A MODEL FOR SAFE AND SECURE EXECUTION OF
DOWNLOADED VEHICLE APPLICATIONS

Phu H. Phung*, Dennis Kengo Nilsson†

*Department of Computer Science and Engineering
Chalmers University of Technology, Sweden
phu.phung@chalmers.se
†Syncron Japan
dennis.nilsson@gmail.com

Keywords: Vehicle software/firmware, language-based security, policy enforcement.

Abstract

Existing secure protocols and code signing mechanisms for vehicle systems to download and install software over the air certify only the origin and the integrity of software; thus, they do not address errors that might not be detected in the development process and cannot ensure that the downloaded software do not contain malicious code. In this paper, we identify such possible threats by developing a threat model for the vehicle software architecture. We propose countermeasures against the threats by preventing or modifying inappropriate behaviour caused by, e.g., malicious or poorly designed applications. We propose a model to deploy the approach which is based on modifying the application at the wireless gateway in the vehicle before being installed. As a result, security policies are embedded into the application and intercepts security relevant execution events. Thus, the execution of downloaded vehicle applications is monitored to ensure the safety and security for the vehicle system and to detect potential cyber attacks.

1 Introduction

Current trends in vehicle systems are extensibility and openness which allow software to be downloaded and installed over the air [1]. However, such extensible systems potentially face difficult security problems. Although several secure protocols or code signing mechanisms [2,3,4] in vehicle systems have been proposed, they certify only the origin and the integrity of software. These solutions cannot ensure that the software do not contain malicious code nor address errors that might not be detected in the development process.

This paper analyses possible threats and potential cyber attacks in the vehicle software architectures including add-on software and ECU (Electronic Control Unit) firmware. For each threat, we introduce a corresponding countermeasure. In order to implement the countermeasures, we study a classic approach of imposing a security policy (defining a countermeasure) on downloaded software via a security reference monitor. The reference monitor [5] is a trusted component which intercepts security relevant resource requests and applies a security policy to decide whether to grant such requests.

We analyse the difference between add-on software and ECU firmware scenarios in vehicles and propose a model to deploy a reference monitor for each scenario. Add-on software and ECU firmware installation and updating over the air are through a wireless gateway in the vehicle. In our proposed model, a transformation module is located at the wireless gateway as a proxy between the downloading module (as a security reference monitor) and the installation/updating module. After downloading, the code is rewritten to “embed” (inline) security policies within it. Thus, the execution of the modified code is enforced by application-specific security policies implementing countermeasures to prevent identified threats. We illustrate the implementation of the reference monitor for add-on software architecture in an aspect-oriented programming language [17] and propose a solution for the implementation for the ECU firmware architecture. The main contributions of this paper are:

• We have analysed threats in the scenarios of vehicle software architectures and have proposed countermeasures against the threats.
• We have proposed a model to deploy the security reference monitor that can implement the proposed countermeasures against threats in the vehicle application scenarios.
• We have illustrated the implementation of the proposed model in the vehicle add-on software architecture and have proposed a solution to implement the model for the ECU firmware architecture.

The next section discusses related research in the area of security in vehicle systems. Section 3 reviews the architecture for application scenarios in the in-vehicle system. Section 4 defines the problem, analyses threats and presents the proposed countermeasures. Section 5 proposes a model (Section 5.1) to implement the proposed countermeasures by applying the reference monitor approach (using code transformation) into the vehicle software architecture. The realisation of code transformation in the model for two vehicle application architectures is presented (Section 5.2).
with several examples illustrating security policies that implement countermeasures. Conclusion and future work directions are presented in Section 6.

2 Related work

The Global System for Telematics (GST) project [1] provides a reference standard for vehicle systems. The standard is Java/OSGi [6] based, describes how a telematics client application can be downloaded and installed over the air from a control centre, and specifies an interface for receiving vehicle data. The OSGi (Open Services Gateway initiative) is a standard framework providing an environment for the modularization of applications into smaller components called bundles. The bundles are developed in Java programming language and run on a Java runtime environment. In practice, the OSGi framework has been used in in-vehicle systems by several car manufacturers. For instance, BMW has used the OSGi specifications as the base technology for its high-end infotainment platform [7]. Open Arena Lab Demonstrator Environment (OALDE) is a project, collaborated by Volvo, Ericsson and Lindholmen Science Park [21], that aims to build a solution for information exchange and interfaces between driver, vehicle, transport management, society and infrastructure. The OSGi framework is also used in OALDE for the vehicle software architecture.

The security sub-project of GST [2] considers the security aspects of the GST project and intends to provide a large-scale infrastructure for telematics applications to ensure security, privacy, and reliability requirements. The presented solutions focus in general on communications security and verifying the integrity of downloaded software in vehicles; however, safety and security of the software for the system has not been considered.

The S3MS project [8] has developed a security-by-contract technological solution which would allow users to download and use applications on a range of devices. A security contract can be defined and checked by a customer’s service provider, a trusted third party, or even by the customer. The security contract determines the restrictions on data and resource usage of the customer device for the downloaded application. This requires an agreement between users and developers.

An emerging trend among vehicle manufacturers is to perform wireless ECU firmware updates [9]. Previous research [3,4] has focused on securing the communication channel in terms of data confidentiality, data integrity, and data authentication. However, if the firmware contains a bug and an attacker exploits it, various attacks on the in-vehicle network are possible [10,11,12]. Consequently, there is a need for fine-grained security mechanisms such as security policy enforcement to monitor application execution.

3 Vehicle software architecture

In this section, we describe details on the architecture for the software download scenario in vehicle that we use for this work. A modern vehicle contains an in-vehicle network consisting of various ECUs and networks such as CAN (controller area network), MOST (media-oriented system transport), and LIN (local interconnect network) that are interconnected through gateways. There is a wireless gateway connected to the CAN bus that allows the in-vehicle network to access external networks such as fleet-management systems and the Internet. The in-vehicle network is illustrated in Fig. 1. The vehicle contains various applications which can be divided into two categories: add-on software and ECU firmware. The add-on software requires an underlying platform that runs the application. On the other hand, for ECU firmware, the application is self-contained, flashed to a flashable ROM, and runs on a microprocessor [13].

Add-On Software

In this work, we consider the vehicle software architecture similar to the architecture in OALDE project [21], which is based on the open standard in the GST [1]. In this architecture, software is installed and executed on the OSGi platform [6], which runs on an operating system (OS) on an on-board vehicle computer. An application in OSGi consists of one or more bundles. The on-board vehicle computer is connected to the CAN bus and provides interfaces for applications such that the applications could access the underlying car infrastructure. The add-on software is downloaded over the Internet via the wireless gateway and is then installed and run within the OSGi framework. The add-on software is often used to provide infotainment services. Software that does not interfere with vehicle controls could be provided by third party providers. As described in [1], the GST standard will provide an environment for an open market for online services establishing seamless internetworking. Moreover, there will be standard interfaces that allow different parties to develop and deploy the new functionality or sub-systems. Such an application example, which could be supplied by a third party, is the location-based service for vehicle that provides information such as restaurants in the vicinity. No connection to vehicle control and maneuverability ECUs should be allowed.

ECU Firmware

Specific firmware is flashed to the flashable ROM of each ECU. The firmware provides various functionalities for the
corresponding ECU. The firmware is typically issued by the vehicle manufacturer because the functionality of the specific ECUs often require access to vehicle control software. For example, all driver assistance applications that communicate with ECUs responsible for the control and maneuverability of the vehicle are built-in firmware.

4 Threat analysis

In this section, we describe the assumptions we make about an attacker in the vehicle software architecture. We analyse various threats based on the assumptions and design countermeasures to address the threats. The countermeasures are based on the monitoring of vehicle software execution to enable safe and secure execution.

4.1 Attacker assumptions

As presented in Section 3, the add-on software is downloaded over the Internet via the wireless gateway. We assume that the downloading process is guaranteed to provide authenticity and integrity via a secure protocol, e.g., [3,14,4]. Thus, the downloaded software is assumed to not be compromised by attackers. In this architecture, an attacker could be a malicious third party software developer that could easily create malicious software. The malicious code could target other ECUs connected to the CAN bus. In addition, developers normally focus on the functionality of software which could lead to an oversight of security aspects such that trusted software might contain threats. For ECU firmware, we assume that some firmware could be provided by third party developers. The malicious code installed in an ECU could send messages on the connected CAN bus and potentially affect other components in the in-vehicle network. We separate the attacker actions for the two different categories.

Add-On Software

The attacker creates add-on software to, e.g., the infotainment system. The actions that the malicious code can take are limited to the functions that the add-on software is allowed to perform. Moreover, the interface between the add-on software and the OS limits the attack actions. The malicious code could try to read data from the OS that it normally should not access or spoof messages that it typically should not send.

ECU Firmware

The attacker creates a firmware that can perform arbitrary actions for the corresponding ECU. The firmware is flashed to the target ECU. For example, the ECU could be controlled to read messages that it normally should not have access to or spoof messages that it typically should not send. Since the ECU firmware allows communication with vehicle control ECUs, an attacker can target the control and maneuverability of the vehicle. Examples of vehicle virus are found in [10].

4.2 Threats and Countermeasures Design

This section describes possible threats of software and firmware in vehicle systems. We identify the threats for the two architectures and consider harmful behaviours that might not be prevented by existing download/installation protocols and might not be controlled in the development process. We list below possible threats together with a corresponding countermeasure to enable safe and secure software execution.

- An unauthorised read event caused by an application should be suppressed. If the add-on software or the ECU firmware tries to read messages in the OS or on the CAN bus that are not meant for the running application, those reads should be detected and suppressed. Therefore, it is important to establish policies to determine which read events should be allowed for the particular application.

- An unauthorised send event caused by an application should be suppressed. If the add-on software or the ECU firmware tries to send spoofed messages or replay messages in the OS or on the CAN bus, those events should be detected and suppressed. In addition, for the ECU firmware, those events should be logged since they could potentially target safety-critical ECUs. Policies that define which messages are allowed to be sent for specific applications must be carefully designed and developed.

- An unauthorised use of a critical resource by an application should be logged and suppressed, and an alert should be generated. If the ECU firmware or the add-on software tries to access and use a critical resource, such as vehicle control or engine management that it should not use, those events should be detected, logged, and suppressed. In addition, an alert about the event should be generated and sent to the proper authorities. The alert should contain which application caused the event, which resources the application tried to access, the nature of the access, and the time of the event. The alert could, e.g., be sent to the vehicle manufacturer, who then could investigate the event further.

- Authorised use of system and critical resources by an application should be limited. ECU firmware and the add-on software should be prevented from overusing system and critical resources. Applications could potentially abuse resources, and thus policies to prevent such abuses must be defined and enforced. Events requesting resources that exceed the limit should be suppressed.

- The execution of add-on software should be monitored by a global policy handler in the software framework. The framework should contain a global policy handler responsible for monitoring the execution of software add-on applications. The policy handler should aggregate the behaviour for all monitored applications and apply policies on a global view of the applications.

- Read and send events in ECU firmware are recorded together with the extracted content of messages. From the trace of extracted information in each ECU, potential cyber attacks can be detected (c.f. [15]).
The basic approach to implementing the proposed countermeasures in this work is to intercept security relevant events of vehicle applications before they are installed or updated. These events can then be permitted to be run, rejected, modified, or recorded at the execution stage according to some predefined policies. This is a variation on the classic idea of a security reference monitor [5], and in what follows we will refer to this mechanism as the “reference monitor”.

The implementation of this idea will be through a transformation of the code before it is installed or updated to the vehicle. Program transformation to embed security policies is an implementation of the reference monitor approach in which a target program is modified so that it adheres security policies on security-relevant actions or events when it executes. First we must decide on what is being intercepted and monitored. From the threat model, we derive that it is system calls to read/send data from/to the in-vehicle network, and to use critical resources that are the security-relevant events. The countermeasures are defined in terms of policies and inlined with these events. We describe the conceptual model to deploy the method in the following section.

5.1 The Transformation Model

Important questions here are how and when the security policy defining the corresponding countermeasure will be embedded into the code. This subsection presents a common model for the two different vehicle software architectures. As mentioned, vehicle applications are downloaded through the vehicle wireless gateway and guaranteed authenticity and integrity by a secure download protocol. In our model, an additional transformation module is placed at the gateway as a middle step of a software installation or updating, i.e., after the software is downloaded and before it is installed/updated to the particular platform. The transformation module receives the downloaded software and transforms it to a new version that adheres countermeasures in terms of security policies. We assume that the security policies are defined at the vehicle manufacturer and included in the transformation module. Thus, in this model, the policy-inlined firmware and software woven with a set of countermeasures could prevent various attacks. The model is illustrated in Fig. 2.

5.2 Realisation

Since there are two categories of vehicle software, the transformation is implemented differently. We present the implementation possibilities for each architecture as follows.

```java
//here to import OSGi libraries
import Global.PolicyHandler;
public aspect ApplicationStart{
  pointcut startApp(BundleContext context):
    execution (*BundleActivator.start(BundleContext) ) && args (context));
  before (BundleContext context): startApp(context) {
    PolicyHandler.monitor(context);
  }
}
```

Listing 1. Put a starting application under the control of the policy handler.

<table>
<thead>
<tr>
<th>Threats</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application reads unauthorised messages to the OS</td>
<td>Suppress the event</td>
</tr>
<tr>
<td>Application sends unauthorised messages to the OS</td>
<td>Suppress the event</td>
</tr>
<tr>
<td>Application accesses unauthorised critical system components</td>
<td>Generate alert, log and suppress the event</td>
</tr>
<tr>
<td>Application has high priority privilege that cannot be terminated</td>
<td>Put it under the control of a global handler</td>
</tr>
<tr>
<td>Application consumes considerable amount of resources, e.g., sending SMS or using high amount of memory (might be due to programming failure)</td>
<td>Suppress the event if amount of allowed resource usage for an individual application is exceeded</td>
</tr>
<tr>
<td>A set of applications consume resources within their limits but their total resource usage is high</td>
<td>Policy handler aggregates resource usage for all applications and the event is suppressed if amount of total resource usage is exceeded</td>
</tr>
</tbody>
</table>

Table 1: Threat model for add-on software.

<table>
<thead>
<tr>
<th>Threats</th>
<th>Countermeasures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application reads unauthorised messages on the CAN bus</td>
<td>Suppress the event</td>
</tr>
<tr>
<td>Application sends unauthorised messages on the CAN bus</td>
<td>Log and suppress the event</td>
</tr>
<tr>
<td>Application accesses unauthorised critical system components</td>
<td>Generate alert, log and suppress the event</td>
</tr>
<tr>
<td>Application consumes considerable amount of resources, e.g., using high amount of memory (might be due to programming failure)</td>
<td>Suppress the event if amount of allowed resource usage for an individual application is exceeded</td>
</tr>
</tbody>
</table>

Table 2: Threat model for ECU firmware.

**Threat Model** As a result, Table 1 summaries the threats posed by add-on software, and Table 2 presents the threats posed by ECU firmware. A corresponding countermeasure for each threat is also described in the tables.
a) Transformation for Add-on Software: As shown in our previous work [16], various sorts of security policies can be defined and enforced in an open system like the OSGi framework using an aspect-oriented language [17]. In this work, we use a similar method to implement the proposed countermeasures in terms of policy enforcement. Since applications in vehicle software architecture are developed in Java and distributed in Java bytecode, an appropriate tool to transform such distribution is AspectJ language. AspectJ is an “industrial strength” aspect-oriented programming (AOP) language [17] that can provide programmatic means to modularise crosscutting functionalities of complex software systems so that program concerns in a software system can be captured and encapsulated into so-called aspects. AspectJ is a language that extends the Java programming language and implements the paradigm of AOP. The transformation module for add-on software uses the AspectJ weaver tool to modify the downloaded software and embed aspects defined in AspectJ language to implement the proposed countermeasures. Listing 1 gives a transformation example that a starting application in OSGi is monitored and controlled by a global policy handler (class PolicyHandler) as designed in Section 4.1. Listing 2 illustrates the definition of a policy implementing a countermeasure against unauthorised read events as shown in the countermeasure table (Table 1). If the application reads an unauthorised message from the OS, the action will be suppressed. MessageControl in Listing 2 is a part of the global policy handler installed in advance in the OSGi framework. The handler is a part of the global library in the framework responsible for monitoring the execution of add-on software. For space reasons, we omit illustrations implementing the handler and other countermeasures proposed in Table 1. Practical issues such as how to install a library is described in [16]. In conclusion, AspectJ could provide a complete and robust tool to modify add-on software for vehicle systems such that the software adhere appropriate policies to prevent potential threats. In this work, we use the OSGi framework as a platform for add-on software that supports downloading of software from third parties. The policies we have illustrated are defined in AspectJ and can be woven into the Java bytecode, which is provided by a third-party provider, by the transformation module in the wireless gateway. The modified software can then be installed to a vehicle and executed properly. Ideally, the vehicle manufacturers should define relevant policies since they are the only ones who know the intended functionality of the application. In that sense, the nature and the functionality of the application should be analysed by the vehicle manufacturer in order to define meaningful policies. For example, third party applications should not be allowed to interact with safety critical in-vehicle components, as studied in our previous work [18].

b) Transformation for ECU Firmware: We discuss a similar model and propose a solution for the ECU firmware scenario. The major difference is that the add-on software is run in a middleware, e.g., the OSGi framework and an underlying OS that can handle policy enforcement, and the ECU firmware is run directly on ECU itself. There exists no inherent support for transformation for ECU firmware.

First, a platform like OSGi for ECU firmware should be provided such that the firmware is run on some form of middleware running on an OS. Future directions for ECU development might include more powerful hardware that run some form of lightweight OS and combine several ECU functionality to one single software. In this future scenario, the ECU firmware could be inclined with security policies.

In addition, ECU firmware is typically developed in nesC or C, and thus aspect-oriented solutions for these programming languages could be used in order to apply our transformation method to implement the countermeasures. Recent research projects, such as Aspect nesC [19], ACC [20], have provided aspect-oriented solutions for nesC and C; however, they do not focus on security policy enforcement. Providing a security policy enforcement mechanism for such languages using an aspect-oriented approach is a sensible solution for applying our model.

Moreover, the cost of using several ECU hardware could be lowered if the ECUs are replaced with a more powerful hardware providing a combined software solution. As seen in other areas, hardware solutions have incrementally been replaced with more advanced software solutions, and the vehicle software systems might follow the same path.

6 Conclusion and future work

We have presented a model to deploy a reference monitor approach for vehicle applications for monitoring their execution to enable safety and security. We distinguish between add-on software and ECU firmware and develop a threat model for each category. For each threat we describe a corresponding countermeasure that can be implemented as a security policy. We show a few examples of how to define such policies in AspectJ.

Add-on software and ECU firmware are treated differently due to the nature of their application. Add-on software is run on a software framework on an underlying operating system and is not allowed to interact with safety-critical vehicle
functionality. We have shown that defining security policies for such applications is a promising solution for preventing potential threats. On the other hand, ECU firmware runs on a microprocessor, and frameworks such as OSGi for ECU firmware are not readily available. We have given a solution for implementing security policies for ECU firmware and shown the benefits of using such policies. As vehicle manufacturers move towards solutions for firmware updates and software downloads over the air, not only protocols for protecting the wireless communication channel must be designed but also solutions for safe and secure software execution in the vehicle must be developed.

**Future Work** The most pertinent future work is to create a testbed for implementing the security polices for add-on software and ECU firmware. A rigorous analysis on ECU functionality and security threats must be performed to determine what policies to develop. A number of attack use cases must be developed and analysed in order to define concrete policies for various firmware and add-on software.

In the context of security policy enforcement mechanisms, the proposed model that supports program transformation to enforce security policies could be developed in an in-vehicle simulation system such as [21]. In addition, future work should also consider the design of policy management mechanisms dealing with issues such as interactive policy construction and the composition of different security policies.

**References**


