User ID approaches

Traditional UNIX (desktop) approach:

- User IDs, one per person
  - **alice**, **bob**, **charlie**, **dan**

Alternative approach for application separation:

- User IDs, one per *application*
  - **mediaplayer***,* **browser***, ...*

= Android approach

(later modified for multi-user support and hardened security)

GENIVI®

Access matrix (principle)

ACTORS (processes)	RESOURCES (interfaces) AudioManager:: AudioInterface	Audiomanager:: PrioritySound	Addressbook:: GetAddress	Networking:: InternetAccess
Mediaplayer				
Browser				
NodeState Manager				
Navigation Application				

GENIVI®

Access matrix (IPC interfaces)

ACTORS (processes)	RESOURCES (IPC interfaces) AudioManager:: AudioInterface	Audiomanager:: PrioritySound	Addressbook:: GetAddress	Networking:: InternetAccess
Mediaplayer	YES	**NO**	**NO**	**NO**
Browser	YES	**NO**	**NO**	YES
NodeState Manager	**NO**	**NO**	**NO**	**NO**
Navigation Application	YES	YES	YES	**NO**

GENIVI®

Access matrix (files)

ACTORS	FILE or DIRECTORY /foo/bar	/etc/netwk.config	addressbook.dat	networksocket
A	Read	**Write**	**--**	Read
B	**Write**	Read	Read	--
C	Read	**Write**	Read	--
D	**Write**	--	--	**--**

GENIVI®

Access matrix (files)

Typical DAC solution

Owner write access
Group read access
Other no access

rw- r-- --- alice users /foo/bar

ACTOR	FILE File 1	File 2	File 3	File 4
A	Read	Write	--	Read
B	Write	Read	Read	--
C	Read	Write	Read	--
D	Write	--	--	--

GENIVI®

Access matrix (files)

perms	owner	group	filename
rw- r-- ---	bob	grp1	File1
rw- r-- ---	alice	grp2	File2
rw- r-- ---	charlie	grp3	File3
rw- r-- ---	dan	grp4	File4

Guaranteed to work for an arbitrary matrix only if:
- one writer, multiple readers
- no other access modes (e.g. execute) are modeled.

=> Limits of expression

Group membership:

alice ~=grp1,grp4

bob ~= grp2,grp3

charlie ~=grp1,grp2

dan ~=

ACTOR	FILE File 1	File 2	File 3	File 4
A(lice)	Read	Write	--	Read
B(ob)	Write	Read	Read	--
C(harlie)	Read	Read	Write	--
D(an)	Write	--	--	--

GENIVI®

Access matrix (files)

We had **Read** and **Read/Write**

Add **Executable** bit into the previous matrix,
and you are already struggling with the expressivity
of the users and groups model.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Access Matrix

Key point:

To fully model an arbitrary access matrix
there is a certain expressivity (degrees of freedom)
required.

Problem: You may run out of degrees of freedom...

Solution: Add an independent (orthogonal) mechanism.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Conclusions about DAC?

Users and group permissions **do** work reliably doing exactly what they do, but nothing more…

Be aware of limits

- File open → file handle stays valid forever.
- File owners can change permissions.

Not always enough expressivity

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

DAC, MAC. diff?

Users/groups DAC:

Owner can change permissions!

Checked on open only

– a valid filehandle stays valid

– a valid filehandle can be passed on (conditionally)

MAC (normally):

Owner cannot change permissions!

Arbitrary restriction, e.g. fail on any access

GENIVI®

DAC, MAC. diff?

Granularity

Users/groups DAC:	Read, Write, Execute
Smack modes	Read, Write, Execute (Append, Transmute)

GENIVI®

Reason to use LSM/MAC?

Not only do the semantics help and in *some* cases provide a stronger lock down...

Typically you need at least one more independent mechanism to be sufficiently expressive

Some LSM/MAC are more expressive, i.e. have more freedoms in themselves, but that is rare.

The point is COMBINING multiple independent mechanisms.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Use multiple mechanisms
Example: Tizen IVI

User ID used to separate (person) user data

Smack to separate application data (coarse-grained)

Manifest and a **separate database** for fine-grained services restrictions

(+**Groups** to cover some needs)

GENIVI®

Use multiple mechanisms
Example: AGL

Similar.
I defer to the upcoming presentation for the details ☺

GENIVI®

GENIVI Reference Architecture

User ID approaches

Traditional UNIX (desktop) approach:

– User IDs, one per person
  - **alice**, **bob**, **charlie**, **dan**

Alternative approach to achieve application separation:

– User IDs, one per *application*
  - **mediaplayer***,* **browser***,*

= Android approach

(Modified for multi-user support and hardened security)

GENIVI®

https://developer.android.com/guide/topics/security/permissions.html :

"Android is a privilege-separated operating system, in which each application runs with a distinct system identity (**Linux user ID** and **group ID**)"

"Parts of the system are also separated into distinct identities. Linux thereby isolates applications from each other and from the system."

User ID per application is a “Misuse of user IDs” ?

Long-running processes (daemons) have always run with unique User IDs. Look for yourself:

$ cat /etc/passwd

(www, apache, spool, mail, ftp, sshd, gdm, pulse, nobody...)

GENIVI®

Noteworthy early design decisions

Persistence API (low-level)

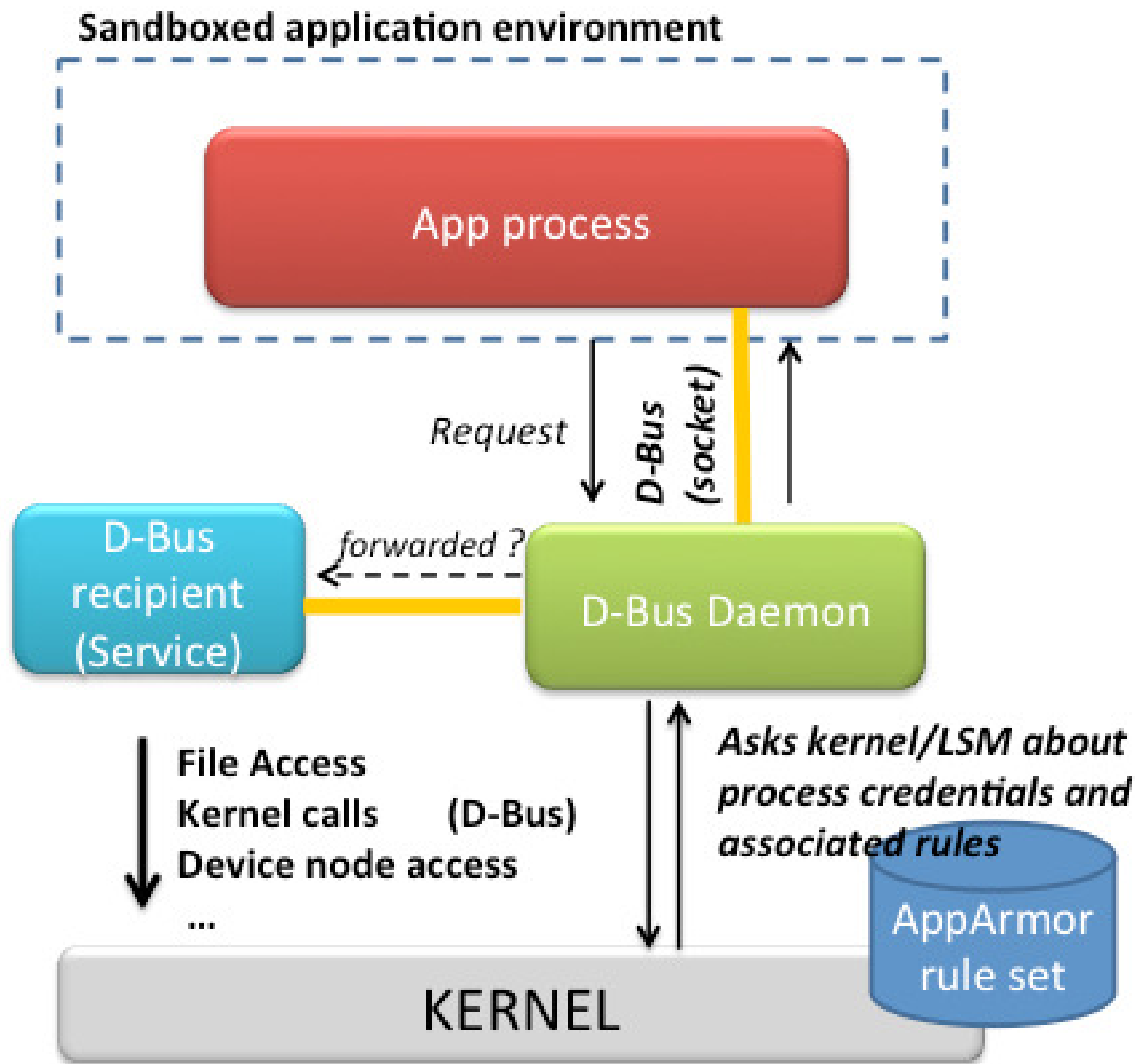
Abstract addressing persistence resource

pclKeyWriteData(unsigned int **ldbid**, const char* **resource_id**,
unsigned int **user_no**,
unsigned int **seat_no**,
unsigned char* data, int size);

How is this secure for user data?
And why is it needed?

- Users are at different seats.
  A seat is an abstract concept and can be mapped differently in the actual vehicle
  Caller must specify User ID for user-specific data
  ... Why not just read the application's Effective User ID?

GENIVI®



AppArmor

(E.g. Apertis)

GENIVI

Copyright © GENIVI Alliance 2016

External policy database (E.g. Tizen IVI)

Sandboxed application environment

App process

Request,

Success or denial

Service access wrapper

Service access wrapper

Request allowed?

yes/no

ACL database

File Access
Kernel calls (D-Bus)
Device node access
...

Service component

Device node access

KERNEL

Copyright © GENIVI Alliance 2016

Keeping user data private

- If processes are run with different (personal) Effective User ID for each “logged in” person, the theory is:
  - Data files will be written so that the user becomes **file owner**
  - Data files will be set up so that only the **owner** can read them
  - Data files are set up so that others except **owner** cannot read them.
  - Processes cannot fake their Effective User ID
  - -> An exploit or malicious application cannot leak personal data.
  - Leak? Means to share one person’s data with another person.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private

- If processes are run with one Effective User ID (long lived daemon)
  - Typically these are claimed to be "insecure" because it is feasible for a process to read all (any) user data at the same time.
  - An exploit does indeed give more direct access to all user's data.

  But...

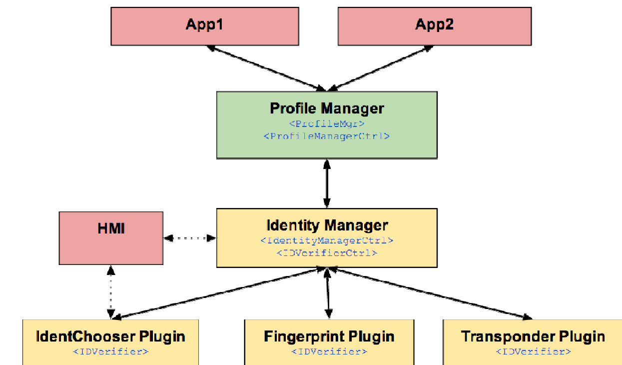
  1. There are reasons for long-lived processes, even "applications"
  2. Stay tuned... We evaluate if the difference in private data leak risk

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Profile Manager distributes the system opinion about the current active user (multiple users, one per seat) to other processes.

PM has a protocol to switch all applications together (synchronized)



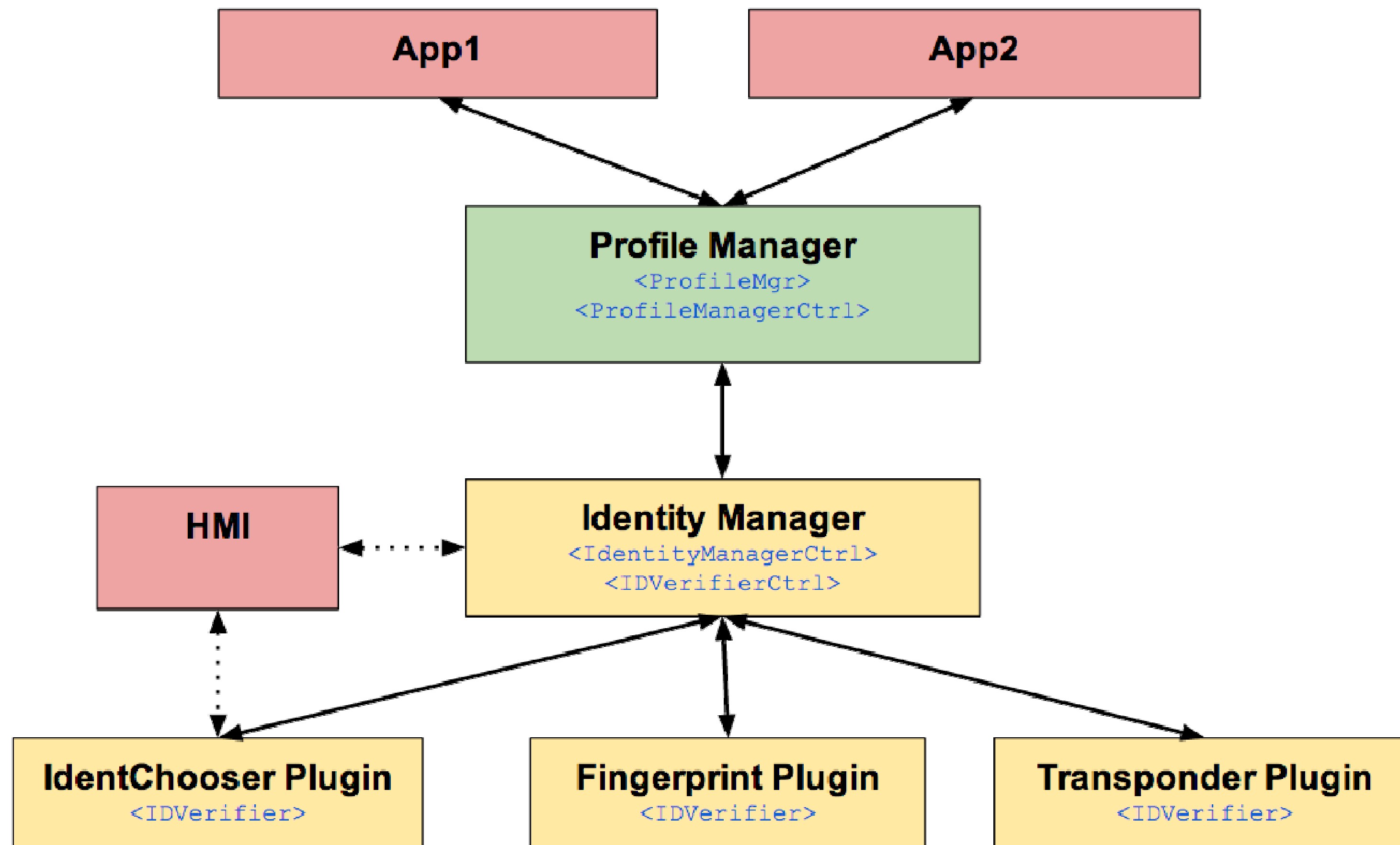
Long-lived processes do not restart just to get a different Effective User-ID

This principle does not make it impossible to also have processes that are restarted with different Effective User ID

GENIVI®

Copyright © GENIVI Alliance 2016

User management subsystem



GENIVI®

Different strokes...for different processes

1. **Infrastructure**/**Platform** ("built-in", vetted, long-lived)
2. **Native*** Applications ("built-in", vetted, long-lived)
3. **Managed** Applications (per-user, 3rd-party, short-lived)

* Note that in GENIVI architecture "Native" Application definition has nothing to do with whether it is compiled or interpreted code.

Different process needs

Managed application Space

Appl

Appl

Launches/controls/monitors

Application Manager

Services,
Daemons

Native Application Space

Native
Applications

Launches/controls/monitors

Launches/controls/monitors

GENIVI Lifecycle subsystem (Similar arrangement for **User Mgmt** & **Persistence**)

systemd

Node State
Manager

NSMc

Node Resource
Manager

Node Health
Monitor

Copyright © GENIVI Alliance 2016

User-aware **Platform Components** and **Native applications**, both of which are assumed to run like daemons and not as “user session” processes, are defined because of requirements on the GENIVI platform.

Native applications are defined **because** OEM requirements had a need for long-lived applications. They *shall not*, as required, restart themselves simply because a user switch happens.

(Consider User Experience of Navi, Media, etc not resetting just because you switch driver, but still providing some user-awareness)

This is by design, and they are as secure as they can be made. These processes are likely privileged to read any user data at any time and need to be carefully vetted for that reason.

Other options:

Decouple long-lived process and user-aware process.

I.e. delegate the reading of user data to a smaller process that is running (restarted) with access to one user data at a time.

GENIVI®

Questions

- Does using user IDs to identify individual applications (Android style) *really* make it impossible to properly protect user confidential data?

GENIVI®

Keeping user data private

Persistence API – User ID is sent in, not inferred from process effective user ID.
BUT:

- A file based persistence back-end is set up so that personal data is written with the actual effective user ID. In other words, the data file is still protected **if** you have made sure of this by running the process with the right Effective User ID.
- Group ID is used to set up application/shared data (see persistence for details)
- Since persistence backends can vary, this varies (system design needed here).

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private

GENIVI Reference Architecture says:

Application manager for ***Managed*** apps
can run its apps with Effective User ID = personal login ID.
It's a valid choice and nothing prevents it.

(In theory nothing prevents it for ***Native*** apps either but it is not the default expected behavior.)

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private

But how much more secure
is a “logged in” approach?

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private alternative 1

Consider "MalApp" running with the same Effective User ID no matter who is logged in*

or

"MalApp" is restarted with different User ID (logged in user).

This is a possible setup for *Managed Applications*

(*More generally defined – Consider that the trusted system *somehow* ensures the untrusted application can only access data of the currently logged in user)

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private alternative 2

“MalApp” runs the same Effective User ID
(or can otherwise access any user’s data at the same time).

This is the default *Native Application* approach.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private

Attack approach 1:

- When *Dan* is logged in **MalApp** reads Alice's data (which it can get access to because it is not protected by the system per-user). **MalApp** can leak that data to *Dan.*

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private

Attack approach 2:

- When *Alice* is logged in **MalApp** reads Alice's data and stores it in any shared or application-level storage*.
  If MalApp has internet access*, it stores data in the cloud.
- Then, when *Dan* is logged in MalApp reads back data from the shared storage and leaks Alice's data to Dan.

*(*Yes, a few additional assumptions but very often these are true)*

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Keeping user data private

Attack approach 2 is just as feasible* for an application that only has access to one person’s data at any single time.

Conclusion: Design 2 may be partly more secure for a short time, but from a fundamental point of view it is often not.

=> If we share sensitive data with an application *at any time*, then *all bets are off*.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

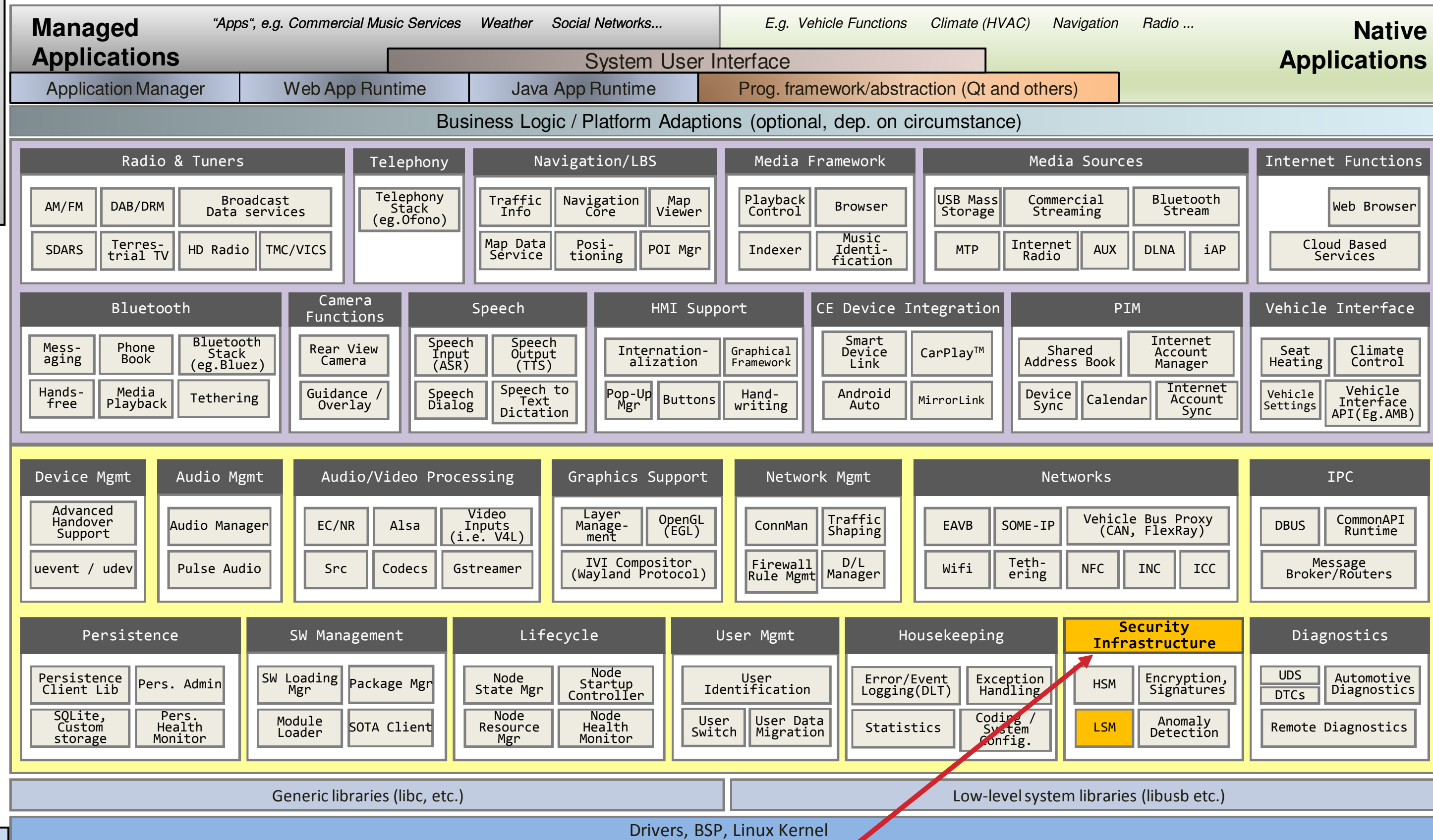
GENIVI Software Reference Architecture

Block Diagram 2.1

Upper Platform (Middleware)

Lower Platform (Infrastructure)

Initial Bootloader



Security throughout, but particular supporting functions are part of this subsystem

Linux Security Modules

Linux Security Modules

"Plugin" enabled at kernel compile. Adds additional permission checks.

Simplified flow:

KERNEL asks itself: "Can this process perform this access?"

1. Does the process have the required CAPABILITY? Yes/No
2. (DAC) Is the effective user allowed the access? Yes/No
   (e.g. file owner + permission bits)
3. Is there an LSM in the kernel or not? Yes/No

   If yes – ask LSM (MAC) the same question.

(Seccomp is already passed at this point)

(* Stacked (multiple) LSMs → ask next one)

First denied access breaks the chain and returns.
If all checks return permitted the operation is permitted.

Linux Security Modules

Label based:

Smack

SELinux

Path/executable rule based:

AppArmor
Tomoyo

Minor / targeted:

Yama

Trends:

* more LSMs, stacking
* liblsm (shared code...)

Others left out for brevity...Google

GENIVI®

Linux Security Modules

Label based:

- Process label compared against file label.
- Labels are independent of user/group.
- Flexible matching rules.

  Example from Smack. SE-Linux has more complex semantics.

  - Any process labeled "audio" can open socket labeled with "audiosink" for writing. ← arbitrary example
  - Any process labeled with "^" (wildcard) can open any file ← built in rule

GENIVI®

Access Control
Why and how - details

Definitions

In here, as well as GENIVI Reference Architecture:

- **Coarse-grained** access control is basically dividing actors into categories or groups and drawing up the fundamental battle lines that limit access.
  - For example, particular processes only have access to hardware devices
  - The application space may use the services layer, but never direct hardware access, etc.
  - System files are closed off from access unless highly privileged
- **Fine-grained** policy means to put access limits on individual interfaces and sometimes partial interfaces (methods).
  - For example, one app may be allowed to make a phone call over Bluetooth, but never to add new pairing of phones.
    This is a *partial* access to the Bluetooth service

MORE INFO: GENIVI Reference Architecture Document, Application Framework chapter.

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Previous work (1)

- Tizen IVI team taught us all a valuable lesson. In an early attempt, a fine-grained and comprehensive **Smack** policy was tried.
- To finely tune which actors had access to which interfaces using Smack failed on its own complexity
- Tizen IVI went back to setting up the fundamental large-scale, coarse-grained, access control rules with Smack, and deferring fine-grained policy to a separate manifest and a separate security policy daemon.
- This approach proved successful and can be recognized (even if further refined) in AGL today.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Previous work (2)

- The Apertis system uses **AppArmor** to lock down all components access, both system components and applications.
- **AppArmor** provides a rich language and tools that help manage complexity of fine-grained policies
- By that approach, **Apertis** manages to both encode the system-level rules, and fine-grained manifest of applications, keeping all the security checks into the Kernel.

(Look out for an update in this week's Application Framework session)

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Linux Security Modules in GENIVI Architecture

- GENIVI recommends implementers of Compliant platforms to apply multiple layers of security... including application of LSMs for Mandatory Access Control and other purposes.
- At the moment, platform vendors are applying different choices of LSMs
  - **SELinux** and **AppArmor** are popular choices among platform vendors.
- We recognize **Smack** as a reasonable and good choice by AGL
- Requiring a (non-Smack) Mandatory Access Control solution, as used by GENIVI platform vendors, would make it impossible to make use of AGL to create a compliant system.
  It would reduce our ability for important collaboration.
- Therefore the GENIVI specification does not (can not) mandate only one.

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Other security mechanisms in GENIVI Architecture

- GENIVI recommends implementers of Compliant platforms to apply multiple layers of security...
  - Optionally use namespaces, if appropriate, for example in application management space
  - Set resource usage and other limits using control groups
  - Augment with seccomp and kernel hardening as appropriate

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Final conclusions

- Use tools to manage the complexity of a full **access control matrix.**
- A Franca IDL + Component Description Language would be a great foundation for code-generating the specific policies against different mechanisms. (ref: Klaus Uhl presentation GENIVI AMM Paris)
- Several companies do this. It’s time for a shared standard!
- Alliance/community work needed in this area

GENIVI®

Copyright © GENIVI Alliance 2016 CC BY-SA 4.0 | AMM Fall 2016

Thank you!

Visit GENIVI at http://www.genivi.org or http://projects.genivi.org

Contact us: help@genivi.org

BACKUP SLIDES

Expressivity differs

Users/groups DAC:

Process Effective User ID
Process Effective Group ID

File Owner (user) and Group
and R,W,X permission bits

SELinux:

Process Security Context
Resource Security Context

Resource class (type) ← any number of types

Read, Write, Execute permission rules

GENIVI®

Expressivity (2)

Users/groups DAC: Process Effective User ID – ONE (1)

Matched against ONE (1) File Owner

Matched against ANY number of Groups

User can be a member of: – ANY group

and R,W,X permission bits

SELinux

Process Security Context – ONE
Resource Security Context – ONE

Resource class (type) ← ANY number of types

Read, Write, Execute permission rules

GENIVI®