**MANDATORY ACCESS CONTROL SECURITY ENHANCEMENT**

LGE DV CYBER SECURITY TEAM – NGUYEN.TRAN

Contents

[1. Overview of the Task 3](#_Toc180362364)

[1.1. Assignment Performance Period 3](#_Toc180362365)

[1.2. Project Title 3](#_Toc180362366)

[1.3. Background of Assignment Selection 3](#_Toc180362367)

[1.4. Project Goals 3](#_Toc180362368)

[1.5. Project Outputs 3](#_Toc180362369)

[2. System Definition 4](#_Toc180362370)

[2.1. General Automotive System Architecture 4](#_Toc180362371)

[2.2. Introduction to MAC 5](#_Toc180362372)

[2.3. Introduction to SMACK 5](#_Toc180362373)

[2.3.1. Access Rules 8](#_Toc180362374)

[2.3.2. Defining Access Rules 9](#_Toc180362375)

[2.3.3. SMACK Capabilities 9](#_Toc180362376)

[2.3.4. Security principles that SMACK developer need to pay attention 9](#_Toc180362377)

[2.3.4.3. Make MAC service ready on time 12](#_Toc180362378)

[2.3.4.4. Don’t give another label access Privileged label 12](#_Toc180362379)

[2.3.5. SMACK Conclusion 14](#_Toc180362380)

[2.4. Introduction to AppArmor 14](#_Toc180362381)

[2.4.1 AppArmor Diagram 14](#_Toc180362382)

[2.4.2. Sequence Diagram 16](#_Toc180362383)

[3. Asset Identification 17](#_Toc180362384)

[3.1. Asset Identification Process 17](#_Toc180362385)

[3.2. Asset List and Damage Scenarios 18](#_Toc180362386)

[4. Security Threat Analysis 18](#_Toc180362387)

[4.1. System Structure Representation (DFD) 18](#_Toc180362388)

[4.2. Threat Modeling 18](#_Toc180362389)

[5. Risk Assessment 18](#_Toc180362390)

[6. Security Requirement 18](#_Toc180362391)

[7. Mitigation and Implementation 18](#_Toc180362392)

[8. Conclusion 18](#_Toc180362393)

[9. Future Work 18](#_Toc180362394)

# 1. Overview of the Task

## 1.1. Assignment Performance Period

August, 2024 ~ December, 2024

## 1.2. Project Title

Mandatory Access Control Security Enhancement

## 1.3. Background of Assignment Selection

* When deployed correctly, MAC has the potential to mitigate the damage of 0day vulnerabilities.
* It’s almost impossible for all programs in the system to be invulnerable, preventing compromised process damaging other components in the system is MAC’s job, it’s the last gate when other security gates have been defeated.
* The above 2 reasons show us the importance of applying MAC to our security toolkit, OEMs also realize that so there are many requirements related to MAC.
* Automotive cyber security have recently caught great attention from government organization. Different regions and countries have their own cyber security regulation.

## 1.4. Project Goals

This project focuses on improving the security of MACs by providing a reasonable setup to minimize the damage to the system when an app has been compromised with the following assumptions:

* As mentioned above, implementing an application without any security flaws is almost impossible. So all apps can be considered as attack vectors.
* It can be added that if we assume that all apps cannot be attacked by hackers, then there is no reason for us to apply MAC to the system.

With that in mind, the major goals of this project are as follows:

* Point out weaknesses in our old implementation of MAC that can be exploited to learn from and then propose solutions to patch the weaknesses.
* Provide security principles and how to install MAC to ensure those principles, for example:
  + Applications have access controlled so that they have least privileged
  + MAC availability must be guaranteed so that there is no downtime (vulnerable).
* Improve security while ensuring system performance.

## 1.5. Project Outputs

* Software architecture design of MAC-Security Enhanced
* Threat analysis and risk assessment of MAC in the context of automotive ECU environment

# 2. System Definition

## 2.1. General Automotive System Architecture

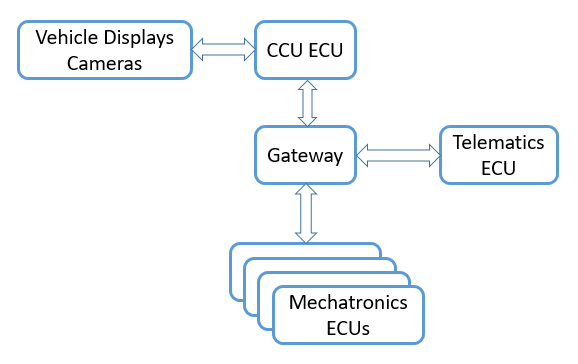


Figure 1 Automotive System Architecture

Automotive system architecture of may differ from OEM to OEM, but consists of the following basic components, as shown in <Figure 1 Automotive System Architecture>:

- CCU (Central Computing Unit) ECU houses core compute functionality for the entire vehicle. The CCU houses multiple "units of compute" that implement the core processing functions for In-Vehicle Infotainment (IVI) and Advanced Driver Assistance (ADAS)

- Mechatronics ECUs is a set of ECU that are responsible for controlling mechanical components of the vehicle. These components includes doors, seats, brakes, air-conditioner, windshield wiper …

- Telematics ECU enables software running in cars to continuously send and receive data off-vehicle. It serves as a single point of entry and exit for all data connectivity from the vehicle

CCU ECU, Telematics ECU, and the Gateway forms an Ethernet network. Mechatronics ECUs and the Gateway forms a CAN (Controller Area Network) network.

Telematics ECU is a key product of LGE VS (Vehicle Solution) Company. It combines advanced technology and connectivity to deliver efficient and intelligent solutions for various industries. With its robust features and capabilities, LG Telematics ECU enables businesses to optimize fleet management, improve operational efficiency, enhance driver safety, and gain valuable insights through data analytics. Whether it's tracking vehicle location, monitoring driver behavior, or managing maintenance schedules, LG Telematics ECU offers a reliable and scalable solution tailored to the needs of modern businesses.

## 2.2. Introduction to MAC

Mandatory access control (MAC) is a security strategy that restricts the ability individual resource owners have to grant or deny access to resource objects in a file system. MAC criteria are defined by the system administrator, strictly enforced by the operating system (OS) or security kernel, and cannot be altered by end users.

MAC is a method or access control policy aimed at restricting access to a resource (also known as an object) based on two key factors: the sensitivity of the information contained in that resource and the authorization level of the user trying to access that resource and its information.

There are some MAC (Mandatory Access Control) security frameworks commonly used in various operating systems:

* **SMACK (Simplified Mandatory Access Control Kernel)**: SMACK is a Linux kernel security module designed for simplicity and performance. It provides a simplified MAC framework that labels processes and files with security attributes and enforces access control based on these labels. SMACK is known for its ease of use and integration with existing Linux systems.
* **SELinux (Security-Enhanced Linux)**: Developed by the National Security Agency (NSA), it provides a flexible and fine-grained MAC system for Linux distributions. SELinux enforces access control policies based on security labels attached to files, processes, and other system objects. The policies define rules for access control, specifying which subjects (processes or users) can access which objects (files, directories, devices) under what conditions.
* **AppArmor**: AppArmor is a MAC system for Linux that focuses on confining individual programs instead of entire system policies. It works by defining profiles for specific applications, restricting their capabilities and access to resources based on these profiles. AppArmor uses path-based profiles to control access to files and resources, making it easier to implement and manage compared to SELinux.

These MAC frameworks enhance the security of operating systems by enforcing access control policies beyond traditional discretionary access controls (DAC). They help prevent unauthorized access and improve system security by limiting the actions that processes and users can perform based on predefined security policies.

In the next part we will talk more about SMACK and AppArmor.

## 2.3. Introduction to SMACK

Smack is the Simplified Mandatory Access Control Kernel. Smack is a kernel based implementation of mandatory access control that includes simplicity in its primary design goals.

Smack is not the only Mandatory Access Control scheme available for Linux. Those new to Mandatory Access Control are encouraged to compare Smack with the other mechanisms available to determine which is best suited to the problem at hand.

Smack consists of three major components:

* The kernel
* Basic utilities, which are helpful but not required
* Configuration data

The kernel component of Smack is implemented as a Linux Security Modules (LSM) module. It requires *netlabel* and works best with file systems that support extended attributes, although *xattr* support is not strictly required. It is safe to run a Smack kernel under a *“vanilla”* distribution.

Smack is an extension to a Linux system. It enforces additional restrictions on what subjects can access which objects, based on the labels attached to each of the subject and the object.

There are four terms that are used in a specific way and that are especially important:

* *Subject:*A subject is an active entity on the computer system. On Smack a subject is a task, which is in turn the basic unit of execution.
* *Object:*An object is a passive entity on the computer system. On Smack files of all types, IPC, and tasks can be objects.
* *Access:*Any attempt by a subject to put information into or get information from an object is an access.
* *Label:*Data that identifies the Mandatory Access Control characteristics of a subject or an object.

These definitions are consistent with the traditional use in the security community. There are also some terms from Linux that are likely to crop up:

* *Capability:*A task that possesses a capability has permission to violate an aspect of the system security policy, as identified by the specific capability. A task that possesses one or more capabilities is a privileged task, whereas a task with no capabilities is an unprivileged task.
* *Privilege:*A task that is allowed to violate the system security policy is said to have privilege. As of this writing a task can have privilege either by possessing capabilities or by having an effective user of root.

The *Figure 2* below shows an architecture diagram for SMACK (Simplified Mandatory Access Control Kernel), a security framework in Linux that provides access control based on security labels.

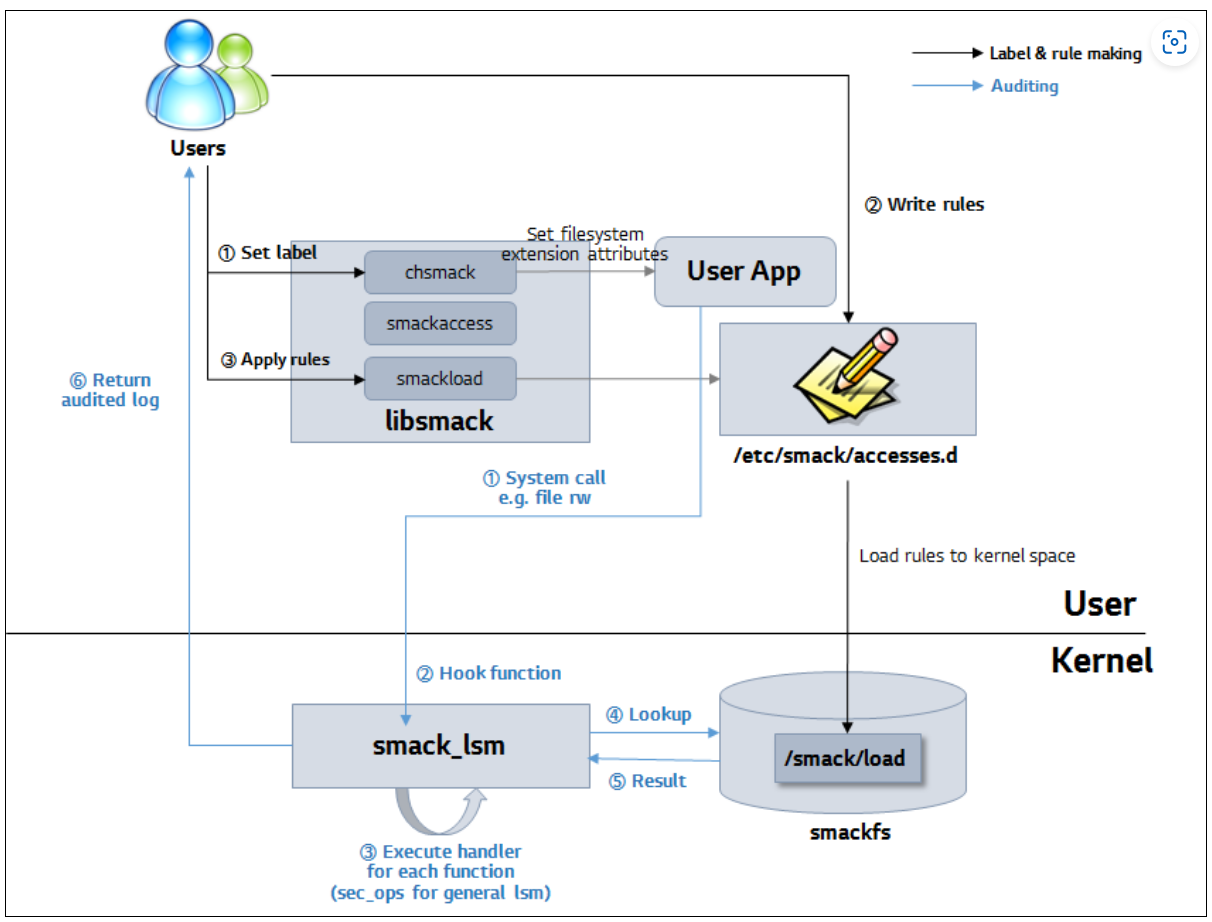


Figure 2 SMACK architect diagram

Here're its key components:

**Key Elements:**

1. ***Users***: These are system users who interact with the system by setting labels and rules for file access. Their interactions with the system are recorded in the audited log.
2. ***User App***: The application used by the users to define labels and rules, interacting with SMACK via libsmack.
3. ***libsmack***:
   * A user-space library that provides APIs to interact with SMACK features such as:
     + chsmack: Used to set filesystem security labels on objects (e.g., files).
     + smackaccess: To check access between labels.
     + smackload: To load access rules into the system.
   * The library communicates with the kernel module and controls access by applying the specified rules.
4. ***/etc/smack/accesses.d***: A directory that contains the access rules, which are defined by the user and written by the User App. These rules are loaded into the kernel space for enforcement by the SMACK framework.
5. ***User Kernel (smackfs):***
   * ***smackfs***: This filesystem stores SMACK rules, labels, and related information in the kernel. The rules from the user-space configuration (/etc/smack/accesses.d) are loaded into this space.
   * ***/smack/load***: This is the point where rules are loaded into the kernel's SMACK subsystem for enforcement.
6. ***smack\_lsm (Linux Security Module):***
   * This kernel module enforces SMACK security policies by hooking into system calls like file read/write (e.g., open, exec, etc.).
   * When a system call is made, smack\_lsm checks the access control rules loaded from smackfs to determine whether the operation is allowed based on the security labels.
   * The handler executes appropriate actions based on the security operations (sec\_ops) defined for LSM.
   * The result of the access control check (allowed or denied) is then returned.
7. ***Auditing***: The diagram shows auditing as part of the feedback loop where a log of the access control decision (success or failure) is returned to the users for review.

**Process Flow:**

1. ***Set Label***: The user sets labels on objects (like files) using chsmack.
2. ***Write Rules***: The user defines access rules in /etc/smack/accesses.d.
3. ***Apply Rules***: The rules are loaded into the kernel using smackload.
4. ***Hook Function***: The smack\_lsm module hooks into system calls (e.g., file read/write) to enforce the access rules.
5. ***Result***: The SMACK system looks up the rules and returns the result of the access check (allow/deny).
6. ***Audit Log***: The decision (whether access was allowed or denied) is logged and returned to the user.

### 2.3.1. Access Rules

The Smack system defines a small set of labels that are used for specific purposes and that have predefined access rules. The rules are applied in this order:

* \* or *star*. The star label is given to a limited set of objects that require universal access but do not provide for information sharing, such as */dev/null*. A process with the star label is denied access to all objects including those with the star label. A process with any other label is allowed access to an object with the star label.
* **\_** or *floor*. The floor label is the default label for system processes and system files. Processes with any label have read access to objects with the floor label.
* **ˆ** or *hat*. The hat label is given to processes that need to read any object on the system.

Processes with the hat label are allowed read access to all objects on the system.

* *matching labels:* A process has access to an object if the labels match.
* *unmatched labels:* If there is an explicit access defined for that combination of process and object labels and it includes the access requested, access is permitted. If there is an explicit access defined for that combination of process and object labels and it does not include the access requested or there is no explicit definition, the access is denied.

I'll take a closer look at labeling in sections 1, 2, and especially section 3, to see what mistakes to avoid and some of the traceoffs of doing so.

### 2.3.2. Defining Access Rules

A Smack access rule consists of a subject label, an object label, and the access mode desired. This triple is written to /smack/load, which installs the rule in the kernel.

When the device is booted and the smack service is executed, it will load some set of rules stored in the /etc/smack/access.d/ directory into the kernel.

I will introduce an example of a case where the set of smack rules is not protected and hackers can take advantage of this to leak information or even worse, escalate privileges. Point to pages

### 2.3.3. SMACK Capabilities

Processes can be given a set of capabilities and its the position of these capabilities that allow certain actions to be taken (i.e.; changing labels on files and processes or changing rules).

To modify labels and rules you must have the correct capabilities, by default in the Linux system several processes are run by the root user and automatically have this capability.

In standard operations, all processes run under the root user are given access to all SMACK labels and have all capabilities enabled. That is any process under root can do pretty much anything it wants with no trouble.

This can be a large security hole. If a process can escalate to running something under the root user, then that potentially malicious code can do anything it wants to the device.

To resolve the issue we've enabled SMACK onlycap. SMACK onlycap allows the system administrator to designate a single label with heightened privilege. Once this is in effect only the properly labeled processes may change labels and rules and all of the other SMACK rules are always in effect regardless of the user the process is running under.

We will analyze what can happen if onlycap is not enabled, or enabled too slowly in section 2.3.4.3.

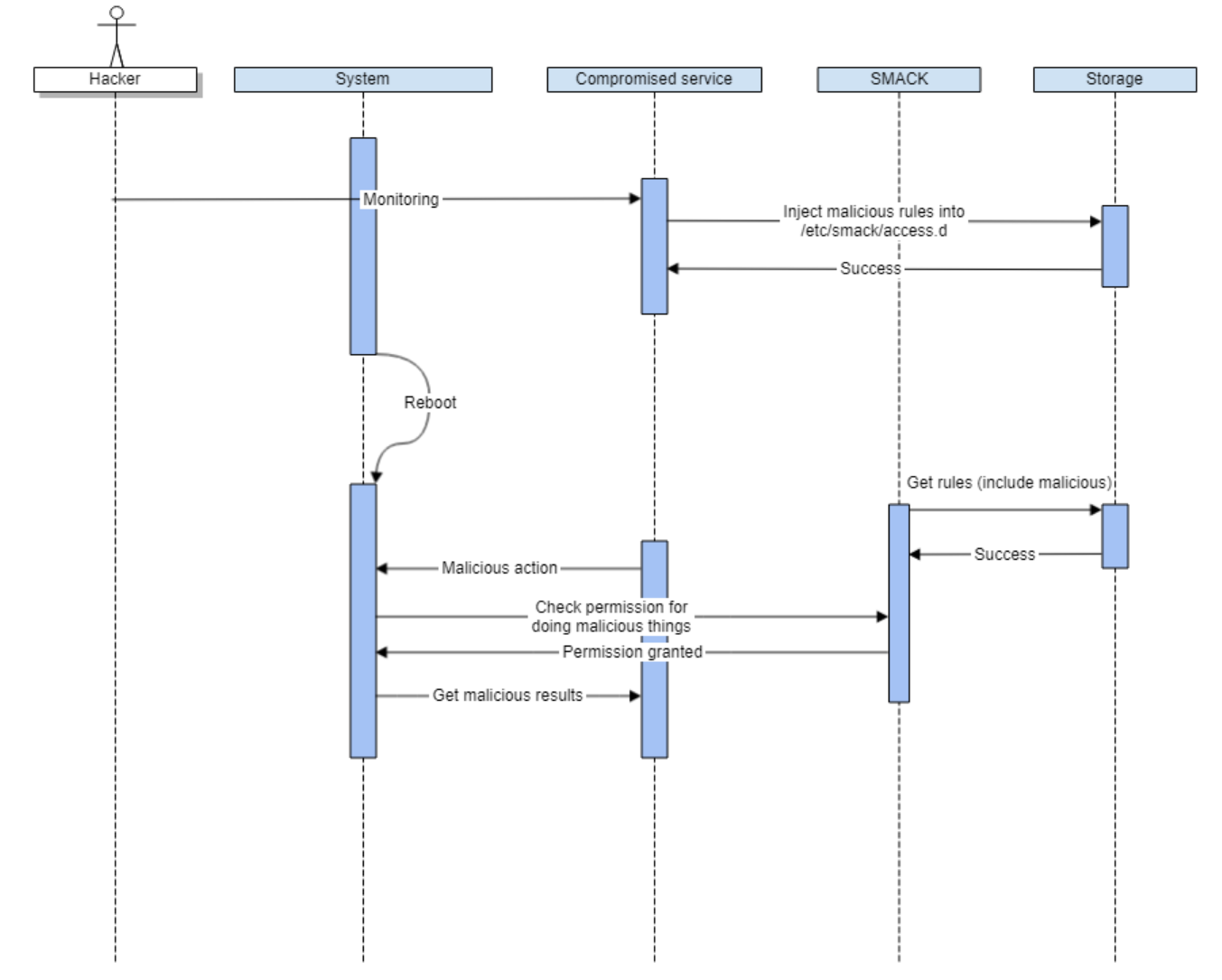
### 2.3.4. Security principles that SMACK developer need to pay attention

#### 2.3.4.1. Protect SMACK configuration files

There are two security properties that need to be kept in mind when implementing rules for smack:

* **Confidentiality**: If a normal labeled subject can read the rules contained in the smack, the attacker may have less difficulty finding erroneous configurations from the administrator.
* **Integrity**: In case a normal labeled subject can overwrite these configuration files, it can grant additional privileges to the label it is carrying (privilege escalation) or remove the privileges of the label causing the system to shut down

In ULC 5.0, developer setup smack but didn’t have right labeled on those rules. So floor can modify those rules, it can’t apply the rule immediately but when the device reset it will make new rules be applied.



The diagram is a sequence diagram that shows a security vulnerability exploit involving SMACK (Simplified Mandatory Access Control Kernel). Here's a step-by-step breakdown:

1. **Hacker** initiates the process, monitoring the compromised service.
2. C**ompromised service** injects malicious rules into the SMACK configuration directory *(/etc/smack/access.d)*.
3. SMACK successfully incorporates these malicious rules.
4. Upon reboot, the system fetches and applies these rules, including the malicious ones.
5. The compromised service checks permissions for performing malicious actions.
6. Permissions are granted due to the injected rules.
7. The hacker performs malicious actions and retrieves results.

Overall, the diagram illustrates how a hacker can exploit unprotected SMACK rules to gain unauthorized permissions and execute malicious tasks.

Affected Project: ULC5.0 (Fixed), JLR VCM (Fixed), JLR TCUA (Fixed)

**Conclusion**: Protecting the sets of smack rules is especially necessary, although /etc/ usually be set to read-only and has integrity protected through DM-verity, we still need to make those rule sets can’t be read by unprivileged access, to make it difficult for hacker to leak and analyze our smack rules.

#### 2.3.4.2. Pay attention on files need to be protected.

It’s hard for developers to find all the files that need to be protected in the system and then set all the label for them properly.

Below is a bug when developer for get to set a proper label for log files in system

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

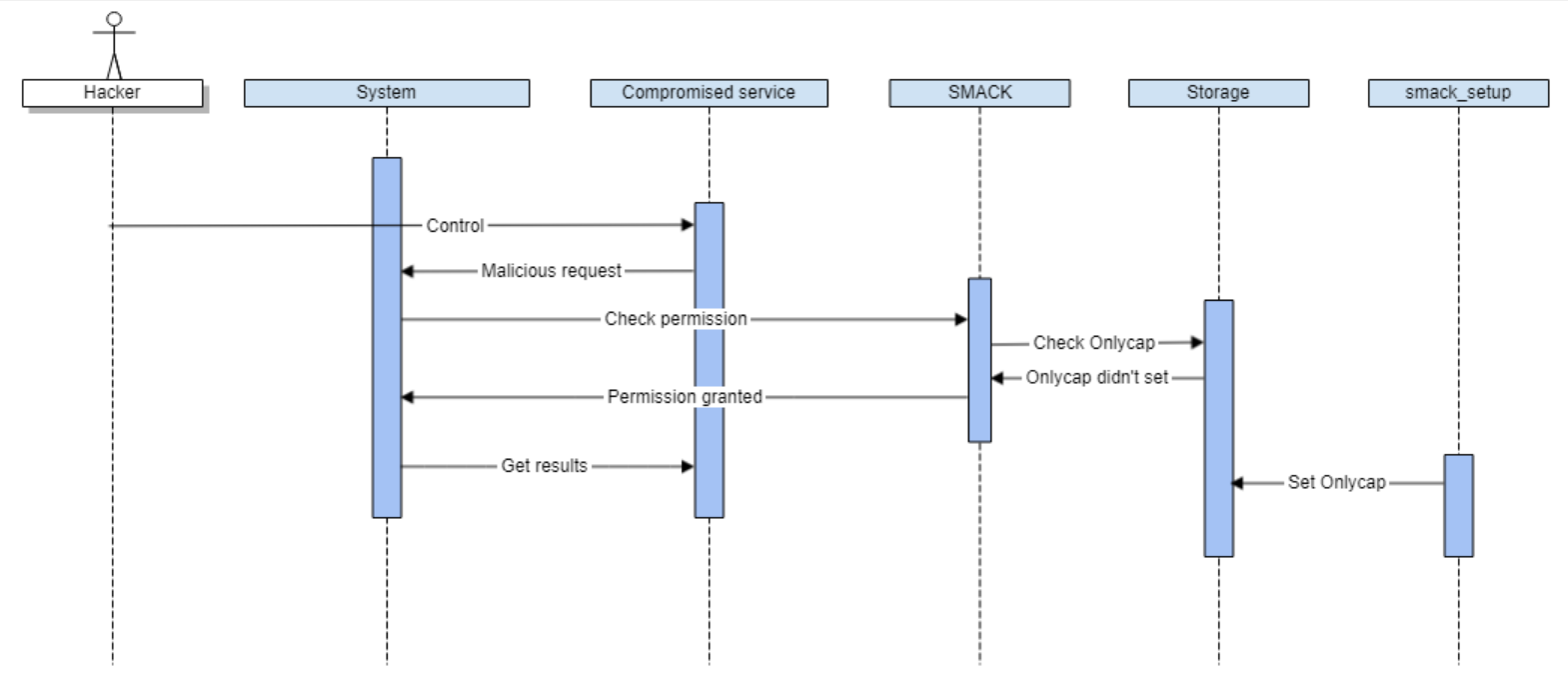
**Affected Project:** JLR VCM, JLR TCUA

**Conclusion:** We used SMACK in many projects but this kind of bug still exist, so developer need to pay more attention to list kind of files need to be protected.

We will come to the solution for this issues with AppArmor section when each program’s developer needs to know which file the program needs to access and then setup a proper profile.

### 2.3.4.3. Make MAC service ready on time

The SMACK setup process requires the system to run a shell script to label the files and then switch the system to enforce mode. If the shell script starts late and runs after compromised processes, those processes can label themselves Privileged and take control of the system.



The diagram is a sequence diagram illustrating a vulnerability in SMACK (Simplified Mandatory Access Control Kernel) related to the ONLYCAP setting not being set immediately after boot. Here's a breakdown:

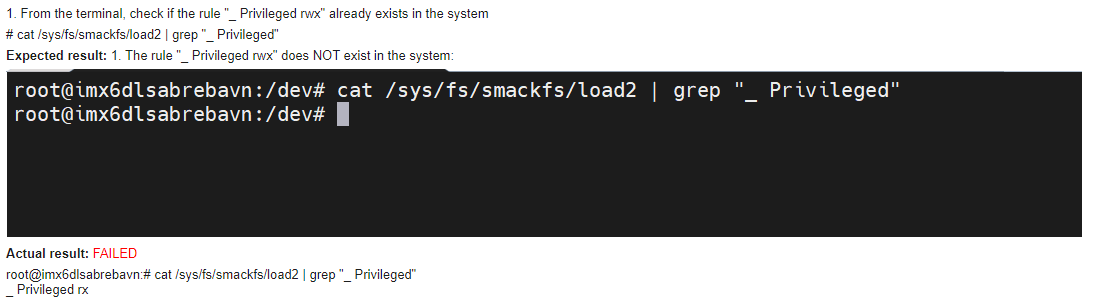
1. A hacker controls the compromised system.
2. The system receives a malicious request.
3. The compromised service checks for permissions.
4. SMACK checks the ONLYCAP setting.
5. The ONLYCAP is found to be unset initially.
6. Permissions are granted due to the unset ONLYCAP.
7. The hacker executes malicious actions and retrieves results.
8. Eventually, the *smack\_setup* process sets the ONLYCAP.

This diagram illustrates a window of vulnerability where permissions are improperly granted because the ONLYCAP is not set immediately, allowing for potential exploitation.

**Affected Project:** JLR VCM

### 2.3.4.4. Don’t give another label access Privileged label

This bug allows floor to access privileges, which is very dangerous if hackers can control floor labels and use them to climb to privileges, we will analyze it more carefully in the example in section 4.3.4.1.

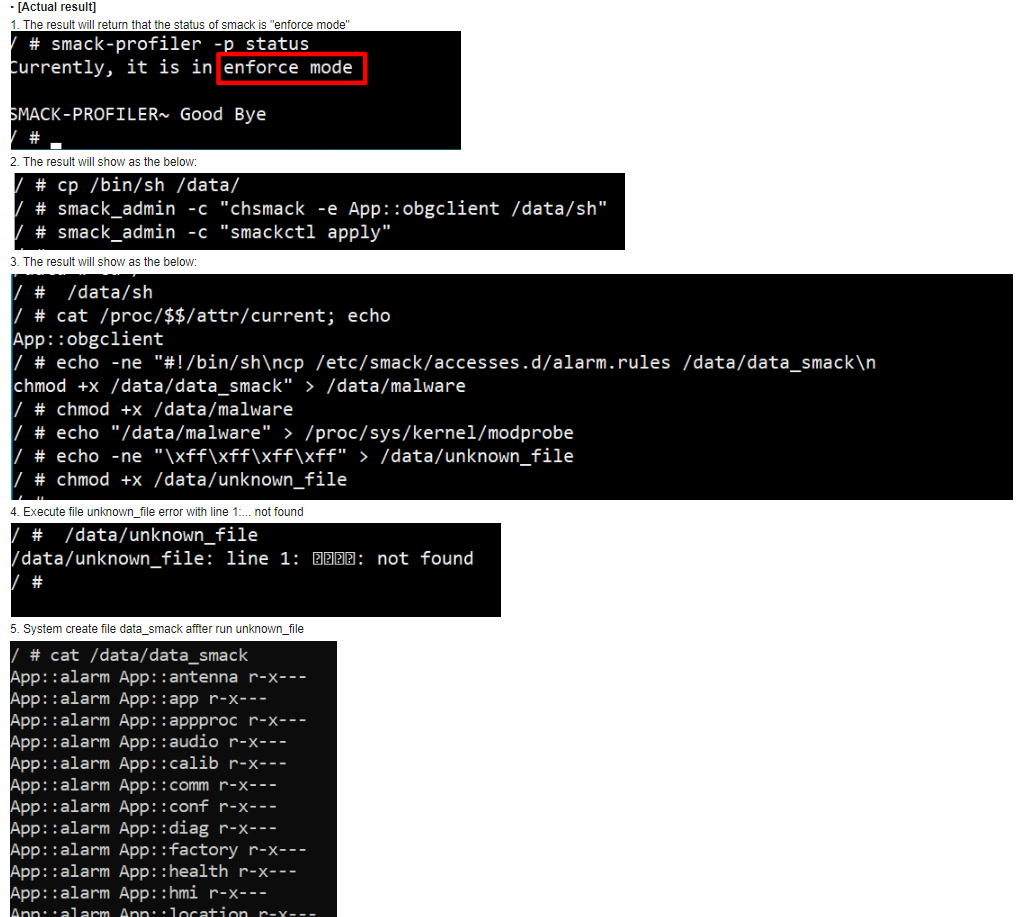


**Affected Project:** ULC5.0, JLR VCM, JLR TCUA

#### 2.3.4.5. Don’t give floor too much privilege

Let's go to the next example to see a very dangerous implementation but still applied by the developer on the project and find out the reason behind why the developer implemented it like that.

I assume that the developer believes that the floor permission cannot be violated so that the floor attacker can access other labels. I will now prove that the developer's assumption is completely wrong.



**Affected Project**: ULC5.0, JLR VCM, JLR TCUA

This is just one of the ways I could find, the idea behind it is to borrow a process with floor permissions to trigger malware created with the process (labeled A), so that the malware will have floor permissions instead of the permissions of label A. I suppose there are other ways.

For example: If an OEM process (usually labeled floor because in many LGE projects they won't label OEM processes) is compromised, it will have access to the LGE label which is supposed to be "SMACK protected"

### 2.3.5. SMACK Conclusion

Through the above analysis, I would like to list some problems that may arise when implementing SMACK.

* Difficulties of systemd: systemd has a label of floor but accessing labeled files of applications during boot leads to 2 problems I already mention in 2.3.4.3, 2.3.4.5.
* Update may result in some files with labels no longer existing in the rule set and when the system accesses these files it will cause a system crash.

The above SMACK issues can be solved with AppArmor, however AppArmor can also have vulnerabilities if not implemented properly.

## 2.4. Introduction to AppArmor

AppArmor ("Application Armor") is a Linux kernel security module that allows the system administrator to restrict programs' capabilities with per-program profiles. Profiles can allow capabilities like network access, raw socket access, and the permission to read, write, or execute files on matching paths. AppArmor supplements the traditional Unix discretionary access control (DAC) model by providing mandatory access control (MAC). It has been partially included in the mainline Linux kernel since version 2.6.36 and its development has been supported by Canonical since 2009.

## 2.4.1 AppArmor Diagram

The *Figure 3* below presents a high-level overview of the AppArmor security framework, which is a Linux Security Module (LSM) that confines programs to a limited set of resources based on security profiles.

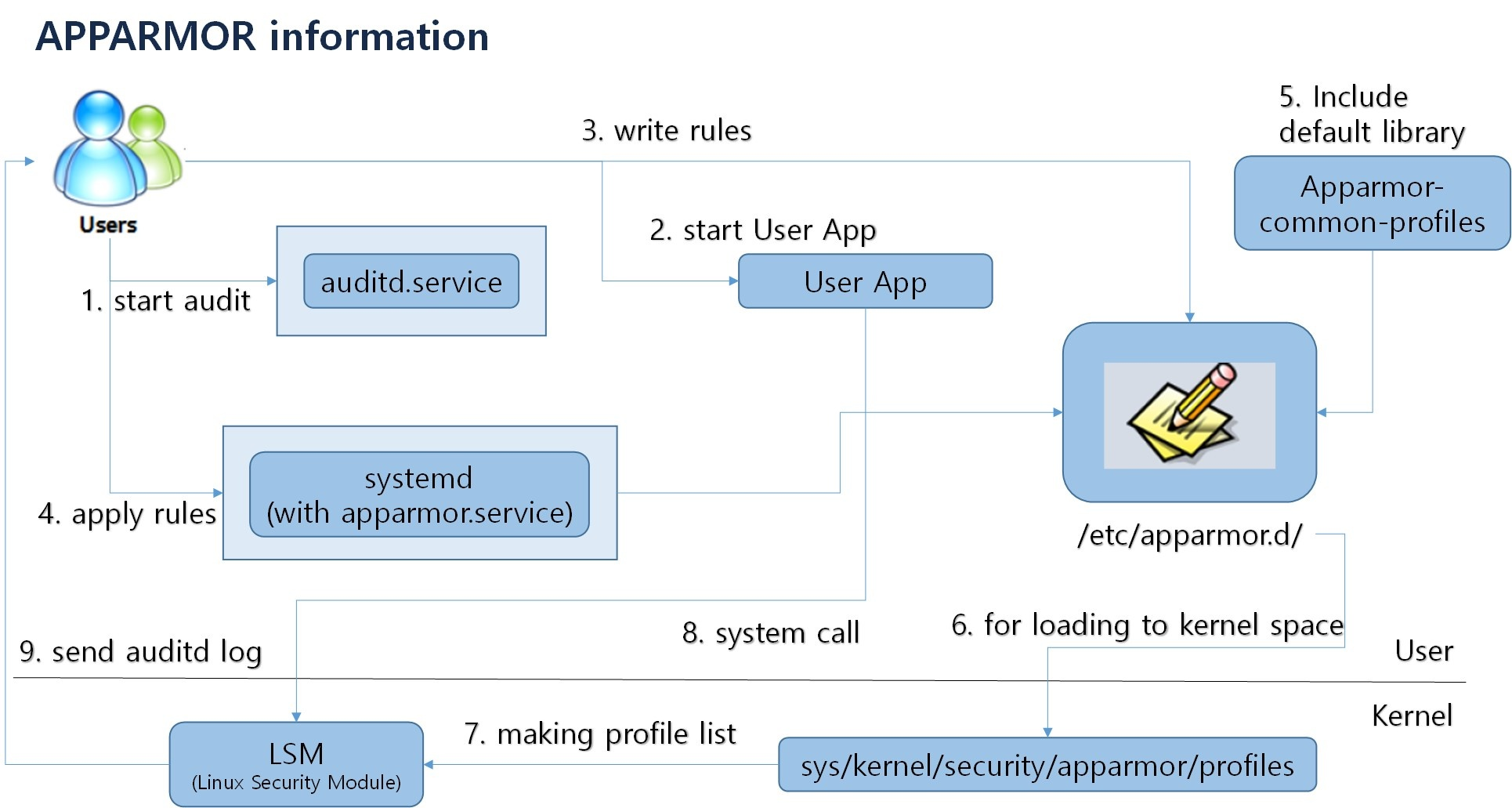


Figure 3 AppArmor Architect Diagram

Here's a detailed breakdown of the components and processes in the diagram:

**Key Components:**

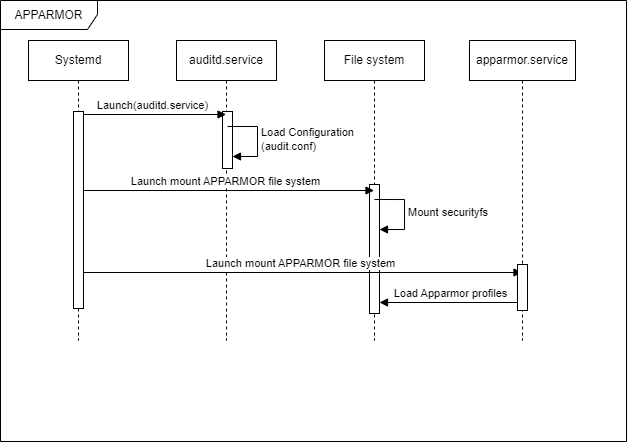
1. ***Users***: These are system users who initiate the audit process, write security rules, and interact with the system to control application behavior based on the AppArmor profiles.
2. ***auditd.service***: A system service responsible for managing the audit log. It captures and records security-related events initiated by the users or triggered by AppArmor.
3. ***User App***: An application started by the users to configure and manage AppArmor profiles. This is where users define or modify security rules.
4. ***AppArmor Profiles***: The profiles are located in /etc/apparmor.d/ and define the permissions for each application (e.g., what files or system calls they can access). ***Apparmor-common-profiles*** is a default library that contains pre-defined profiles for common applications, which can be included by users in their configurations.
5. ***systemd (with apparmor.service)***: systemd is the init system that controls services on Linux systems. The apparmor.service is a part of systemd and ensures that AppArmor is properly loaded and applied to enforce the security profiles.
6. ***/etc/apparmor.d/***: This is the directory where user-defined AppArmor profiles reside. These profiles are written by users and stored in the AppArmor configuration directory, which are later loaded into kernel space.
7. ***Linux Security Module***: This module enforces AppArmor security profiles. It is responsible for controlling access to system resources based on the profiles provided by users. The profiles are loaded and used by the kernel to ensure applications are constrained by the defined policies.
8. ***/sys/kernel/security/apparmor/profiles***: The profiles in /etc/apparmor.d/ are loaded into this directory in kernel space for enforcement by the AppArmor LSM. Once profiles are loaded into the kernel, they restrict access for programs based on their corresponding rules.

**Process Flow:**

1. ***Start Audit***: Users initiate the audit process using the auditd.service to monitor and log events.
2. ***Start User App***: Users start an application to configure and manage security rules for AppArmor.
3. ***Write Rules***: The users write security rules (profiles) and store them in the /etc/apparmor.d/ directory.
4. ***Apply Rules***: The systemd service, along with apparmor.service, applies the security profiles to the system.
5. ***Include Default Library***: Users can also include pre-defined profiles from the Apparmor-common-profiles library.
6. ***Load Profiles to Kernel Space***: Profiles from /etc/apparmor.d/ are loaded into kernel space for enforcement by the AppArmor module.
7. ***Making Profile List***: The Linux Security Module (LSM) compiles the list of active profiles that define access permissions for applications.
8. ***System Call***: When an application makes a system call, AppArmor checks the profiles in the kernel to ensure the call is allowed based on the defined rules.
9. ***Send Audit Log***: The result of the enforcement (allowed or denied access) is logged and returned to the auditd service, providing feedback to the user.

## 2.4.2. Sequence Diagram

The diagram is a sequence diagram illustrating the initialization process of AppArmor using systemd.



Here's a breakdown of the steps:

1. *Systemd* starts the *auditd.service*.
2. *auditd.service* loads its configuration file (*audit.conf*).
3. The AppArmor filesystem is mounted.
4. The sequence proceeds to mount *securityfs* on the filesystem.
5. Finally, *apparmor.service* loads the AppArmor profiles.

# 3. Asset Identification

In terms of software security, assets can include software, hardware, and even information that is considered important to protect. In this project, damage scenarios were created based on security characteristics (confidentiality, integrity, availability, authentication, authorization, and non-repudiation) to identify assets relevant to security goals.

## 3.1. Asset Identification Process

In this project, the System Context Diagram and Sequence Diagram described in previous sections were utilized to understand and identify assets worth protecting within the system.

## 3.2. Asset List and Damage Scenarios

Damage scenarios in which the security characteristics of the security characteristics mentioned above are violated are created, and the shaded items are External Entities and are not assets, but were included as reference items to analyze the data flow associated with the security assets.

# 4. Security Threat Analysis

In order to recognize threats to the system, we expressed the system using DFD and performed modeling on threats that could infringe six security properties using STRIDE.

## 4.1. System Structure Representation (DFD)

Based on the System Context Diagram and Sequence Diagram described in previous sections, different scenarios were derived and modelled with DFD (Data Flow Diagram). Threat analysis were conducted for three different DFDs corresponding to three different scenarios.

Add DFD Model here (TBD)

## 4.2. Threat Modeling

For threat modeling, Microsoft's STRIDE methodology was applied. The reason STRIDE was selected among other threat models is that it provides necessary tools and is the most familiar tool for threat analysis in LG Vehicle Solution Company as well as automotive industry.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Threat | Desired Property | Description |
| S | Spoofing | Authenticity | Pretending to be something or someone other than yourself |
| T | Tampering | Integrity | Modifying something on disk, network, memory, or elsewhere |
| R | Repudiation | Non-repudiation | Claiming that you didn't do something or were not responsible; can be honest or false |
| I | Information Disclosure | Confidentiality | Someone obtaining information they are not authorized to access |
| D | Denial of services | Availability | Exhausting resources needed to provide service |
| E | Elevation of privilege | Authorization | Allowing someone to do something they are not authorized to do |

# 5. Risk Assessment

# 6. Security Requirement

# 7. Mitigation and Implementation

# 8. Conclusion

# 9. Future Work