Security of Embedded Software

An Analysis of Embedded Software Vulnerabilities and Related Security Solutions

CLÉMENT GABORIAU--COUANAU
Abstract

The increased use of computer systems for storing private data or doing critical operations leads to some security issues gathered in the area cybersecurity. This neologism leads people to think about the security of information systems and general-purpose computers. However, with the growth of the Internet of Things, embedded systems are also concerned with these issues. The speed of development of this area often leads to a backwardness in the security features.

The thesis investigates the security of embedded systems by focusing on embedded software. After classifying the vulnerabilities which could be encountered in this field, a first part of this work introduces the realisation of a document gathering guidelines related to secure development of embedded software. This realisation is based on an analysis of the literature review, but also on the knowledge of engineers of the company. These guidelines are applied to the project of a client.

The result of their application allows us to prove their consistency and to write a set of recommendations to enhance the security of the project. The thesis presents the implementation of some of them. Particularly, it introduces a way to secure an Inter-Process Communication (IPC) mean: D-Bus, through a proof of concept. The result shows that the security policy of D-Bus is efficient against some attacks. Nevertheless, it also points out that some attacks remain feasible. The solution is implemented on an embedded board to analyse the computational overhead related to this embedded aspect. As expected, a more complex and detailed a policy is, the higher the overhead tends to be. Nevertheless, this computational overhead is proportional to the number of rules of the policy.

Keywords

Embedded Software, Cybersecurity, Guideline, Inter-Process Communication (IPC), D-Bus, Monitoring
Abstract

Den ökade användningen av datorsystem för att lagra privata data eller göra kritiska operationer leder till vissa säkerhetsproblem som samlas i området cybersäkerhet. Denna neologism leder människor att tänka på säkerhetssystemen för informationssystem och allmänt tillgängliga datorer. Men med tillväxten av saker i saken är inbyggda system också berörda av dessa frågor. Utvecklingshastigheten för detta område leder ofta till en underutveckling säkerhetsfunktionerna.

Avhandlingen undersöker säkerheten för inbyggda system genom att fokusera på inbyggd programvara. Efter att ha klassificerat de sårbarheter som kan uppstå i det här fältet introducerar en första del av det här arbetet realisationen av ett dokument av riktlinjer om säker utveckling av inbyggd programvara. Denna insikt bygger på en analys av litteraturgranskningen, men också på kunskap om ingenjörer i företaget. Dessa riktlinjer tillämpas på en kunds projekt.


Nyckelord

Inbyggd Programvara, Cybersäkerhet, Riktlinje, Interprocesskommunikation (IPC), D-Bus, Övervakning
I would like to thank my industrial supervisor, Marc-Antoine MARTIN, and the responsible of the client’s project, Christopher LAMBERT, for their help and guidance during the whole thesis. I am also grateful to the entire team of the project for their support during the realisation of this thesis.

I would like to thank Cyril DEGOUT and his team of engineers for having provided me interesting tasks at the beginning of my thesis. More generally, I would like to thank the company and its staff for their warm welcoming and their good mood during my internship.

I am grateful to Thomas GENET who gave me some of his time to improve the guidelines we wrote through this work.

I would like to thank my examiner Pr. Elena DUBROVA, and my academic supervisor, Dr. Sha TAO, for their feedback to improve the quality of my work.

Finally, I would like to thank my parents and my brother for their support during the most difficult parts of my internship.
# Table of Contents

1 Introduction ........................................................................................................ 7  
   1.1 Background .................................................................................................. 7  
      1.1.1 Thales Services SAS .............................................................................. 7  
      1.1.2 Internship Department ......................................................................... 8  
   1.2 Problem ........................................................................................................ 8  
   1.3 Purpose ......................................................................................................... 9  
   1.4 Goal ............................................................................................................. 9  
      1.4.1 Deliveries .............................................................................................. 9  
      1.4.2 Benefits ............................................................................................... 9  
      1.4.3 Ethics and Sustainability ....................................................................... 10  
   1.5 Methodology / Methods .............................................................................. 10  
   1.6 Delimitations .............................................................................................. 10  
   1.7 Outline ........................................................................................................ 10  

2 General Analysis of Embedded Software Vulnerabilities .......... 12  
   2.1 General Definition ...................................................................................... 12  
      2.1.1 Embedded System ............................................................................... 12  
      2.1.2 Security Principles .............................................................................. 12  
         2.1.2.1 Security Goals ............................................................................... 12  
         2.1.2.2 Security Terminology .................................................................... 13  
         2.1.2.3 Ensuring Security ......................................................................... 13  
   2.2 Vulnerabilities in Embedded Systems .................................................... 14  
      2.2.1 Characterisation of attacks on embedded systems ............................... 15  
         2.2.1.1 Attacks against core functionalities ............................................. 15  
         2.2.1.2 Attacks on fault-tolerance mechanisms ....................................... 19  
         2.2.1.3 Attacks on firmware updates ...................................................... 20  
      2.2.2 Generic Taxonomy ............................................................................... 20  
      2.2.3 Taxonomy for sub-fields of embedded systems .................................. 23  
   2.3 Vulnerabilities Classification in Embedded Systems ..................... 23  
      2.3.1 Programming Errors .......................................................................... 24  
         2.3.1.1 Buffer Overflows .......................................................................... 24  
         2.3.1.2 Memory Management Error ....................................................... 27  
         2.3.1.3 Input Sanitisation and Format String ......................................... 27  
         2.3.1.4 Integer Overflows ........................................................................ 29  
      2.3.2 Improper Use of Cryptography ............................................................ 30  
      2.3.3 Weak Access Control or Authentication .............................................. 30  
         2.3.3.1 System-Level Attacks ................................................................. 30  
         2.3.3.2 Information Disclosure ............................................................... 31  
   2.4 Overview of Literature Review ................................................................. 32  

3 Toward Secure Embedded Software Development .............. 33  
   3.1 Basic Security Mechanisms ....................................................................... 33  
      3.1.1 Cryptography ...................................................................................... 33  
         3.1.1.1 Hash Functions ............................................................................ 33  
         3.1.1.2 Encryption Algorithms ............................................................... 33  
      3.1.2 Access Control .................................................................................... 34  
         3.1.2.1 Access Control Policy Models .................................................... 34  
         3.1.2.2 ACL and Capability-Based Approach .......................................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Company’s Focus</td>
<td>36</td>
</tr>
<tr>
<td>3.3 Related works</td>
<td>36</td>
</tr>
<tr>
<td>3.3.1 General Software Approach</td>
<td>36</td>
</tr>
<tr>
<td>3.3.2 Embedded Software Focus</td>
<td>37</td>
</tr>
<tr>
<td>3.3.3 Learnings from previous work</td>
<td>39</td>
</tr>
<tr>
<td>4 Thesis’s Work Methodology</td>
<td>40</td>
</tr>
<tr>
<td>4.1 Knowledge enhancement Document Writing</td>
<td>40</td>
</tr>
<tr>
<td>4.1.1 Method</td>
<td>40</td>
</tr>
<tr>
<td>4.1.2 Quality Assurance</td>
<td>40</td>
</tr>
<tr>
<td>4.2 Analysis of vulnerability and countermeasures</td>
<td>41</td>
</tr>
<tr>
<td>4.2.1 Method</td>
<td>41</td>
</tr>
<tr>
<td>4.2.2 Quality Assurance</td>
<td>41</td>
</tr>
<tr>
<td>5 Knowledge Enhancement Document Creation</td>
<td>42</td>
</tr>
<tr>
<td>5.1 Process Description</td>
<td>42</td>
</tr>
<tr>
<td>5.2 Scope of the guidelines</td>
<td>42</td>
</tr>
<tr>
<td>5.2.1 Content</td>
<td>42</td>
</tr>
<tr>
<td>5.2.2 Goal</td>
<td>42</td>
</tr>
<tr>
<td>5.2.3 Audience</td>
<td>42</td>
</tr>
<tr>
<td>5.2.4 Form</td>
<td>43</td>
</tr>
<tr>
<td>5.3 Building the draft</td>
<td>46</td>
</tr>
<tr>
<td>5.4 Improving our Document</td>
<td>46</td>
</tr>
<tr>
<td>5.4.1 Interview’s Feedback Summary</td>
<td>47</td>
</tr>
<tr>
<td>5.4.2 Bureau Veritas’ Document Analysis</td>
<td>48</td>
</tr>
<tr>
<td>5.5 Final Document</td>
<td>49</td>
</tr>
<tr>
<td>6 Security Analysis of Client’s Project</td>
<td>50</td>
</tr>
<tr>
<td>6.1 Recommendations</td>
<td>50</td>
</tr>
<tr>
<td>6.2 Example of Security Analysis: Attack Trees</td>
<td>51</td>
</tr>
<tr>
<td>6.3 Performance Analysis of AES</td>
<td>52</td>
</tr>
<tr>
<td>6.4 Static Metrics Monitoring Tool</td>
<td>56</td>
</tr>
<tr>
<td>6.4.1 Project’s need</td>
<td>56</td>
</tr>
<tr>
<td>6.4.2 Tool Principle</td>
<td>57</td>
</tr>
<tr>
<td>6.4.3 Results</td>
<td>58</td>
</tr>
<tr>
<td>6.5 Dynamic Metrics Analyser</td>
<td>59</td>
</tr>
<tr>
<td>6.5.1 Embedded Metrics Recording</td>
<td>59</td>
</tr>
<tr>
<td>6.5.2 Unembedded Analysis</td>
<td>60</td>
</tr>
<tr>
<td>6.5.3 Full Process Working</td>
<td>60</td>
</tr>
<tr>
<td>6.5.4 Possible Improvements</td>
<td>61</td>
</tr>
<tr>
<td>6.6 Preliminary Discussions</td>
<td>62</td>
</tr>
<tr>
<td>7 D-Bus Security Analysis</td>
<td>64</td>
</tr>
<tr>
<td>7.1 Interest</td>
<td>64</td>
</tr>
<tr>
<td>7.2 D-Bus Principles</td>
<td>64</td>
</tr>
<tr>
<td>7.2.1 General Presentation</td>
<td>64</td>
</tr>
<tr>
<td>7.2.2 Building Blocks</td>
<td>65</td>
</tr>
<tr>
<td>7.2.2.1 Connections and Services</td>
<td>65</td>
</tr>
<tr>
<td>7.2.2.2 Objects</td>
<td>65</td>
</tr>
<tr>
<td>7.2.2.3 Interfaces</td>
<td>65</td>
</tr>
<tr>
<td>7.2.2.4 Properties</td>
<td>65</td>
</tr>
<tr>
<td>7.2.2.5 Methods</td>
<td>65</td>
</tr>
</tbody>
</table>

2
Appendix D. Measurement Parameters

Appendix C. Default Policies

Appendix B. Hardware Specifications

Appendix A. Interview Structure

References

8 Conclusion on Thesis Work

8.1 Deliveries

8.2 Conclusion and Evaluation

8.3 Personal Learning and Limitations

8.4 Future Work

References

Appendix A. Interview Structure

Appendix B. Hardware Specifications

Appendix C. Default Policies

Appendix D. Measurement Parameters
Table of Illustrations

Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Matrix Organisation of Thales Services</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Relations between the different entities involves in the security of a system (OWASP Top 10 Risks)</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Example of the principle of a Denial-Of-Service attack on the CPU resources by using a malicious task which exhausts resources</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Example of an attack based on forging message to impersonate the sending process and execute arbitrary code</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>Relations between the different dimensions of the taxonomy (original picture received from the author of [21])</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Graphical representation of the memory layout</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>Simple example of a sanitisation happening at the wrong place leading a vulnerability to reappear in new projects</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Representation of Access Control Matrix, Access Control List, and Capability Approach</td>
<td>35</td>
</tr>
<tr>
<td>9</td>
<td>Link between securing techniques and their corresponding SDLC step(s)</td>
<td>38</td>
</tr>
<tr>
<td>10</td>
<td>Sketch of an attack tree dealing with the case of accessing critical data</td>
<td>52</td>
</tr>
<tr>
<td>11</td>
<td>Encryption/Decryption time of AES protocol depending on the key length for a log 3kB</td>
<td>55</td>
</tr>
<tr>
<td>12</td>
<td>Encryption/Decryption time of AES protocol depending on the key length for a log 3MB</td>
<td>56</td>
</tr>
<tr>
<td>13</td>
<td>Step of the Jenkins job of the static analyser</td>
<td>58</td>
</tr>
<tr>
<td>14</td>
<td>Example of a binary-centric graph: evolution of the metrics depending on the versions of the project for one of the process</td>
<td>59</td>
</tr>
<tr>
<td>15</td>
<td>Full process working of the dynamic metrics analyser</td>
<td>61</td>
</tr>
<tr>
<td>16</td>
<td>Resulting graph generated by the tools representing the CPU usage (in percent) of one of the process</td>
<td>62</td>
</tr>
<tr>
<td>17</td>
<td>Representation of an example of use of D-Bus with a simple architecture (From D-Bus Documentation [48])</td>
<td>67</td>
</tr>
<tr>
<td>18</td>
<td>Summary of the architecture of our system for the use case</td>
<td>71</td>
</tr>
<tr>
<td>19</td>
<td>Graphical representation between the number of additional rules and the overhead</td>
<td>80</td>
</tr>
</tbody>
</table>
Tables

Table 1. Presentation of the different dimensions and their layers in the taxonomy built in [21] ..............................................................................................................21
Table 2. Software Process introduced in IEEE 1074...........................................45
Table 3. Parameters used to brute force the AES Algorithm .........................53
Table 4. Time to brute force AES depending on the key length ....................53
Table 5. Executed Script Shell Code for the benchmark.......................................54
Table 6. Time to encrypt/decrypt different files with AES depending on their size and the key length of the cryptographic mechanisms (values in parentheses represent the standard deviation) ........................................................................55
Table 7. Summary of the different components of D-Bus and their naming convention.......................................................................................................................66
Table 8. Computational overhead measure by perf tool during the execution of the core system depending on the policy ..............................................................80
Table 9. Atmel SAMA5D3 Xplained Specifications........................................93
Table 10. Policy of the session configuration file .................................................94
Table 11. Policy of the system configuration file ................................................94
Table 12. Header of the performance report.......................................................95
Acronyms

ACID  Atomicity, Consistency, Isolation, and Durability
ACL   Access Control List
AES   Advanced Encryption Standard
ANSSI Agence National de la Sécurité des Systèmes d’Information
CCTV  Closed-Circuit TeleVision
COTS  Commercial Off-The-Shelf
CPU   Central Processing Unit
CVE   Common Vulnerabilities and Exposures
CWE   Common Weakness Enumeration
DAC   Discretionary Access Control
D-Bus Desktop Bus
DDoS  Distributed Denial of Service
DES   Data Encryption Standard
DMA   Direct Memory Access
DoS   Denial of Service
DPA   Differential Power Analysis
GNU   GNU’s Not Unix (Recursive Acronym)
HMI   Human Machine Interface
HPET  High Precision Event Timer
IDS   Intrusion Detection System
IPC   Inter-Process Communication
LSM   Linux Security Module
MAC   Mandatory Access Control
MD5   Message Digest v5
NIST  National Institute of Standards and Technology
OS    Operating System
OSI   Open Systems Interconnection
OWASP Open Web Application Security Project
PIT   Programmable Interval Timer
PLC   Programmable Logic Controller
POC   Proof-Of-Concept
RAM   Random Access Memory
RBAC  Role-Based Access Control
RNG   Random Number Generator
RSA   Rivest-Shamir-Adleman
RTC   Real Time Clock
RTOS  Real Time Operating System
SCADA Supervisory Control And Data Acquisition
SDLC Software Development Life Cycle
SHA   Secure Hash Algorithm
SMART Specific, Measurable, Attainable, Realistic, and Time-Bound
SMM   System Management Mode
TSC   Time Stamp Counter
1 Introduction

1.1 Background

1.1.1 Thales Services SAS

Thales Services SAS is one of the subsidiaries of the French multinational company Thales Group. Thales Group deals mainly with electronics in different fields as defence, security, transportation, and aerospace through different affiliates, for instance, Thales Communication or Thales Underwater Systems. Thales Group, with 62 000 employees, including 20 000 engineers, is spread over 56 countries and had a turnover of around EUR 14 billion in 2015.

One of the global activities of Thales is about Secured Information Systems and Communications (SIX). Integrated in this activity, Thales Services SAS is one of the components of the business line about Critical Information System and Cybersecurity (CIC) of Thales Group.

The mission of Thales Services is to develop and deliver IT services based on critical information systems. These services are provided during all the possible developing steps: from the preliminary study to the integration and exploitation of the solutions. Thales Services employs 3 700 people in France and its turnover was around EUR 415 million in 2015.

The expertise fields of the company are divided into four centres:

- User-Oriented Software (abbreviated LOU in French);
- Machine-Oriented Software (abbreviated LOM in French);
- Consulting, Technology, & Expertise;
- IT Operation and Cybersecurity.

These fields are spread over the business line which delimits the geographical areas: transversal region, Paris and Western region, South-Western region, and Eastern region. Due to this wide offer, Thales Services works on different markets like public services, energy, transportation, aeronautics, space, or social health-care. Figure 1 summarizes the organisation of Thales Services.
1.1.2 Internship Department

This internship is done on the site of Grenoble of Thales Services in the centre Machine-Oriented Software of the South-Eastern region. Due to the increasing demand of some clients about problems concerning embedded systems, this expertise field has started to hire and train more people about this specific kind of systems. The team of embedded systems currently gathers 20 people. They work on different projects, from communication technologies to medical area.

1.2 Problem

This thesis is at the intersection of two important phenomena. Firstly, embedded systems are becoming more and more present. As the Zion research team shows in [1], the global embedded system market revenues are likely to grow of 40% over six years, increasing from USD 150 billion in 2015 to USD 225.34 billion by 2021. The main driver of this growth would be the increasing demand for power efficient and smarter electronic devices. In addition, the new possibilities given by the automotive industry and the new applications provided by using multi-core architecture in embedded systems could accentuate this growth.

On the other hand, cybersecurity has taken a more important place in the design choice step than formerly. The market related to this area is planned to grow from USD 122.45 billion in 2016 to USD 202.36 billion by 2021 as underlined in [2]. Thales Group understood the importance of this new area and has decided to increase the importance of cybersecurity in its activity. It has become one of the European leader in cybersecurity with 5000 IT and security engineers and 1500 cybersecurity specialists. Thales Group takes care of the cybersecurity of 9 of the 10 Internet giants.

The intersection of these two areas leads to new challenges since embedded systems differ from general-purpose computers or data centres. Although some vulnerabilities are commons, the threats or the possible actions are different. In the case of Thales Services, the problem is to provide products
(e.g., software and consulting advice) that meet the expectations of the clients but also to ensure a level of security depending on the area of applications.

Therefore, it leads us to investigate what are the main and most critical software vulnerabilities that can be encountered in the area of embedded software. How is it possible to address these vulnerabilities and what are the issues which could appear due to the specificity of embedded systems?

1.3 Purpose

Thales Service created this master thesis to widen their knowledge about how to secure embedded systems. Thank to this work, it could allow the company to meet the business line requirements of the group. Moreover, the IT services market is quite competitive, clients have the choice between a wide range of companies. IT services companies should prove that they can add more value than the client expected. By developing their skills in this area, it could provide Thales Services a business advantage in comparison to the other IT services companies. The thesis deals with the vulnerabilities and they countermeasures in the case of embedded software.

1.4 Goal

The different goals are presented through the deliveries required by Thales and the benefits which are expected by the company.

1.4.1 Deliveries

A list of deliveries was defined by Thales Services at the beginning of the master degree project and updated through the thesis:

- The master thesis report: Track all the work done during the internship. This report must not be seen only as a document that acknowledges the work of the student for the university, it will be used by the company to increase the skills of the co-workers.
- A guidelines document: Create a document which gathers a set of guidelines and good practices related to the development of secure embedded software.
- A security analysis of a client’s project: Apply the guidelines to a real use case and provide solutions. These solutions should deal with:
  - Implementation of a static metrics monitoring tool;
  - Implementation of a dynamic metrics analyser;
  - Analysis and Securing an IPC (Inter-Process Communication) mean.
- Realisation of a presentation during a LunchTalk (monthly presentation done at Thales Services Grenoble, open to all employees of the company and done on the lunch time) to show the work done with the embedded team.

1.4.2 Benefits

As underlined previously, this project aims to increase the knowledge of the embedded systems team of Thales Services about security in this area. The guidelines document could be reused in different projects to ensure some
security standards, but also to teach company’s engineers about security good practices. Moreover, it could also benefit to its clients by applying this knowledge in the case of concrete problems.

In addition, the implementations of tools and proof-of-concepts (POCs) could be starting points for some features of other projects, or it could also be food for thought for making design choices.

1.4.3 Ethics and Sustainability

The number of embedded systems are constantly growing, especially with the Internet of Things, leading users to let these systems access to their private data or to do critical operations. Moreover, sustainability is not put aside as underlined by the current affairs about the suspicions of fraud in the automotive industry regarding embedded anti-pollution devices [3].

Therefore, although this work does not deal directly with ethics and sustainability, its applications are likely to be related to these areas.

1.5 Methodology / Methods

In the case of the first part about the investigation of the vulnerabilities and the ways to address them, a literature review was carried out by focusing on embedded software components which are more or less critical.

The realisation of the guidelines was based on our literature review. We followed a qualitative approach with additional literature review analyses and interviews to write these good practices.

The setting up of the solutions was based on the security analysis we did about a client’s project. The tools were developed for general-purpose computers whereas the implementation of the secured IPC was done on a real board with our own embedded Linux distribution. Some benchmarking solutions were implemented to analyse the performance of the embedded device.

1.6 Delimitations

Cybersecurity and embedded systems are large areas. This work focuses on the software part of embedded systems. Physical attacks on hardware is out of scope of this thesis. The company ran at the same time another master thesis about the analysis of the ARM TrustZone technology to look at this hardware aspect of cybersecurity. Moreover, a lot of vulnerabilities can be found on a general-purpose computer. This work highlights more the vulnerabilities which are critical due to the nature of embedded systems. In addition, the vulnerabilities linked to networked and distributed embedded systems (for instance, DDos or Wireless Security) are not investigated in this thesis.

1.7 Outline

The content of the thesis is structured as follow. Firstly, Chapter 2 introduces the case of security of embedded software. This literature review presents basic concepts of security, attacks on embedded software, and a classification of the related vulnerabilities.
Chapter 3 deepens the previous literature review by giving a presentation of basic security mechanisms. It introduces the company’s needs and the work related to the redaction of good practices in this area.

Chapter 4 describes the different methodologies which were used during the thesis to realise the different tasks. It also provides an overview of these tasks.

Then, Chapter 5 presents our process for writing a document which gathers guidelines and good practices for secure embedded software development. This chapter also contains a first analysis of the resulting document.

Chapter 6 presents the application of our guidelines on a real case: a client’s project. The analysis done by applying our good practices led to the realisation of some tasks (e.g., analysis, tools, and proof-of-concepts) presented in this chapter.

Chapter 7 deals more in-depth than the previous chapter with one feature which was identified as critical by the company: securing D-Bus (Desktop Bus). This chapter gathers the presentation of this mean of communication, the security analysis, and the implementation of a use case with the related conclusions.

Chapter 8 draws a generic conclusion over the entire work done during this internship. It summarizes the different realisations and the related learnings.
2 General Analysis of Embedded Software Vulnerabilities

2.1 General Definition

The next section introduces general concepts about embedded systems and security before diving into the subject and presenting the relation between these two areas.

2.1.1 Embedded System

An embedded system, as defined in [4], “is an electronic product that contains a microprocessor (one or more) and software to perform some constituent functions within a larger entity”. However, some points should be added to describe more accurately embedded systems. These systems, as [5] underlined, have access to limited hardware resources (e.g., computation speed, energy supply, and memory) leading also to limit the software functionalities provided by this kind of system. [6] highlighted the importance of these interactions between the hardware and the software by using the term cyber-physical system. Embedded system is not the juxtaposition of software applications and pieces of hardware; it is also about the tight relation between the physical and the computational environments. In addition, embedded systems interact with their environment. Therefore, they should meet higher reliability and quality requirements than general-purpose computers. This level of requirement could be raised in the case of the most safety critical applications which require real-time interactions with deadlines to meet.

2.1.2 Security Principles

2.1.2.1 Security Goals

Security could be defined as the capacity of an entity to set up mechanisms to protect resources for which it has received protection responsibility [4]. This capacity is built around the following requirements [7]:

- Confidentiality: Data cannot be read by an unauthorised entity.
- Integrity: Data cannot be modified by an unauthorised entity.
- Availability: An authorised user should be able to access to the different services provided by the system/application.
- Authentication: Any entity claiming a specific identity must prove it, this entity must be authenticated.
- Authorisation: It states the different actions which can be performed by an authenticated entity over the resources provided by the system/application.
- Accountability: It refers to the process of controlling what actions entities do on the system’s resources.
- Non-repudiation: An entity cannot deny the fact of having done a specific action.
2.1.2.2 Security Terminology

As far as security risks are concerned, a terminology is used to describe the different components which can be encountered during a security assessment of a system by the Open Web Application Security Project [8]:

- **Asset:** It refers to any person or object who or which should be protected.
- **Vulnerability:** It is a gap or a weakness in all the security efforts set up to protect a program. It can be used by a threat to access without authorisation to an asset.
- **Weakness:** It is not necessarily a vulnerability. It is a software problem that could lead to be a vulnerability.
- **Threat (or Threat Agent):** It designates anything which can exploit, accidentally or intentionally, a vulnerability leading it to access to an asset and deteriorate it. The term “attacker” is also used to refer to a threat.
- **Exploit (or Attack Vector):** It refers to the way a threat takes advantage of a vulnerability.
- **Risk:** It is the intersection of these three concepts: asset, vulnerability, and threat. A risk is the possibility that a threat uses a vulnerability to damage an asset and impact a system.
- **Countermeasures (or Control):** It describes the security techniques implemented to protect the assets of the systems.

![Figure 2. Relations between the different entities involves in the security of a system (OWASP Top 10 Risks)](image)

Figure 2 shows how an attacker could use a vulnerability to impact a system. For instance, a malware is a threat. A buffer overflow in our program is a vulnerability. There is a risk that the malware uses this buffer overflow to damage our system. It could exploit it by sending a specific input to our program.

2.1.2.3 Ensuring Security

The main difficulty of securing a system lies in the asymmetric nature of security. A single vulnerability is sufficient for an attacker to damage a system, whereas a defender must find and solve all the possible vulnerabilities that could appear during the development process. The latter is quite impossible due
to the difficulty to list all these possibilities, because at the time of development, some features could be identified as non-critical as no attacker has tried to exploit it as a vulnerability before.

Embedded systems have common security concerns with general-purpose computers like the confidentiality of communications or the integrity of systems. However, due to their design constraints, some well-known vulnerabilities of general-purpose computers can become easier to exploit or more dangerous. Moreover, as underlined by Markus Maybaum in [9], one scientist at NATO Cooperative Cyber Defence Centre Of Excellence's Research and Development Branch, the constraints about cost, size, and resources lead to trade-offs, which reduce the opportunity of developing high security mechanisms. The cyber-physical nature of embedded systems offers new attacks surfaces and new assets to target like the limited computation power, the battery, or the actuators.

Time-to-market is also a design requirement that could be the source of lowering security policies. Due to the high competitiveness of the companies in this area, there is a desire to be the first one to bring a new product on the market. However, it leads to shorten some steps in the design part of the software life cycle like defining the requirements about security or testing. Although patching has appeared as a good solution for software companies, especially to solve vulnerabilities discovered after the release of a product. The author of [10] underlined this security problem in the case of IoT. Security requirements must be integrated as soon as possible in the design process of embedded systems. When a vulnerability is discovered in the case of IoT, it could be more harmful than a vulnerability on a general-purpose computer since, in the latter, it is often possible to push patches to the different computers of a network, or to a specific brand of OS (e.g., the remote update centre of MS Windows, OS X and Linux). On the contrary, embedded systems, such as modems and routers, often need that users manually download the updates which leads these embedded devices to rarely be updated. Developers must take precautions to reduce the risk of apparitions of vulnerabilities in these systems.

Therefore, ensuring security in embedded systems does not deal with only applying security techniques to embedded systems. Every design choice is often in disfavour of other ones.

### 2.2 Vulnerabilities in Embedded Systems

A vulnerability is defined regarding the possibilities to be exploited by an attacker. Therefore, the analysis of the vulnerabilities should be done by analysing possible threats. In addition, a vulnerability is defined in the case of one software program which underlines the fact that each vulnerability is specific to one implementation of one embedded system. According to these points, and to better understand the problem of security in embedded systems, we investigated some classifications of attacks on embedded systems. By looking at the possible attacks, it led us to identify some classes of vulnerabilities.

As described in the delimitation of the thesis, the investigation is focused on the attacks on software of embedded systems. Hardware attacks (e.g., physical tampering, probing, or side-channel attacks) are not analysed.
2.2.1 Characterisation of attacks on embedded systems

As it was underlined, attacks and vulnerabilities are system-dependent. [11] classified software-oriented attacks on embedded systems and the vulnerabilities which allow these attacks. Although this analysis was done in the case of an embedded system for avionics, the main classes of attacks could be reused in most of embedded systems.

The classification was established by assuming that embedded systems have multiple similarities with the general-purpose information systems. Therefore, a first set of attacks could be derived from these similarities. They are referred as the attacks against core functions. The author dealt with another class of attacks: the attacks against fault-tolerance mechanisms. This set of attacks is more related to the most safety-critical embedded applications. Since our work deals with embedded systems in a general manner, these one were less investigated. In addition, we will also deal with the attack related to firmware updates.

2.2.1.1 Attacks against core functionalities

Attacks on the processor

Either in general-purpose computers, or embedded systems, this piece of hardware is the core component of any computer. It oversees the execution of the different instructions. CPUs are becoming more and more complex by integrating new improved features to increase their performance. This growth of complexity also increases the possibilities of apparitions of vulnerabilities.

Firstly, since CPUs often rely on caches to speed-up the computations, the management of these small memories could be critical. Cache could be a back-door to access to cryptographic keys or other critical pieces of information. Therefore, they could be the source of data flow.

Moreover, Denial-of-Service (DoS) attacks can be carried out on a CPU by asking it to execute undefined instructions [12] and by exhausting its resources as showed in the Figure 3. It could lead the CPU to an unpredictable state which would be critical for the system.
Attacks on memory management

The term “memory” represents different kinds of memory: from the RAM to the registers, including the memory parts of the I/O controllers. Therefore, this set of attacks could have different targets. However, their goal remains the same: modifying or accessing to the content of the memory to, either corrupt the systems, or access to some pieces of information. After presenting the possible approaches to access to memory, we will deal with the targets of these attacks.

Way of accessing to the memory

The main way to access to the memory is through the processor. Normally, some security policies regarding the privileges of the applications are enforced to prevent any malicious applications to access to the memory. However, by exploiting vulnerabilities like buffer overflows or badly-formatted string inputs, it could lead the system to execute malicious code or to do a privilege escalation. Moreover, depending on the location of the vulnerability, the attack can be more powerful. If the vulnerability is located in the code of the kernel, it could grant an attacker with the highest privilege level, giving him a full control of the system. This attack vector is often due to implementation mistakes that create these vulnerabilities.

Memory can also be accessed through the Direct Memory Access mechanisms (DMA) thanks to the I/O controllers. This technique allows to access to the main memory, but also to the memory of the other controllers. The latter type of access is referred as peer-to-peer attacks since they do not require to go through the main memory, it is a direct communication between the controllers. This set of attacks cannot be applied to all kinds of chipsets.
Moreover, since they do not require the intervention of the CPU, they are more difficult to detect than the attacks which access to the main memory.

**Characterisation of memory management attacks**

The attacks regarding memory access can have different targets behind the memory: the applications, the OS, the hypervisor, or the CPU environment.

- **Application**: Targeting the memory of one specific application could be difficult to discover. The attacker could focus on one application due to its specific function, to escalate its privileges, or to impact the execution of the applications which share resources with the targeted one.
- **OS**: Attacking the OS kernel could give a full access to the OS to an attacker, allowing him to manage the different applications. Kernel rootkits are an example of malicious software developed to corrupt the kernel by exploiting actual vulnerabilities.
- **Hypervisor**: In the case of embedded systems, hypervisors could allow to isolate different systems to deal with security concerns and mix-critical system. Therefore, attacking the memory of a hypervisor could allow an attacker to get outside the virtual machine and access and control the whole system of virtual machines.
- **CPU environment**: An attacker could be interested in modifying some specific registers of the processor to corrupt it, to modify the interruptions, or to run malicious code.

**Attacks on communications**

Dessiatnikoff, in [11], analysed the Inter-Process Communications (IPC) mechanisms. They are used to exchange data at the application level. They can operate at different levels; processes could run over the same kernel or different ones.

These attacks, which target the application layer of the OSI model, are similar to generic attacks on communication channels. Different threats could prevent the strict application of the security principles as presented in [13]. The main four types of threats are:

- **Interception**: Data or services are accessed by an unauthorised party.
- ** Interruption**: An attacker harms the system to prevent an authorised user from accessing and using data and services.
- **Modification**: An unauthorised user changes data or a service leading it to differ from its original version.
- **Fabrication**: Data or services are created by an unauthorised user.

These threats could be realised through these five main communication channel attacks:

- **Eavesdropping**: An attacker accesses to the channel and can listen, without authorisation, to the channel.
- **Masquerading**: An attacker sends or receives messages by impersonating another communicant.
- **Denial of Service**: The goal of this attack is to interrupt a service, usually by flooding the channel with messages.
• Replay: An attacker reuses older messages to send them again on the communication channel to trigger some specific behaviours.
• Message Tampering: An attacker intercepts and changes the content of the messages.

Combination of these different attacks could be done depending on the goal of the attacker, but also the security policies of the communication channel. A well-known attack which combines these different attacks is the man-in-the-middle attack where the attacker inserts himself between the two communicants, like a checkpoint, without informing them. Therefore, the attacker can eavesdrop the messages, impersonate one communicant, forge some messages, drop some of them, and resend older messages. [14] sketched a simple attack based on forging messages with all possible IDs and sending them to the receiver to execute desired instructions. Figure 4 shows the principle of this attack.

![Figure 4. Example of an attack based on forging message to impersonate the sending process and execute arbitrary code](image)

**Attacks on time management**

Current systems provide different clocks or interval timers to cadence the processes, for instance: the Real Time Clock (RTC), the Time Stamp Counters (TSC), the Programmable Interval Timers (PIT), or the High Precision Event Timer (HPET). These clocks differ by their features like their frequency or the number of bits used in their counter. In the case of a RTOS, some processes could have to meet hard-deadlines. Therefore, an attack on the speed of the clock could be critical. An attacker could slow down the speed of the clock, leading the RTOS not to be synchronised with the environment. Even if the
processes continue to meet their deadlines in the RTOS scheduler, this
difference of pace would cause these processes to miss their deadlines in the
environment.

**Attacks on process management**

The OS should be able to run different processes at the same time. Then,
it should be able to split efficiently the different resources. One task cannot use
all the CPU resources letting the other ones waiting. Two processes cannot write
at the same time in the same shared resource (e.g. memory, files, buses, and
drivers). An attacker could leverage the process management to harm the
system. As we underlined, it could do a DoS attack by exhausting the resources
of the CPU. But it can also create a deadlock by modifying resources in charge
of the synchronisation. A deadlock appears when each process of a group is
waiting for another process of the group to release a lock, leading all the
processes of the group to wait. The problem of deadlock is also common to the
IPC.

**Attack on scheduling**

As we highlight in the previous paragraph, several threads could be
executed at the same time on one processor. It requires the OS to split their
execution time on this processor. Moreover, some tasks could have higher
priority than other tasks with some deadlines to meet. The goal of the scheduler
is to ensure that the rules about the priorities, the deadlines, and the retribution
of the CPU processing time are enforced. Therefore, an attack on these features
could lead to create deadlocks or to miss deadlines, which is harmful in the case
of a RTOS. Chun-Tao et al. presented in [15] a study about priority inversion in
the case of embedded inversion. Depending on the scheduling policies used by
the OS, an attacker could lead low-priority tasks to steal execution time of
higher priority tasks.

**Attack on cryptographic mechanisms**

Depending on the strength of the cryptographic mechanisms, an attacker
could try to access, without permissions, to the system. For instance, collisions
were found for the MD5 cryptographic algorithms as underlined in [16].
Moreover, by leveraging some attacks on memory management, an attacker
could try to retrieve a part of a secret key (or the full key) giving him the
possibility to access to the system.

**Attack on ancillary functions**

Ancillary functions refer to the functions dealing with all secondary tasks
like the temperature management, power supply management, or the over-
clocking. The System Management Mode (SMM) oversees these functions in
the case of the x86 processors. New vulnerabilities have been discovered in this
area for few years leading an attacker to run code with high privileges and to
directly harm physical components of the system.

2.2.1.2 **Attacks on fault-tolerance mechanisms**

Since the fault-tolerance mechanisms are specific to the area of safety-
critical embedded systems, this kind of attacks is beyond the scope of this work.
However, it is important to underline that they could be the target of the attackers by increasing the attack surface. Therefore, when setting up fault-tolerance mechanisms, developers should not only think about protecting the system from environmental failures, security requirements should be integrated in the design steps of these components. An attack against one of these mechanisms could prevent the system from recovering from a failure.

2.2.1.3 Attacks on firmware updates

These attacks were not classified in the thesis of Dessiatnikoff in [11] due to the scope of his topic. The goal of this class of attacks is to put a modified version of a firmware to execute and access to critical processes and data.

As we highlighted in the beginning of this section, patching embedded systems is not an easy task. However, firmware updates are common in the case of printers [17] and routers [18] and they could be used by an attacker to hijack the system. This set of attacks could be classified into the class of attacks on memory since they aimed to alter the part of the memory containing the firmware but also on cryptographic mechanisms. Abrahamsson analysed in [19] the update process of firmware and underlined that the features related to the integrity checking, the authentication through digital signature, or the version management could be targeted.

2.2.2 Generic Taxonomy

Although [11] tried to list all possible classes of attacks on his system, it is not possible to say that the list is exhaustive because some possible attacks could have not been discovered yet.

As Igere and Williams showed in [20], using taxonomy is generally a good way to approach a new area since it allows classifying and describing the different elements of this field. Moreover, it could allow realising a security assessment of a software. Although embedded systems have common vulnerabilities with general-purpose computers, it is better to rely on a taxonomy which is specific to a set of computer systems [20].

Papp et al. developed in [21] an attack taxonomy for embedded systems. This classification is interesting for several reasons. Firstly, it meets the requirements defined by [20] to create a good classification:

- It should be specific to an application or a system.
- It must have a layered or hierarchical architecture.
- No mutual-exclusion should be ensured between the different classes.
- The level of classification should deal with the attack impact, attack type, attack targets, and the source of vulnerability.

Moreover, Papp et al. applied their taxonomy to the CVE, the Common Vulnerabilities and Exposures [22]. It is a system used to reference the different information-security vulnerabilities and exposures, which have been disclosed publicly. By filtering the vulnerabilities from the CVE list related to embedded systems, they highlighted the main trends in attacking embedded systems.

They used five dimensions to establish their classification. These dimensions and the different layers are summarized in Table 1. The layers about
attack targets are more general than the classification brought by Dessiatnikoff, allowing to have a more general approach.

Table 1. Presentation of the different dimensions and their layers in the taxonomy built in [21]

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Precondition</th>
<th>Vulnerability</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet facing device</td>
<td>Programming errors</td>
<td>Hardware</td>
<td></td>
</tr>
<tr>
<td>Local of remote access to the device</td>
<td>Web based vulnerability</td>
<td>OS/Firmware</td>
<td></td>
</tr>
<tr>
<td>Direct physical access to the device</td>
<td>Weak access control or authentication</td>
<td>Application</td>
<td></td>
</tr>
<tr>
<td>Physically proximity of the attacker</td>
<td>Improper use of cryptography</td>
<td>Device</td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Insecure Configuration</td>
<td>Protocol</td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Attack Method</th>
<th>Attack Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control hijacking attacks</td>
<td>Code Execution</td>
<td></td>
</tr>
<tr>
<td>Reverse engineering</td>
<td>Integrity violation</td>
<td></td>
</tr>
<tr>
<td>Malware</td>
<td>Information leakage</td>
<td></td>
</tr>
<tr>
<td>Injecting crafted packets or input</td>
<td>Illegitimate access</td>
<td></td>
</tr>
<tr>
<td>Eavesdropping</td>
<td>Financial loss</td>
<td></td>
</tr>
<tr>
<td>Brute-force search attacks</td>
<td>Degraded level of protection</td>
<td></td>
</tr>
<tr>
<td>Normal use</td>
<td>Miscellaneous</td>
<td></td>
</tr>
<tr>
<td>Unknown</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Although the term used to describe the different layers are quite explicit, an interested reader could refer to [21] to get the complete list of definitions. However, we will define some of them which are related to vulnerabilities since our study is focused on this point. Before defining the vulnerabilities, we would like to explicit the use of the layer “Unknown” in the different dimensions. Since the authors based their analysis on the CVE list, this category is used when a dimension was not described in the list (e.g., no description of the vulnerability itself or of the attack method) or when it was specific to a unique case which did not allow the authors to classify it in one layer of the dimension.
We can see that the researchers split the vulnerabilities between four categories ("Unknown" excluded):

- **Programming Errors:** This category gathers the CVE records related to programming errors like buffer overflows or bad management of pointers. This kind of vulnerabilities allows an attacker to do control flow attacks.
- **Web based vulnerability:** It represents the vulnerabilities that can be found in the web base management interface.
- **Weak access control or authentication:** Using weak, default, hard-coded in plain text passwords are examples of weaknesses that lead to this set of vulnerabilities.
- **Improper use of cryptography:** Although cryptographic algorithms are a way to ensure some security policies, a bad management of the encryption keys or a bad implementation of the random number generator (RNG) could be exploited.
- **Insecure Configuration:** It represents the cases where the system was set up with parameters which are defined as insecure like permissive whitelists or enabling a telnet connection.

Then, the researchers classified the CVE records based on their taxonomy. Figure 5 shows their results. The thickness of a line represents the density of CVE records which are in a category. From Figure 5, we can see that programming errors, web vulnerabilities, and weak access control or authentication are a source of a lot of vulnerabilities. These vulnerabilities mainly target the software components like the OS, the firmware, and the application. They allow to inject code and control the system. Another important point is the fact that these CVE records were mostly present on devices connected to the Internet.

However, we must take a step back about this point. Since the CVE list is based on publicly known vulnerabilities, open systems are more likely to be listed in these records than proprietary closed systems. For instance, automotive or avionic companies do not put in the CVE records the vulnerabilities which were present in their systems to avoid giving information to an attacker about their systems. Therefore, we could not assume that Internet facing devices are more concerned with these vulnerabilities. However, as far as the analyses about the type of vulnerabilities is concerned, since developers are likely to use same development techniques in the different areas of embedded systems, we could deduce that the same set of vulnerabilities could be encountered with the same importance in the open- and closed-network devices (apart from the web-based vulnerabilities).
The fact that programming errors are a bigger source of vulnerabilities is underlined in [23]. From 1988 to 2012, buffers errors represented 35% of the vulnerabilities types with a critical severity in the whole CVE records.

### 2.2.3 Taxonomy for sub-fields of embedded systems

Some taxonomies were built to deal with some specific areas of embedded systems. We looked at three of them to confirm the data extracted from the general taxonomy:

- In [24], the authors introduced a taxonomy about cyber-attacks on SCADA systems. SCADA systems, which stand for Supervisory Control and Data Acquisition, are used to monitor and control physical systems. They are based on complex networks with central computers, Programmable Logic Controller (PLC), and other embedded devices.
- In [25], a taxonomy about networks of embedded sensors was presented.
- [26] dealt with Closed-Circuit Television (CCTV) and video surveillance systems.

These taxonomies presented the same kinds of vulnerabilities and attack vectors introduced in [21], for instance: programming errors like buffer overflows or management of the privileges and access control. However, depending on the systems, some attacks vectors are easier to set up than others, leading a paper to describe them more accurately. [24] gave some importance to the description of attacks based on the communication stack, whereas [26] addressed specific attacks like visual layer attacks and covert channel attacks in one section of the article.

### 2.3 Vulnerabilities Classification in Embedded Systems

Based on the possible attacks which can be done on embedded systems and the result of the CVE analysis done by Papp et al., we decided to focus on a set of vulnerabilities which are more related to embedded devices. Web-based
vulnerabilities are not presented here. The fact that these vulnerabilities could damage embedded systems is a side-effect.

The following classes of vulnerabilities will be presented: programming errors, improper use of cryptography, and weak access control or authentication. Although improper use of cryptography was not represented so much in the results showed in Figure 5, we decided to present this class of vulnerability because it could be related to the vulnerabilities about weak passwords.

In this section, the different classes of vulnerabilities are presented and organised in the layers introduced in the taxonomy from [21]. Some layers of abstraction were added to better understand the differences between them, based on the work introduced by Thompson and Chase in [27].

2.3.1 Programming Errors

2.3.1.1 Buffer Overflows

Buffer overflows are one of the most common vulnerabilities. A buffer overflow is likely to happen in software which is written with programming languages which do not check the boundaries of the arrays. The main idea to trigger a buffer overflow is to access to a value of an array through an index value which is bigger than the actual array size. It leads to access to an area outside of the array. By accessing to this area, an attacker could read or write data in a memory location where he was not supposedly able to access. Buffer overflows can target different parts of the memory: it could be the stack, the heap, or the memory of other components.

As Hoglund and McGraw underlined in [28], although applications for computers servers are more and more robust to buffer overflows by applying new cutting edge techniques, these vulnerabilities have shift to the area of embedded systems where the structure of the code and the programming language lead these vulnerabilities to appear more easily.

In the next paragraphs, we decided to focus on the presentation of the stack and heap overflows. Buffer overflows on other memory structures remain similar. Before presenting them, a short presentation about the basics of memory architecture is done to understand these cases.

Memory Presentation

This section does not deal with the memory at the hardware level or the memory management done by the OS. We focus on the memory layout presented in [29] in the case of C programs. This memory layout is also used in the case of other languages.

The memory layout is split between five different components: the text segment, the initialised data segment, the uninitialised data segment, the stack and the heap.

- Text segment: It consists of instructions which are executed by the CPU. Since different programs can execute it, and to avoid unnecessary copy, this segment is often sharable. However, the access is usually in read-only mode to avoid any modification of the code.
• Initialised data segment: It contains the variables which are explicitly initialised inside the program code. This segment is also referred as the data segment.

• Uninitialised data segment: This segment gathers, on the contrary to the previous segment, uninitialised variables. The kernel initialises these variables to the arithmetic value “0” or to the “null” pointer. This segment is often referred as the “bss” segment which stands for Block Started by Symbol, the name of an operator from the old assembler UA-SAP (United Aircraft Symbolic Assembly Program).

• Stack: This segment is in charge of storing automatic variables and related pieces of information which have to be saved at each call of a function. Due to this feature, the stack does not have a fix size during the program execution, it increases and decreases as the program is executed. When a function is called, the return address and pieces of information related to the caller’s environment are saved on the stack. Then, the called function allocates spaces on this stack for its variables. When the execution of this function is finished, its data is removed from the stack. The stack uses a LIFO structure: “Last In, First Out”. The operations of putting and removing data on the stack are often referred respectively as “pushing” and “popping”.

• Heap: This component contains the variables which are allocated dynamically. Like the stack, its size is not fixed during the execution of the program.

Figure 6 shows a graphical representation of this abstract layout. Usually, the stack starts at high addresses and grows towards lower addresses. This picture gives a general idea of the memory organisation. However, depending on the implementation, there could be some differences. The different segments could be non-continuous in the physical memory based on the memory mapping used by the OS.
Stack Overflows

Stack overflows occur when a variable is written on the stack and the size of this variable is larger than the allocated space in this area. Since the stack stores other variables, but also the return addresses which will be called at the end of the functions, these overflows could modify these values. Overflowing the allocated memory in the stack often leads to a segmentation fault: the software tries to access to a protected area of the memory. However, attackers do not limit themselves to raise a simple exception to crash the program. A common exploit is to modify a return address. Therefore, when the execution encounters this return address, it calls the function at this address. Depending on the privileges given to the application, the attacker could be able to run a critical application.

Heap Overflows

On the other hand, heap overflows target the heap of the memory. Since heap memory oversees the allocation and deallocation of dynamic memory during the execution of a program, its structure is more complex than the stack: it is an interlacing of allocated and free space. Therefore, overflowing an allocated area does not lead to directly access to another allocated area. Moreover, due to its dynamic aspect, it is harder to track the size of the free space between two allocated spaces. However, the way of exploiting it is similar to the previous exploit: changing the value of a variable which represents an address to access to an instruction located at this address. For instance, thanks to a heap overflow, we could modify the value representing a pointer to a file in
order to point to another file which is more critical, leading to unauthorised accesses to this critical file.

2.3.1.2 Memory Management Error

We underlined in the section about attacks that they could be carried out on different memory components. We saw that buffer overflows could be carried out on the stack or the heap. Moreover, the dynamic structure of the heap could lead to more vulnerabilities than a simple overflow. These vulnerabilities are referred as memory management errors. Candaele presented in [30] five common memory management errors which could be possible vulnerabilities in a program:

- Heap overflows and underflows: Overflows were presented in the previous section. Underflows, which are less common, occur when an operation of read or write is done before the allocated memory.
- Dangling pointers: Also called use-after-free pointers, these pointers point to objects which have been already freed.
- Double free: It refers to the action of freeing an object which has been already freed, causing a double free error.
- Invalid Free: This error occurs when the developer frees an object which has not been received allocated memory beforehand.
- Uninitialised reads: When memory is allocated to an object, it does not lead the object to be initialised in all cases (especially with programming languages like C and C++). Therefore, calling this uninitialised object could raise this error.

However, these vulnerabilities are not sufficient for an attacker to carry out a heap attack, he must gather an additional condition: he should be able to exploit the memory allocator. To exploit the memory allocator, it should be set in a predictable state. When this state is reached, different kinds of attack could be done based on the vulnerabilities created by the memory management errors. These attacks often aim to modify the behaviour of the allocator by illegally changing the meta-data of the heap.

- Heap Overflow Attacks: Their main principle was described previously. It is based on overwriting pieces of information.
- Heap Spraying Attacks: The goal of the attacker is to use the possibility that an application can read an arbitrary address of the memory. The attacker has put (“has sprayed”) executable code such as shell code in the heap and tries to execute this code through the application.
- Dangling Pointers: Dangling pointers could lead to a double free vulnerability which could be used by an attacker to inject code.
- Off-by-One Error: This attack is a special case of a use of a buffer overflow where the attacker writes in the adjacent memory location by one byte exactly.

2.3.1.3 Input Sanitisation and Format String

Input sanitisation refers to the operation of checking inputs fed to a program to avoid having malicious contents inside the inputs which could be interpreted by the program in a wrong way. Format string vulnerabilities are an
example of the lack of sanitisation of the inputs. It is based on the fact that an attacker can leverage format functions provided by C/C++ programming languages to execute malicious instructions, modify the memory, and access to confidential pieces of information by using formatting character like “%x” or “%n”. Therefore, if the inputs are not checked and are directly fed into the formatting function, it could give access to confidential information to an attacker.

A trivial example is the use of the function `printf`. For a string input called `inUser`, calling `printf("%s",inUser)` only prints the value given by the user, whereas calling `printf(inUser)` is able to translate the formatting character fed by the user.

As far as format string vulnerabilities are concerned, since they target a specific class of functions, it is quite easy to fix them by sanitising the inputs of the related functions. However, this point shows that other vulnerabilities could appear based on the way the sanitisation is done. It is important to directly solve the problem at the source and not in one of the function calling the source of the problem. If the sanitisation is done beforehand too early, the vulnerability could occur again in next projects where the code is reused. If the critical function is directly called without going through the function in charge of sanitising the inputs, the vulnerability will become accessible again. This example is represented in the Figure 7. This case shows not only the vulnerability due to the lack of inputs sanitisation, but also the vulnerability due to incorrect sanitisation.
2.3.1.4 Integer Overflows

Although modern computers do not put too many constraints on the size of the numbers which are manipulated due to their available resources, embedded systems are more likely to optimise the size of the parameters; preferring a char over an integer could save memory. However, by enforcing this constraint, developers could create possible integer overflows vulnerabilities. Integer operations are done in a specific range depending on the number of bytes allocated by the system according to their type (e.g., unsigned char, signed int, and long). An integer overflow appears when an operation causes a value to go outside its allocated range. Depending on the system on which is executed the application, a simple modulo operation could be applied (which is the most common case), or the application could crash.

Integer overflows are more difficult to exploit than buffer overflows. However, they could be critical depending on the use of the result of the computations. For instance, if an integer is used to define the size of an array, an integer overflow could be used to define a smaller array than expected, giving the opportunity to an attacker to do a buffer overflow.

Identifying them is difficult since it depends on the type of the variables and the type of computations done by the application. Cast operations could be a source of some of these vulnerabilities and they are often detected by compiler. However, saving the multiplication of two char inside a char is more difficult to track, especially if a lot of steps have been applied.
2.3.2 Improper Use of Cryptography

Some techniques for encrypting, hashing, or managing private/public key could be not secure enough. For instance, using own implemented functions and not relying on common hashing functions like the Secure Hash Algorithm v2 (SHA-2) could be a vulnerability. Simple hash algorithms could be easy to crack. However, due to the constraints of embedded systems, it could lead developers to make some trade-offs between the hash algorithm level of security and the overall performances of the system.

As we underlined in the section about attacks, some collisions were found in some hash functions [16]. Another example of bad practice is the use of the Linux crypt(3) function with the default setting. The encryption is done with the DES algorithm on the first 8 bytes which could be cracked by brute-force.

Moreover, only applying hash functions to a password without a salt could be vulnerable to rainbow attacks. A salt is a predefined value coupled to the password during the hashing process to make the retrieval of the credentials harder.

2.3.3 Weak Access Control or Authentication

2.3.3.1 System-Level Attacks

Permissions Problems

Enforcing a security policy with multiple processes/users that can access and modify several resources according to their level of privilege could be a source of vulnerability. In fact, different kinds of vulnerabilities can appear in a multi-privilege system:

- Permission misconfiguration: An application was designed with some privileges, giving the ability to an attacker to distort the application to breach the security policy.
- Escalation of privilege: Vulnerability allowing an attacker to escalate privilege (access to a higher level of privilege).
- Shift of privilege: An attacker could use this vulnerability to be granted with another level of privilege which is not necessary higher than his. The goal of the attacker could be to access to some specific operations which can only be done by this set of users (or the root which is harder to access). This vulnerability underlines that attacks on privileges could also be tackled in a horizontal manner.

This class of vulnerabilities is often coupled with a buffer overflow. The buffer overflow could be used to inject code, whereas the permission vulnerability could grant the required level of privilege to execute this code.

Permitting Default or Weak Passwords

As underlined in [31], among the 10 million passwords analysed by Keeper Security, the top 25 passwords used in 2016 represented more than 50% of this database. Therefore, these passwords could be easily cracked with dictionary attacks.
This vulnerability seems obvious: weak passwords are easy to guess. However, this design issue is due to the difficulty for a user to remember a huge list of complex passwords or the choice of making the installation steps easier by providing default passwords. However, this kind of behaviour makes the task of password crackers easier. By mixing brute-force attacks, dictionary attacks, and rainbow tables, an attacker could easily retrieve a weak password and then access to the whole system.

Embedded systems are quite concerned by this family of vulnerabilities since they often require authentication of a user to be used. Moreover, to set up the embedded device, a default password is often provided.

**Dynamic Linking and Loading**

With the growth of the complexity of software, the reuse of COTS (Commercial Off-The-Shelf), and the extensibility design requirements are some of the reasons that lead software program not to be monolithic. They tend to be split between different modules and libraries. This architecture leads these programs to rely on libraries which could be static, shared, or dynamic. Dynamic library could be a source of vulnerabilities since programs call them at run-time. On the other hand, static ones are only called at the compilation time. Therefore, a modification of these dynamic libraries could be problematic.

The first issue could be to block the access to this library during the execution of the program. Depending on this implementation, the program could continue its execution without having executed the desired functions located in the library. If the functions which should have been called in the library are in charge of doing critical operations like access control management or encryption of private data, these operations will not be realised. In the case of an encryption library, data will be sent in plain text.

Another issue with dynamic libraries is their replacement by a malicious library. Since the dynamic library can inherit the privilege of the parent application, an attacker could leverage this level of privilege to execute malicious code. An attacker could retrieve the information fed by the users to the library.

Nevertheless, static libraries are not perfect; some security concerns can appear. If one static library is infected before the compilation, it leads the compiled binary to contain the malicious component of the library. Therefore, all the copies of the binary are infected. On the other hand, in the case of dynamic libraries, only the devices containing the malicious libraries are infected. The copies of the binary are not affected.

2.3.3.2 Information Disclosure

This family of vulnerabilities is related to letting an attacker retrieve information which is critical. They are often used to break the confidentiality principle of security. However, depending on the type of information which is accessed, for instance, passwords, the attacker could impact assets of the systems in various ways.
Storing Passwords in Plain Text or Weakly Encrypted

This class of vulnerabilities is linked to the vulnerabilities presented previously about authorising weak passwords or using weak encryption schemes. In this part, the issue is oriented towards the way of saving the password. Despite of requiring a password for using an application, if this password is stored in a plain text file, it is easy to bypass the authentication process. The design step dealing with passwords should not be under-evaluated. Letting passwords in clear in compiled code is not secured since an attacker can reverse engineering the code or analyse the memory to retrieve it.

Although hashing passwords appears as a solution to avoid storing them in plain text, we saw that the use of weak encryption schemes could lead an attacker to retrieve easily the plain text passwords.

Memory Leaking

During their execution, processes could require to save intermediate information in the memory. It could be at a low-level by directly writing a value in the memory or at a higher level like creating temporary files. Memory leaks could appear during the execution: it is possible to access to the data when the process is running, but also after the execution: the process has left data in the memory after finishing its execution.

Memory leaks could often occur when a variable was not deallocated or freed properly without zeroization. Even if the pointer to this memory location was lost and the memory is likely to overwrite this location soon, an attacker could try to retrieve the forgotten pieces of information before this overwriting process. To probe the memory for this data, the attacker could have to use specific tools with some high level of privileges. However, if he succeeds in retrieving data, he will be able to access to other protected areas of the system.

Although the creation of temporary files is not very specific to the area of embedded systems, this operation could happen in some cases with some OS’s. The same kinds of vulnerabilities may appear as previously if temporary files are not deleted properly. Moreover, an attacker could also rely on the fact that these files could appear in a file explorer. Coupled with a weakness of privilege management, it could help an attacker to exploit this vulnerability.

In addition, for most advanced embedded systems with a high-level OS, the management of swap files joined with incomplete deletes without zeroization provides similar opportunities to an attacker as temporary files. However, the tools used are different and it often requires the attacker to physically access to the memory.

2.4 Overview of Literature Review

The technological watch done on vulnerabilities allowed us to better understand the threats that can aim embedded software. It led us to underline the asymmetrical aspect of security in general where one weakness is enough to impact a whole system.
3 Toward Secure Embedded Software Development

Before dealing with the techniques that could be applied to protect and secure embedded software during their development, we will speak about some mechanisms which are at the basis of the securing techniques used to meet security requirements.

3.1 Basic Security Mechanisms

The goal of this section is to help the reader to understand what are the machineries behind. The two mains tools we will speak about are cryptography [32] and access controls [33].

3.1.1 Cryptography

The area of cryptography has been used to meet the security requirements (Confidentiality, Integrity, Availability, Authentication, and Non-repudiation) in the case of communications. One of the main application of cryptography is encryption for confidentiality. But it has also led to the development of mechanisms such as digital signatures and messages digestes. Cryptographic algorithms could be split between three classes: hash functions, symmetric encryption algorithms, and asymmetric encryption algorithms.

3.1.1.1 Hash Functions

The goal of hash algorithms is to provide a function whose outputs are easy to compute, but finding the input based on a given output is computationally hard: it is an efficient one-way function. Hash functions have the following properties:

- Pre-image resistance: For a given output $y$ of the hash-function, it is difficult to find a corresponding input $x$ such as $f(x) = y$.
- Second pre-image resistance: For a given couple of values $(x, y)$ such as $f(x) = y$, it is difficult to find another input $x'$ different from $x$ such as $f(x) = f(x') = y$.
- Collision resistance: It is hard to find a pair of two different inputs $x$ and $x'$ such as $f(x) = f(x')$.
- Avalanche effect: Although this property is not compulsory, it is desirable to ensure it. One small change in the input leads to completely different outputs. This property makes the task of an attacker harder to go through the output space.

Hash functions are often used for checking the integrity of data through checksum values. They are also used in the case of challenge-response protocols. Secure Hash Algorithm (SHA)-1, 2 and Message Digest v5 (MD5) are examples of common used hash algorithms.

3.1.1.2 Encryption Algorithms

The goal of an encryption algorithm is to transform a plain text into an unreadable text for everyone (i.e., data could not be understood) apart from the ones who have the good knowledge (i.e., a key). Decryption refers to the reverse
operation which consists in using the right knowledge to transform an encrypted text into the corresponding plain text. Encryption algorithms are based on Kerckhoffs’ principle which underlines that encryption algorithms should not be assumed secret, i.e., everyone could access to it. However, the key must be kept confidential and it should be impossible to retrieve information without knowing the key. This point leads to two kinds of design for the key: either symmetric, or asymmetric key.

Symmetric key algorithms used the same key for both encryption and decryption of the data. Therefore, anyone with the key could encrypt and decrypt data. Algorithms such as DES, 3DES, and AES are examples of symmetric encryption algorithms.

On the other hand, asymmetric algorithms do not use the same key to encrypt and decrypt data. The keys of this pair are often referred as the private (secret) and public (non-secret) keys. Data encrypted with the public key could only be decrypted with the private key and vice-versa. Usually, the way “encryption with public key-decryption with private key” is used to encrypt and send data to the owner of the private key, whereas the other way is used for signing and checking the signature of messages which are sent by the owner of the private key. This design allows an entity “A” to broadcast its public key to other entities and when one of these entities, called “B”, wants to communicate with “A”, this entity “B” will use the public key to encrypt the sent data. Then, the entity “A” will be the only one which can decrypt the data thanks to its private key. RSA and Diffie-Hellman are examples of asymmetric algorithms.

Although this asymmetric approach provides more security than a symmetric encryption, asymmetric encryption introduces a bigger overhead than symmetric one. In the case of embedded systems, it could impact the performance of the system. Moreover, the security provided by these algorithms also depends on the size of the keys which are used. Longer keys make the task of retrieving the data harder. However, as it was previously underlined, it introduces an overhead in comparison to smaller keys. Therefore, trade-offs about key sizes should often be made.

3.1.2 Access Control

This technique is used to manage the access, either to data or to operations (control flow), to only authorised users, processes, or programs. It is based on making the decision to grant or deny the access based on a security model and a set of permissions. Access control relies on policy models such as Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-Based Access Control (RBAC). These models are implemented through mechanisms like access control matrices, access control lists, or capabilities.

3.1.2.1 Access Control Policy Models

The control policies could be configured and managed through three main ways: either with Discretionary Access Control (DAC), Mandatory Access Control (MAC), or Role-Based Access Control (RBAC).

In the case of DAC, the access control is let to the discretion of the owner. This entity can set the control mechanisms for the objects it owns. This policy
model is one of the most used approach due to its simplicity. One common example is the case of Unix filesystems which is coupled to access control list.

On the other hand, MAC was introduced to provide a higher degree of security than the default DAC. The system administrator is in charge of defining the MAC policy which is enforced by the security part of the kernel or the OS. Moreover, end users cannot alter this provided access and cannot modify this policy. The idea is to block operations unless it has been explicitly authorised [40]. There is a notion of whitelisting. This feature requires additional work to configure it in comparison to a DAC. Security modules were developed based on this policy model such as SELinux, AppArmor, and GrSecurity.

RBAC, which is also referred as Non-Discretionary Access Control, is more related to the protection of a real-world organisation. A central administrator is in charge of defining roles and associating a set of permissions to these roles. Then, users are linked to these roles depending on their function. Decisions are made based on the role of the users.

### 3.1.2.2 ACL and Capability-Based Approach

Access control could be enforced through different control mechanisms. Two of the main used mechanisms are access control lists (ACL) and capability-based access control. In the ACL model, each resource has a list describing what are the operations which can be done by each entity. On the other hand, the capability approach defines for each entity what are its access rights. There is also a third approach which is easy to represent for a human being but which is not efficient in term of resources, it is the access control matrix where lines represent the resources, columns represent the entity, and each cell states the permissions for the related couple (entity, resource). The main problem of the last approach is the issue of sparse matrix.

In the case of a file system, Figure 8 represents the three previous cases. The table represents the access control matrix, the lines represent the ACL technique, and the columns show the capability approach.

![Figure 8](image)

**Figure 8.** Representation of Access Control Matrix, Access Control List, and Capability Approach

Although they could seem equivalent, ACL and capability approaches differ on some points which could lead to prefer one over the other one depending on the design constraints. For instance, capability approach can
allow to transfer the capability to another entity which is not the case of the ACL. On the other hand, the management of the revocation of capability is harder than the ACL which only consists in removing an element from the list. The Confused Deputy problem is a good example to show how capability could outperform access control lists.

3.2 Company’s Focus

There is not a unique solution to secure a full system to prevent an attacker from using previous vulnerabilities to impact a system. A lot of techniques have been developed to protect system and these techniques could be applied at different levels of the software life cycle. Some papers focus only on presenting in details one or two techniques to cope with a given vulnerability. Other papers deal with the way to increase the general level of security of systems. These techniques could range from design choices (e.g., isolating a process from others) to the use of specific tools (e.g., activating a memory protection in the compiler settings).

Since one of the goals of this internship is to higher the knowledge of the company about cybersecurity in the case of embedded software, it leads us to develop a document to present pieces of information related to this topic. The idea of this document is to gather guidelines which could be used by engineers in the company during projects. To create this document, we first looked at previous and related works.

3.3 Related works

Since security has become more and more important in computer sciences area in general, a lot of best practices, guidelines, principles, and techniques have been presented and described to higher the security of these systems.

3.3.1 General Software Approach

Firstly, a part of these documents consists of documents written by organisations to develop standards and to help companies to deal with security in computer systems. For instance, the National Institute of Standards and Technologies (NIST) provides a framework for taking into account security during the different steps of the System Development Life Cycle (SDLC) [34]. Although this document is quite complete, it is also quite complex and quite long, which could prevent engineers from spending time on its analysis. Non-US governmental organisations have been also involved in this process. For instance, the French national digital security agency (ANSSI: Agence Nationale de la Sécurité des Systèmes d’Information) also regularly publishes some guidelines about different topics (e.g., security recommendations related to GNU/Linux Systems). The level of details of these documents is variable (from few pages of principles to detailed solutions to implement/configure). One important point is the fact that the documents written by the ANSSI always start with a small table with a list of the targeted audiences such as developers, system administrators, and users.

Moreover, researches were carried out to formalise and analyse the way of introducing security guidelines. Peteanu presented in [35] a set of best
practices to ensure secure software development. He targeted developers as well as business analysts. The author presented his guidelines by organising them through main principles. In addition, the guidelines were accompanied by simple examples. Moreover, he also dealt in other parts with the security mechanisms that could be used and with web-related issues. The author pointed out the importance to integrate these security concerns in all the different steps of the software development life cycle. McGraw carried out a similar work in [36]. This book presents principles, guidelines, and techniques to apply to secure a software based on NIST’s work during the SDLC. In comparison to the NIST document, this book is useful by providing some use cases to better understand the theory. It also presents some already implemented solutions and discuss some issues which are left open. Wheeler also proposed a guide for secure programming in [37]. The organisation of this set of guidelines is different from the previous ones. One chapter is dedicated to the design of a secure architecture whereas other chapters are dedicated to specific vulnerabilities linked to implementation issues such as input validation and buffer overflows.

Yaşar et al. presented in [38] a survey of these three already developed best practices ([35]–[37]). This document only presents the list of best practices of these previous works in short lists; they mainly used the title of the chapters and sections of the books leading to provide a few details about what techniques to apply. The level of details is quite small. However, the goal of this work was to find possible improvements for dealing with best practices. Their main conclusion was about the integration of security concerns during software development: security must be spanned over the whole SDLC, it is not one step. Moreover, by going through these different sets of guidelines, it could be seen that they overlap at some points. For instance, one of the best practices given by Peteanu is “Interacting with Users”. This point could gather the principle of “Validate All Input” and “Secure the Interface” presented by Wheeler. The point from this example is the fact that there is not one unique set of guidelines that embraces the other ones.

Nevertheless, these guidelines were developed in the case of generic software development. Due to the growth of web solutions at the beginning of the third millennium, some authors also introduced guidelines which are specific to web development such as considering cross-site scripting. However, our work is aimed to deal with the case of embedded software. Due to the similarities of embedded software development life cycle with generic software development, some of these pieces of information could be reused or adapted with a minimum of overhead. Nevertheless, these sets of guidelines and best practices should be extended with the specific issues related to software development.

3.3.2 Embedded Software Focus

To cope with these generic approaches, some papers leveraged the knowledge from these works to directly apply it to the case of embedded systems. Ravi et al., in [39] and Khelladi et al., in [40], introduced the challenges that could arise when designing embedded systems. A part of their presentation is based on McGraw’s researches. Then, the partitioning of the
different techniques depending on the steps of the SDLC is reused. In these two papers, the authors described, with more or less details, some possible techniques. Moreover, they did not focus only on the software, they underlined the challenges related to embedded systems such as power consumption and tamper-resistance mechanisms.

They did not limit their work to present these challenges, Ravi et al. also carried out experiments such as doing Differential Power Analysis (DPA), whereas Khelladi et al. measured the energy consumption of different encryption algorithms for several embedded processors. Therefore, it led this paper to tackle the design and architectural challenges, and then, propose a survey of some of the techniques that could be encountered with different levels of details.

One important piece of information which is reused in these different papers is the graphical representation of the links between best practices and the steps of the SDLC in which they are used. This representation first appeared in a paper of McGraw [41] which was reused in his book. Figure 9 represents this organisation. Moreover, the NIST document about integrating security in software development also use the SDLC to order their idea.

![Figure 9. Link between securing techniques and their corresponding SDLC step(s) ](image)

In addition to these formal papers, a lot of short lists about applying security concepts and principles could be found in a lot of places on the Internet. This short lists often list the main principles of security such as “Apply least privilege principle” or “Compartment your program”. Although these sets of guidelines allow to get an idea of what is security, they are very short summaries of all the work done by security researchers and could be too vague to be apply in a project.
One good example of this kind of lists is the “Embedded Top 10 Best Practices” provided by OWASP [8]:
1. Buffer and Stack Overflow Protection
2. Injection Prevention
3. Firmware Updates and Cryptographic Signatures
4. Securing Sensitive Information
5. Identity Management
6. Embedded Framework and C-Based Toolchain Hardening
7. Usage of Debugging Code and Interfaces
8. Transport Layer Security-Enhanced
9. Usage of Data Collection and Storage – Privacy
10. Third Party Code and Component

We could see that the list is just a general checklist that gives starting points to investigate. In addition, OWASP provides explanation and examples with these best practices. However, this way of presenting security concerns could be seen as too light in the case of a company project.

3.3.3 Learnings from previous work

One of the first conclusion from this study of these previous works is the fact that the concept of Software Development Life Cycle is at the core of the integration of security. Security must be integrated at each step to enhance the general security level of a system.

In addition, we could see from these previous works that presenting security concerns in the case of software, and more specifically, in the case of embedded software, strongly depends on the targeted audience: from general developers to advanced security scientists, the level of details can vary widely. Pieces of information presented to general audience is a short summary of all the research papers that were addressing the research community.

Due to the level of details, it could lead engineers to an overhead in understanding the guidelines and best practices. If there are not enough details such as “Compartment your application”, the engineers should investigate first all the possible solutions or ideas to compartment their application. Some possible solutions could be to rely on virtualisation solutions or to apply partitioning in the design of the submodules of the application. These investigations could take time for a software engineer. On the other hand, a paper describing in details all the features of hardware virtualisation could be time-consuming to understand it in comparison to the value it will bring. It is important to keep in mind that projects are time-boxed and the time invested in these investigations should be well-defined.
The aim of this thesis is to provide a way of increasing the knowledge of company’s engineers about the cybersecurity in the case of embedded systems. This goal is split between different tasks. The first task is to develop a document that could be used by these engineers during their project to ensure a secure development. Based on this work, the company would like to apply this set of guidelines and best practices to one of their embedded system project for one of their client. Based on this analysis, some recommendations and POCs (Proof-Of-Concept) will be made.

Therefore, the work of this thesis is split into two parts: writing a document to help the development of secure embedded software and carrying out an analysis on a use case. These two parts were done with different methodologies.

4.1 Knowledge enhancement Document Writing

4.1.1 Method

The part about the redaction of the document followed a qualitative research method: we worked on gathering knowledge and theories from literature to build a new knowledge for the company. The first step of this work was to gather pieces of information from the literature we used in the previous parts. Therefore, we carried out a conceptual research because we did an analysis from gathered data of literature reviews. Our research methodology relied on grounded theory because we attended to develop a theory to follow for the company. To organise these gathered data, we performed interviews with engineers in the company. The idea was to define a first version of this document and to refine it through the feedback of the engineers. Since, the number of engineers in the company is not so important and since the goal of these feedbacks is to receive possible enhancements of the document, using quantitative approach was not recommended, we relied mostly on qualitative questions. These interviews were semi-structured around a predefined questionnaire which contained open-questions.

4.1.2 Quality Assurance

One of the main quality assurance concerns to ensure in this first work was the transferability: the document could be reused as a new starting point for the engineers to carry out research about specific issues of cybersecurity in the case of embedded software. This concern was addressed through the result of the interviews. We had to check the good understanding of the readers. The validity, another point of the quality concerns, was addressed through the second part of this work.

Since interviews were conducted, we had to take care about ethics. Especially about the questions concerning the interviewees. These questions should not disclose personal or critical data.
4.2 Analysis of vulnerability and countermeasures

4.2.1 Method

On the other hand, we carried out a security audit in the case of a project of one of the company’s client by relying on the document we wrote. This step was planned to allow us to validate the quality of our work. Beside the recommendations that could appear from this analysis, some POCs were implemented. To do this work, we followed a quantitative research method. It allows us to verify the functionality of the systems. One important point during the design of the systems was the fact that the functional requirements should be SMART: Specific, Measurable, Attainable, Realistic, and Time-Bound. The most important aspect in our case was the “Measurable” one which confirmed our choice of quantitative method. We used an experimental research method because we analysed the performance of the system regarding different aspects (from computation speed to security requirement). We carried out experiments by coding POCs and analysing the results.

4.2.2 Quality Assurance

As far as the quality assurance was concerned, we had to emphasize the assurance of the replicability aspect since the idea is to reuse these POCs in the case of other projects. Therefore, we had to document all the hardware and software configurations we used to set up this experiment. Validity was ensured through well-described protocols with trusted measuring tools.
5 Knowledge Enhancement Document Creation

5.1 Process Description

Making a document, or other materials (e.g., it could be a video), that increases and enhances the knowledge of an audience on a specific topic is not limited as summarizing existing data. Value must be added by organising and reformulating the data depending on the different aspects (e.g., goals, use cases, and readers).

To create this document, we followed the methodology we described previously. More precisely, we followed this process:

1. Define formally the environment: goals, topics, audience, etc.
2. Gather data from the literature reviews.
3. Organise and reformulate the data.
4. Perform interviews and get feedback.
5. Update document (e.g., organisation, content, reformulation) based on the analysis of the results obtained from 4.

5.2 Scope of the guidelines

This part aims to define the points we had to think about when we wrote our guidelines. We had to start by defining the scope of the paper by answering to general questions such as:
- What will be the content of the document? (Content)
- Why this document is written? (Goal)
- Who will use it? (Target)
- ...

These pieces of information were retrieved by looking at the requirements about the internship and through the informal interview with my supervisor.

5.2.1 Content

The content of this guide must present security techniques, guidelines, best practices, and principles which should be applied or followed during the development of embedded software.

5.2.2 Goal

This guide aims to increase the knowledge of company’s engineers specialised in embedded systems about security for embedded software and to be used during project as a kind of checklist.

5.2.3 Audience

The guide targets embedded systems engineers of the company. Since I was integrated to the teams of engineers, I could extract additional features based on my own non-intrusive observations. The following points help to better define the target:
- Different age groups (i.e., from junior engineers to senior ones);
- Different experiences;
- Different career paths;
- Different roles in project (e.g., business analysts, architects, implementers, and testers);
- Embedded systems background;
- No background about security (apart from a few of them);
- No specific field of applications of embedded systems (Could work for a medical project or an energy management company);
- Work in a digital and connected environment;
- Could work in a team or alone;
- Could work within the company or within the client company.

These points were useful to define the form of the guide. By taking them into account, it improved the understanding of the targeted audience. We saw in the part about related work how the form of a document could be modified depending on the audience.

5.2.4 Form

This part deals with the way the guide conveys information to the audience based on the previous points.

Numerical text-based document

Due to the environment, the guide should be a numerical text-based document. It addresses the fact that the engineers are used to work with this kind of documents at the different steps of the project. It makes the sharing, the access, the search, the update, and the traceability of the document easier.

Short Document

Since using the document should be convenient and should not lead to a time-overhead in the project, a short document allows engineers to use it without wasting too much time. The previous works were often quite long (e.g., standards of more than one hundred pages and books of several hundreds of pages). Asking every participant to go through this documentation during a project could cost money to the company through the invested time.

A balance between the time required to go through this document and the security improvements provided by using these guidelines should be found. In addition, one difficulty was to measure “short” because a document of ten pages is shorter than a document of twenty pages. However, if the document of twenty pages has well-organised sections of four pages, it could be faster as expected to find a piece of information than in the document of ten pages.

Organised Structure

Although only listing security techniques without structure could be used in short lists, in the case of longer ones, it could make the reading of the document more difficult. By structuring the guidelines, it could help engineers to go through the document faster. Moreover, engineers are more likely to remember structured ideas than an unordered list of items.
Based on our literature review about embedded software vulnerabilities presented in Chapter 2, it led us to think to build our structure around these vulnerabilities. However, when investigating techniques to higher the security level of an application, we encountered some of them that were not directly aiming to solve a specific vulnerability. Therefore, relying on this structure could have led to repeat some guidelines or to forget some of them. We extracted from the analysis of related works that most of the guidelines followed the different steps of the software development life cycle. Since it was an already used structure in the industry to define guidelines, we decided to follow this organisation based on the different development steps. Another advantage of this structure is to provide a specific section for each sub-category; the architect can directly focus on the part about system design whereas developers can directly go to the part about the security techniques to apply during the programming process.

Although the work of McGraw presented a partitioning of the software development life cycle (Requirements and Uses cases, Design, Test plans, Code, Tests results, Field feedback), we relied on a more standard terminology. The IEEE 1074 is a Standard for Developing Life Cycle Processes. It defines the set of activities and processes which must be addressed for developing and maintaining a software. Therefore, it includes the case of embedded software. Table 2 presents the different process groups and their related processes [42]. We used this terminology to define the different sections and sub-sections that organises our document. We did not reuse all the terms presented in this table because some processes did not contain security guidelines.
Table 2. Software Process introduced in IEEE 1074

<table>
<thead>
<tr>
<th>Process Group</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Life Cycle Modelling</td>
<td>Selection of a Life Cycle Model</td>
</tr>
<tr>
<td>Project Management</td>
<td>Project Initiation</td>
</tr>
<tr>
<td></td>
<td>Project Monitoring and Control</td>
</tr>
<tr>
<td></td>
<td>Software Quality Management</td>
</tr>
<tr>
<td></td>
<td>Concept Exploration</td>
</tr>
<tr>
<td></td>
<td>System Allocation</td>
</tr>
<tr>
<td>Development</td>
<td>Requirements</td>
</tr>
<tr>
<td></td>
<td>Design</td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
</tr>
<tr>
<td>Post-Development</td>
<td>Installation</td>
</tr>
<tr>
<td></td>
<td>Operation and Support</td>
</tr>
<tr>
<td></td>
<td>Maintenance</td>
</tr>
<tr>
<td></td>
<td>Retirement</td>
</tr>
<tr>
<td>Integral</td>
<td>Verification and Validation</td>
</tr>
<tr>
<td></td>
<td>Software Configuration Management</td>
</tr>
<tr>
<td></td>
<td>Documentation Development</td>
</tr>
<tr>
<td></td>
<td>Training</td>
</tr>
</tbody>
</table>

**Language**

Although the audience are French engineers, the use of English in most of the papers related to computer sciences and the international aspect of this thesis, the document should be written in English. At the end, a French translation could be provided.

**Speech Format**

Since this document provide pieces of advice, we could use the format “Verb (Imperative form) + Complement + (Examples)”. An example of this format gives “Use access control mechanisms (e.g., MAC and Capability-based mechanisms)”. We could also use the format “You should + Verb + Complement+ (Examples)”. However, this is longer than directly writing the action to do through imperative form. Moreover, depending on the guidelines, it could not be possible to provide examples.

**Speech**

Since our goal is to convey information, we should take care of the way we are using the language. A low-quality language could lead to
misunderstandings or could require additional time for the reader to understand the ideas. In order to provide a good quality, we could follow the guidelines provided by Georges Orwell in “Politics and the English Language” [43] which are:

1. Never use a metaphor, simile, or other figure of speech which you are used to seeing in print.
2. Never use a long word where a short one will do.
3. If it is possible to cut a word out, always cut it out.
4. Never use the passive where you can use the active.
5. Never use a foreign phrase, a scientific word, or a jargon word if you can think of an everyday English equivalent.
6. Break any of these rules sooner than say anything outright barbarous.

5.3 Building the draft

To build our document, we followed this process:

1. We listed all the principles, guidelines, best principles, techniques, mechanisms, etc. we could extract from the literature we had used so far. Although our literature review was aimed to identify vulnerabilities in the case of embedded systems, the papers which were presenting these vulnerabilities were not limited to only introducing the vulnerabilities. They also presented some counter-measures or security principles.
2. We extended our literature based on the references present inside these documents. At the end, all the pieces of information came from academic papers, organisation papers, and books which could be retrieved on online database (e.g., IEEE Xplore Digital Library, ACM Digital Library, Springer Link, and Google Scholar Search Engine). At this stage, no data was retrieved from other sources.
3. We gathered pieces of information together depending on the software development life cycle they were related to.
4. Similar pieces of information were merged and summarized.
5. We performed an iterative process to reformulate each guideline and organise them.

At the end of this process, we obtained a first draft of our set of guidelines.

5.4 Improving our Document.

This first draft could not be directly used by the company. It should be improved through an iterative process. To do so, we planned to perform semi-structured interviews with engineers at the company. This approach allowed us to ask open questions. In opposition to closed questions, this technique allows to get more than the opinions of the interviewee, we could retrieve valuable pieces of advice. However, closed questions were used to quantify some points (such as the features which we should improve first). We decided to perform interviews by following a structure to ensure that we went through the different aspects we wanted to improve. Moreover, by following this structure, it allowed us to bound the time of the interviews on the contrary to fully unstructured interviews.
Due to these decisions, we had to set up interview questions based on the aspects we wanted to evaluate and improve. As the set of guidelines, we planned to update this structure depending on the proceedings of the interviews. We reused previous interviews to enhance the next ones.

As far as the interviews’ content is concerned, we first retrieved data about the interviewees to verify the consistency with our targeted audience. We payed attention to the issues related to ethics. Then, we used the generic hourglass “shape” to conduct these interviews. We first asked about the generic impression and the layout. Then, we dealt with the introduction and the structure of the document. It led us to speak about the content by analysing the understandings of the interviewee and his learnings. Then, we focused on a part on which the interviewee was the most interested in. We branched out the discussion on possible improvements.

The structure we used for the interviews is presented in Appendix A. Interview Structure. We only did one interview with one engineer of the company. Although we planned to do more interviews, Bureau Veritas published in mid-June (five weeks before the end of the thesis) a document similar to the one we were working on called “Cybersecurity Guidelines for Software Development & Assessment” [44]. Due to this event, we decided to compare this document with our existing guidelines to improve them.

The next paragraphs present the result of the interview and the analysis of the document of Bureau Veritas.

5.4.1 Interview’s Feedback Summary

This interview was performed on a raw set of guidelines, the format and layout were not done yet at this stage. The interviewee was a developer, which had also provided pieces of advice about architecture choices in his career. He did not have a security background. He had followed some trainings about generic cybersecurity issues. He underlines the following points:

- The document seemed quite complete and tends to be exhaustive.
- The preface was welcoming and user-friendly.
- The accumulation of imperative was a bit aggressive in opposition to the introduction.
- A glossary for the acronyms was missing.
- Relying on the Software Development Life Cycle was a good idea for organising the guidelines.
- Some guidelines should be enhanced with explanation about their goal or the idea behind.
- He learned things by reading this document and he planned to reuse some guidelines in his next projects.
Therefore, we decided to perform the following enhancements:

- Add opening sentences at the beginning of each sub-section to seem less aggressive and to explain the how and the why of this subset of guidelines.
- Add a glossary at the end.
- Improve the guidelines that the engineers had marked as unclear.

5.4.2 Bureau Veritas’ Document Analysis

Due to the publication of this document, we changed our way of working and decided to compare it to our existing guidelines to see their strengths and their weaknesses. Although at the time of the analysis our document was not finished (e.g., some remarks had not been integrated yet and the document did not follow an official template), the content of our guidelines was defined, which was the most important point to compare with the guidelines written by Bureau Veritas.

The document written by Bureau Veritas and our guidelines have a lot of similarities such as the goal of the document, its audience, and the way to organise the idea. In addition, we were able to see that our current guidelines are consistent in regards to the document of Bureau Veritas. There were no contradictions between their objectives and acceptance criteria and our guidelines.

However, we saw that they made different choices about the structure of the document such as integrating examples in appendices. Although these appendices are quite helpful to give concrete examples to the reader about some security aspects, we decided not to integrate this kind of appendix in our guidelines. After investigating the available resources, we saw that a reader could easily access to many examples for different techniques and tools. Therefore, we focused on using keywords that could allow the reader to retrieve these examples. In addition, there is also the issue of knowing where to draw the line: if we put this example, should we also integrate this example? Our choice was to not integrate example.

From this analysis, we saw that our guidelines have a higher granularity than the ones presented in the document of Bureau Veritas. Nevertheless, we pointed out that some points of the objectives of this document were not presented (or were not sufficiently explicit) in our guidelines. Therefore, it allowed us to enhance our set of guidelines.

A part of the acceptance criteria presented in the objectives OBJ_DES_030, OBJ_DES_70, OBJ_DES_60, and OBJ_DES_80 allowed us to enhance our guidelines about the security of the software tools used in the project. A part of the objective OBJ_CHE_010 and OBJ_CHE_020 led us to introduce the keyword “CWE” and “CVE” that we had forgotten to introduce in our document despite having dealt with them in our literature review. Finally, we highlighted the concept of regression testing in our guidelines thanks to the objective OBJ_OPE_20.

Another interesting point is the formalism about the numbering of the guidelines used in the document of Veritas. In the case of our guidelines, we only introduced a simple numbering with the format G_XXX. We realised later, during the improvement steps, that this formalism was precarious. Adding or
removing a guideline impacts all the following ones. In addition, we saw that the formalism used by Veritas allows the reader to easily identify the part of the Software Development Life Cycle which is targeted by this objective. For instance, all the objectives OBJ_DES_XXX deal with design objectives. Our numbering does not provide this feature. Therefore, a future work could be to enhance this formalism.

We did not think about the formalism of the referencing at the beginning of the redaction because we were not planning to distribute it. However, at the end of the internship, some managers were interested to share this document to other entities of the company. It was at this moment that we saw that the formalism should be improved to better support modification and referencing.

5.5 Final Document
The final document is not presented in this thesis due to its confidentiality. However, we presented a way of writing a guidance document. At the end, we presented 169 guidelines dealing with all the steps of the Software Development Life Cycle over 19 pages (images included). Then, we met our requirements.

The formalism of the referencing system will have to be improved in the next version. This point was not addressed during the thesis since we pointed out this problem at the end of the project.

To evaluate the consistency of our document, in addition to the interview and the analysis of the documents of Bureau Veritas, we decided to apply it to a real project.
6 Security Analysis of Client’s Project

To check the consistency of our guidelines, we decided to use them in the case of a real project of one of the company’s client which is currently under development. We analysed the documents which describe the requirements and the architecture of the project. Due to the complexity of this project, we did not dive into the system design, the development, and the testing steps of the software. Then, the analysis of the guidelines related to the implementation, testing, and deployment were less in-depth.

The company provided us the three following documents:
- Applicative Software Architecture Document (rev 2);
- Sub-System Specification Applicative Software (rev 3);
- Commercial Linux distribution evaluation.

From this analysis, we wrote a set of recommendations to improve the security features of the project. If it was possible, we presented a use case or we developed a proof-of-concept or a tool related to this recommendation. However, due to the confidentiality of the project, we will not present the full analysis of the project. We will only present recommendations which were identified as non-critical.

Then, this chapter is organised as follow: after having presented these non-confidential recommendations, we will introduce the different analyses we did and the tools we developed to increase the security.

As far as our guidelines are concerned, we did not encounter contradictions between them and the specifications, i.e., no specification was telling to perform a non-recommended action from our guidelines to increase the security of the systems. This point was important to confirm the consistency of our guidelines. Then, during our analysis:
- Either we found a specification which matches this guideline. Then, we could confirm the consistency of this guideline.
- Or we found a specification which addresses partially or not a guideline. Then, if we had enough information, we wrote a recommendation.

6.1 Recommendations

From this analysis, we decided to give possible recommendations for the project. However, we had to consider that the project had already started when we did this analysis. Therefore, it was useless to give pieces of advice related to some design choices such as hardware, implementation languages, or some libraries. On one hand, some of these choices were required by the client and could not be changed. On the other hand, due to the tight relation between design choices, time, and cost (i.e., the longer we wait to make a change, the more expensive this change is likely to be), modifying one of this aspect could lead to a complete refactoring of the project, which is not possible. We gave the following recommendations:
- Security analyses should be done. They should tackle the different features of the project.
  - Risk assessment and threat modelling of the system and its environment.
Security analysis of the different COTS/libraries used in the project.

Security analysis with a focus on the external communication. The result of these analyses should be tracked and should lead to identify the most critical parts of the system regarding security by quantifying and rating the results.

- Set up a policy for the access to the communication mean, especially D-Bus.
- The access control mechanisms (e.g., DAC, MAC, and Capability-based) that enforce the different policies should be defined.
- Resources limitations (e.g., CPU and memory usage) for the different processes should be implemented in the software.
- A precise list of the different cryptographic mechanisms used in the project should be written. This list should also contain the parameters, such as key length, of the mechanisms, and their impact on the system.
- An analysis of the data life cycle, especially the persistent data and its end of life, should be carried out to highlight the most critical points related to its management.
- The project documentation should list the different tools and modules related to protection mechanisms and exploit mitigation techniques integrated in the architecture.
- Some static and dynamic tools related to the evolution of the project should be integrated.

From the set of recommendations, we worked on the following points:

- We used one security aspect to present an example of a security analysis.
- We developed a benchmarking solution to analyse some cryptographic parameters.
- We developed a tool to track the evolution of the size of the binaries of the project.
- We developed a tool to record the performances of the system under test (useful for resource analysis).
- We carried out an analysis of some security mechanisms provided by D-Bus.

The following sections present with more details these previous points. Since securing D-Bus was a main part of the thesis, we dedicated a full chapter, Chapter 7, to this work which consisted of an analysis of the features of D-Bus and the implementation of a use case.

Moreover, another intern worked on integrating in the continuous integration environment a tool to compute and display the information related to the coverage provided by the unit tests.

6.2 Example of Security Analysis: Attack Trees

We worked on a simple example of threat modelling analysis which focused on the access to critical data and which was not clearly described in the
specification documents. The project deals with an embedded device which interact with human beings.

To do our analysis, we relied on attack tree approach. First, we took the attacker’s viewpoint. Since the device contains private user data, these data are identified as critical. Therefore, these data must be kept confidential. Attacker may want to access to them.

From the document, we could learn that these data are stored in logs and in persistent data. Logs and persistent data can be accessed through external communication through a specific device or the HMI (Human Machine Interface). The data are encrypted in the device storage with the AES algorithm. Moreover, an anonymization function can be applied when the device is moved to an area defined as unsecure, such as the area for firmware update or maintenance. Figure 10 presents the resulting attack tree.

![Attack Tree Diagram](image)

Figure 10. Sketch of an attack tree dealing with the case of accessing critical data

We can see that accessing the log menu is quite easy since the HMI should be ergonomic.

This analysis remains superficial since it is only based on the information that could be retrieved and inferred from the provided documents but it could be a starting point for deeper analyses.

### 6.3 Performance Analysis of AES

The case of cryptography is addressed through the fact that no cryptographic library is needed. This choice was most likely done because Linux already has a cryptographic module. Although there is no specific need, the fact of stating it shows the importance of this component in the design choices. AES was chosen with a key length of 128 bits.
We did a benchmark analysis on a similar board to see the overhead that could be introduced by using longer keys. The goal of this performance analysis was to show the trade-off between performance and security which could arise in the case of embedded systems. Another goal of this kind of analysis was to give an example of the skeleton of a quick analysis which could be integrated in the design choice document to explain why some decisions were made.

This study aimed to find a trade-off between having a high level of security (i.e., the key is long) and having good performances (i.e., the encryption algorithm is fast – its encryption key is small).

First, we looked at the security provided by the key length by computing the time required for an attacker to brute force this encryption mechanism. By assuming that the attacker could access to the fastest computer which is currently existing [45]: The National Supercomputing Centre in Wuxi, China: Sunway TaihuLight – Sunway MPP, Sunway SW26010 260C 1,45 GHZ, Sunway NRPC. It could execute instructions at around 93 Pflops/s. We assumed that it requires 1000 operations to try one combination [46].

Table 3 gathers the different parameters used to represent the attacker’s resources. Table 4 gathers the time required to go through all the space of combinations of the AES encryption mechanisms depending on the key length.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Speed</td>
<td>93 Pflops/s</td>
</tr>
<tr>
<td>Operations per combination</td>
<td>1000 operations</td>
</tr>
<tr>
<td>Conversion sec ↔ year</td>
<td>31557600 s/year</td>
</tr>
</tbody>
</table>

Table 4. Time to brute force AES depending on the key length

<table>
<thead>
<tr>
<th>Key Length (bits)</th>
<th>Number of combination</th>
<th>Time (years)</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>3.40E+38</td>
<td>1.16E+17</td>
</tr>
<tr>
<td>192</td>
<td>6.28E+57</td>
<td>2.14E+36</td>
</tr>
<tr>
<td>258</td>
<td>4.63E+77</td>
<td>1.58E+56</td>
</tr>
</tbody>
</table>

This first analysis shows the security aspect of the AES encryption mechanisms. A key length of 128 bits seems sufficient, particularly for a product which requires a long-term support of ten years.

Now, we will look at its performance. We run the encryption/decryption algorithms on an Atmel SAMA5D3 Xplained Board on two files. Table 9 in Appendix B presents the board specifications.

As far as the file are concerned, we used a small one of 3 kB and a bigger one of 3 MB. These sizes were chosen to be representative of the log files that will be used on the device. Since user data could have around 12 fields (e.g., ID, name, and gender), by overestimating the size of each field to 256 Bytes, it leads to have a user data of 3 kB. In addition, according to the specifications, the log should be able to store up to 10 000 history entries. By assuming a history entry is around 256 Bytes, then with the 10 000 history entries, we could overestimate the size of a log file to 3 MB.
We relied on the AES mechanism provided by the module OpenSSL and on the command `perf stat` to gather the results. The script which was executed on the embedded board is presented in Table 5. We run 100 times the same command of encryption and decryption. The results of the analysis are gathered in Table 6. Figure 11 and Figure 12 present these results with bar charts.
Table 6. Time to encrypt/decrypt different files with AES depending on their size and the key length of the cryptographic mechanisms (values in parentheses represent the standard deviation)

<table>
<thead>
<tr>
<th>Log Size</th>
<th>Technique</th>
<th>128 bits</th>
<th>192 bits</th>
<th>256 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Log (3 kB)</td>
<td>Encryption</td>
<td>58,2 (0,66%)</td>
<td>57,4 (0,86%)</td>
<td>56,0 (0,98%)</td>
</tr>
<tr>
<td></td>
<td>Decryption</td>
<td>53,3 (1,16%)</td>
<td>52,0 (1,16%)</td>
<td>53,3 (1,22%)</td>
</tr>
<tr>
<td>Big Log (3 MB)</td>
<td>Encryption</td>
<td>636,2 (0,16%)</td>
<td>677,3 (0,13%)</td>
<td>716,7 (0,15%)</td>
</tr>
<tr>
<td></td>
<td>Decryption</td>
<td>797,4 (0,12%)</td>
<td>833,2 (0,12%)</td>
<td>872,2 (0,11%)</td>
</tr>
</tbody>
</table>

Figure 11. Encryption/Decryption time of AES protocol depending on key length for a log of 3kB
Firstly, from Table 6 and Figure 11, we can see that standard deviation is too big to see the differences of performance for encryption/decrypting of a small log. However, as far the big log is concerned, Figure 12 emphasizes how performances decrease with the growth of the key length. Doubling the length of the key (from 128 bits to 256 bits) leads to an overhead of 12.6% for encryption and an overhead of 9.4% in the case of decryption. Therefore, increasing the length of the key in the case of small files seems not to lead to decrease the performance of the system. Security could be increased without impacting performance. However, in the case of bigger files, bigger key lengths lead to slow down computations. In regard with the time required to crack the AES algorithm and with the field of application of the product, it seems that there is no point in relying on keys longer than 128 bits to encrypt and decrypt files.

### 6.4 Static Metrics Monitoring Tool

#### 6.4.1 Project's need

The project uses a continuous integration tool: Jenkins [47]. This tool oversees the retrieving of the sources from a GitHub server, the building of the binaries from these sources, the management of the merge requests, and the evaluation of the unit tests. Our recommendation about monitoring the complexity of the project met a need of the client’s project. They wanted to integrate in this environment a tool to monitor the size of the binaries built from the sources.
When the sources are retrieved from the Git, they are compiled for the targeted device. It leads to the creation of a set of binaries. The idea is to track that we do not incorporate a security flaw such as an overuse of the memory between two versions of the project.

6.4.2 Tool Principle

The goal of the tool is to measure some metrics of the generated binaries each time a new version is released. In addition, these measurements should be saved to present their evolution. We decided to measure the two-following metrics:

- Section size (i.e., text, data, bss, and total)
- Size of the binaries

The former could be retrieved from the `size(1)` Linux command whereas the latter could be computed through the `stat(1)` Linux command.

The saving process is done via a shell script which is run on a slave machine controlled by the Jenkins server. The outputs of the previous commands are saved in text format. Then, a Python script parses these files to generate tables and save them as CSV files. After this process, the Python script oversees the computations of some statistics (e.g., max and min), the generation of some graphs, and the generation of a HTML report.

In addition, since Jenkins allows the automation of the job with a pipeline approach, we created a dependency between our job and the job in charge of compiling the binaries: each time the build of the binaries is a success, our job is launched on the compiled binaries.

We used the NumPy (http://www.numpy.org/) and Plotly (https://plot.ly/python/) Python libraries in our Python script. Moreover, we relied on the HTML Publisher Plugin of Jenkins to display our report. Figure 13 represents the different steps of our Jenkins job and the external libraries/plugins which were used.
At the end of process, the engineers can access to a HTML report which gathers:
- Tables generated from the parsing of the output of the commands;
- Graphs generated from the previous tables;
- Computed statistics over the whole set of versions of the project.

Moreover, the tables and the graphs represent the data in two different ways:
- Binary-Centric: The evolution of the metrics depending on the version of the project for one binary.
- Metric-Centric: The evolution of one metric depending on the version of the project for all the binaries.

Figure 14 gives an example of one of the graph which is generated thanks to the Plotly library.
Figure 14. Example of a binary-centric graph: evolution of the metrics depending on the versions of the project for one of the process

6.5 Dynamic Metrics Analyser

6.5.1 Project’s need

As we underlined in our recommendations, it is important to monitor the resources consumption during execution to detect as soon as possible malicious or unexpected behaviour. The project had already asked an intern in another location to do log management and log analysis thanks to advance tools such as ElasticSearch and Kibana to do preventive maintenance and behaviour analysis.

However, due to the complexity of the solutions, and since the test campaign was coming, they asked for a quick and reliable solution to record and analyse some dynamic metrics. One of the main constraints was to directly deploy and run the tool on the embedded device, which led us to:

- Use the tools already installed on the custom Linux distribution, i.e., we cannot install new ones;
- Minimise the performance impact;
- Make the tools simple to use, i.e., it should not require a long training.

6.5.2 Tool Principle

To meet the previous constraints, we split our tool in two parts. A first part oversees the recording of the metrics of the test sessions. Then, the other part runs the analysis of this recording to make them understandable by the engineers. Thanks to this split approach, it allowed us to rely on more powerful tools for the analysis since we could do it on a general-purpose computer.

6.5.2.1 Embedded Metrics Recording

Since we cannot modify the built custom embedded Linux distribution of the embedded board, we had to rely on basic tools provided by the Linux kernel. We decided to use the Linux top command which display the running
tasks of the systems. Normally, this command provides a dynamic view of the
tasks and their resources consumption. But, it is also possible to use it in batch
mode (i.e., it sends the output of the command to a file) thanks to the command
line options “-b”. In addition, the sampling time could be parametrised through
the command line options “-d”.

The choice of the sampling time is quite important since it could lead to
a performance overhead which could impact the other processes. In addition,
the smaller the interval time is, the faster the log file grows. This aspect is quite
important because, in the case of an embedded device, the memory is limited.

Therefore, we estimated the size of each sampling of the top command.
Firstly, to minimise the output of the top command, we piped (i.e., use of Linux
pipe “|”) its output to the Linux commands grep and awk. We planned to record
eleven processes for each sample, which represents eleven lines. By
overestimating the size of a line to 60 Bytes (due to our formatting), it leads to
a size of 660 Bytes per sample. According to the testers, a test session last 20
minutes on average. Then, for a sampling rate of one second, it leads to a log
file smaller than 1 MB (around 792 kB), which could be supported by the board.
We did not reduce the sampling rate because the other intern concluded that it
was sufficient. However, it could be interesting to carry out a deep analysis to
find the Pareto optimum between accuracy and performance overhead. Since
the goal of this tool was to be a simple solution, we did not deepen our analysis.

As far as the recorded metrics are concerned, we only kept the CPU and
memory usage provided by the top command. We leveraged the “-p” command
line option to only get the processes which interests us, i.e., the processes of the
application.

6.5.2.2 Unembedded Analysis

As we underlined, the analysis is done on a general-purpose computer.
We could use advanced tools to perform this analysis. Since we had already
developed the static metrics monitoring tool, we decided to reuse the
knowledge we got from this part for this tool. We reused a Python script to parse
the data. However, since we use the awk command to format the output of the
top command, the parsing was easier.

Then, we reused the NumPy Python library to manage the data and the
Plotly Python library to generate graph. In addition, we took advantage of the
Pandas Python library (http://pandas.pydata.org/) to generate HTML tables
for our HTML report. In opposition to the static metrics monitoring tool in
which we wrote our own HTML templates, we did not write ones for this tool
because its goal was to provide a quick solution before the tools of the other
intern were released. Then, we did not format the outputs and we used the
default format provided by the Pandas library.

6.5.2.3 Full Process Working

The tool which was developed worked by applying consecutively the two
previous parts. Figure 15 represents the full process of the use of the tool and
underlines the separation between the embedded and unembedded parts.
6.5.3 Results

We deployed our tool by sending it to some testers with a small README. The testers succeeded in setting the environment, recording metrics, and analysing them. Therefore, our tool met the requirement about the ease of use. Moreover, we looked at the size of the log file which were generated during the test sessions. As expected, their size remains below the 500 kB.

Figure 16 shows one of the graphs which are generated by the Python script. Only two graphs are generated, one for the CPU usage and one for the memory usage.
As far as the performance overhead is concerned, we leverage the top command to record its own resources consumption as well the full pipe with the grep and awk command. By looking at the result, we saw that the overhead was negligible (around one percent of CPU and memory usage).

6.5.4 Possible Improvements

Since the goal of the development of this tool was to provide a temporary working solution while the more advanced tools were in development, the resulting tool was sufficient. However, we saw that some improvements could be made to enhance it.

Firstly, we could improve the parsing script to be more fault-tolerant. In our Python script, we assumed a specific format for the end of the logs file. However, the redirection of the standard output stdout could be stopped in the midst of a line writing, leading the log file to end with an uncomplete line. This point could be quite easy to integrate in our Python script.

Another possible improvement concerns the graph. As we can see in Figure 16, the line has a saw-tooth shape. It is due to the recording of the metrics by the top command. It could be interesting to smooth the graph by applying an average filter over a few seconds.

6.6 Preliminary Discussions

The realisation these analyses, proof-of-concepts, and tools helped us to improve the security of the project. Thank to our small threat modelling, we
provided an example of a good security practice: integrating security analyses in the review. This simple presentation was sufficient to give a possible way of proceeding. The implementation of the benchmarking solution of the AES algorithm was not limited to provide the company with a performance analysis of AES in regards to the security. We developed a solution which could be easily reused in other projects to quickly analyse the performances of an embedded device regarding its cryptographic libraries. This kind of solution could help the designers team in making decisions about integrating or not a hardware component dedicated to the encryption. Finally, although the main goal of the tools we developed was to provide monitoring solutions, they increased the security of the project by auditing the complexity and the performances of the system which could be useful to detect a malicious behaviour.

The next part of our work was dedicated to analyse the security of an IPC used in the project: D-Bus. In comparison to the previous analyses, this one was more extensive.
7 D-Bus Security Analysis

7.1 Interest

The security analysis we performed on the architecture documents of the project, thanks to our guidelines, led us to identify a critical element: D-Bus. This element is used to communicate between the processes of the application. The product has already integrated security mechanisms in the processes (at the application/presentation layers of the communication) such as encryption or integrity check mechanisms. However, the engineers have not investigated the possibility of securing D-Bus. From the documentations, it seems possible to configure a policy for a bus. Therefore, the goal of our task was to investigate the possibilities provided by D-Bus to secure it. After having presented the main building blocks of D-Bus, we will present the policy configuration of D-Bus based on the documentation of D-Bus [48]. Then, we will present a use case of this application of this policy. This use case will lead us to analyse the security, but also the performance, of the system.

7.2 D-Bus Principles

7.2.1 General Presentation

D-Bus is an Inter-Process Communication (IPC) mechanism. It provides a mean of communication between processes executed on the same host. An application can register to one or several buses. This application can provide services on this bus to other applications, but they can also use the services provided by this bus. D-Bus was created in 2002 and it is a part of the freedesktop.org project. It is currently maintained by RedHat.

D-Bus implementation provides different tools:

- A library, called libdbus, which allows the applications to communicate.
- A daemon, called dbus-daemon, which oversees the creation of the mean of communication. It also oversees the routing of the messages. This daemon is based on libdbus.
- Some bindings and wrappers which allows developers to use different languages. These bindings are based on a low-level API written in C.

D-Bus is a non-transactional mechanism, i.e., it does not ensure the ACID (Atomicity, Consistency, Isolation, and Durability) properties as requested by a transactional system. D-Bus is connection-based and stateful, i.e., it can track and keep in the memory the connections. In addition, this mechanism processes messages as discrete elements. D-Bus is a binary protocol, the carried data is non-textual which avoids the problem of parsing and serialisation. Applications can connect to the bus via a socket.

D-Bus protocol provides two types of bus with their own configuration. The first one is the system bus. On a desktop, there is only one system bus for all users and it is used for system services and low-level events (e.g., kernel events, network connection notifications, and USB (de)connection). On smaller systems, such as embedded systems, it could be the only available bus. The second type of bus is the session bus. This bus is used inside a user session. It is linked to a Xsession. Therefore, it is created at the opening of the user session.
and it is deleted when the user closes his session. One main difference between these two buses is their default security policy, the policy of the system bus is more protective than the one of the session bus. A bus instance is identified by an address which is defined by the system configuration.

7.2.2 Building Blocks

D-Bus is built around different components: services, objects, interfaces, and clients. A service is composed of object(s) which implement(s) interfaces. Clients can use the services registered to the bus.

7.2.2.1 Connections and Services

Before exposing services to a D-Bus instance, or using them, a process must connect to this D-Bus instance. A connection is represented with one or more names which are called bus names. When the process connects to D-Bus, the daemon is in charge of assigning a unique name which could not be reassigned during the whole life cycle of this D-Bus instance, even if it is the same process which does a de-connection/reconnection. In addition, it is possible to request a well-known name which is more human readable. Well-known names are used to offer services to D-Bus. Like unique connection name, there could be only one well-known name at a time. However, these names could be reused after being dropped.

7.2.2.2 Objects

They are linked to one service and uniquely identified thanks to an object path. Client processes use objects to expose what they can provide to the bus. Other client processes access to objects through proxies, which are references to these objects. They are used as local representations of the objects inside the client processes. Objects can contain interface(s).

7.2.2.3 Interfaces

Interfaces are also identified with unique names and they contain members (i.e., properties, methods, and signals).

7.2.2.4 Properties

It is a set of fields of the interface which could be directly accessed in read and write mode. It supports different types from the basic ones (e.g., integer, Boolean, and bytes) to more advanced ones (e.g., string-like types and containers types).

7.2.2.5 Methods

They can be used by remote processes to execute an exposed method by a process. Remote processes can feed parameters as inputs. The process executing this method can return the result of the method. Then, it is like calling a method in any programming languages, however, the method is executed remotely by another process.
7.2.2.6 **Signals**

They are used as notification and are unidirectional. D-Bus transmits the signal to the clients that registered beforehand to it. Signals can contain parameters which could be retrieved by the clients which listen to it.

7.2.2.7 **Messages**

Messages could be related to signals or methods. In the case of the signal, messages contain the signal and its parameters. On the other hand, as far as methods are concerned, messages could be either a method call (i.e., a client does a remote method call with some parameters), or a method reply (i.e., the server replies to the client and sends the output of the method executed with the parameters sent by the client). In addition, messages could also be error messages in case of exception.

7.2.2.8 **Setting up everything together**

Table 7 summarizes the different components of a D-Bus instance and their naming convention. Figure 17 gives a graphical representation of the organisation of these elements.

<table>
<thead>
<tr>
<th>Component</th>
<th>Identifier</th>
<th>Identifier format</th>
<th>Defined by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>Address</td>
<td>unix:abstract=/tmp/dbus-fu4nohtEEA</td>
<td>System Configuration</td>
</tr>
<tr>
<td>Connection/</td>
<td>Bus name</td>
<td>:1.16 (unique) or com.company.MyService (well-known)</td>
<td>D-Bus (unique) or the owning program (well-known).</td>
</tr>
<tr>
<td>Services</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Path</td>
<td>/com/company/MyService/MyService</td>
<td>The owning program</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>Interface</td>
<td>com.company.MyService.ServiceInterface</td>
<td>The owning program</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Member</td>
<td>Member</td>
<td>myMethod</td>
<td>The owning program</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.3 D-Bus Configuration

7.3.1 Configuration File

When a D-Bus instance is created, it requires a configuration file describing basic properties of the bus. This configuration file uses a format called `busconfig` which is based on XML. The system and session buses have default configuration files respectively called `system.conf` and `session.conf` which are presented in Appendix C. The configuration file can allow a user to fix resource limits and define security parameters. It is also possible to include links to other configuration files or folders to provide a high modularity. A documentation of the configuration file could be found in the documentation of `dbus-daemon` [49]. We will focus on the `<policy>` element of the configuration file. The next part is dedicated to providing a summary of the main parameters of the policy.

7.3.2 Policy Configuration

In the configuration file, it is possible to define a security policy for the bus inside the `<policy>` tags. The policy is built with `<allow>` and `<deny>` tags. It is similar to a firewall policy. As underlined in the documentation and in our guidelines, the policy should be restrictive as much as possible. The idea is to deny everything, and then, to allow a small set of messages. It is like a whitelist: operations are permitted if and only if they are explicitly written in the policy. The `<policy>` can have one of the following attributes:

- `context = "(default|mandatory)"`
• \textit{at\_console} = \texttt{"(true|false)"
• \textit{user} = \texttt{"username of userid"}
• \textit{group} = \texttt{"group name or gid"

These attributes allow administrator to define different policies inside a same file which will be applied in a specific order. According to the documentation, the policies are applied in this order:

1. All \textit{context} policies in \texttt{"default"}
2. All \textit{group} policies (undefined order)
3. All \textit{user} policies (undefined order)
4. All \textit{at\_console} policies in \texttt{"true"}
5. All \textit{at\_console} policies in \texttt{"false"}
6. All \textit{context} policies in \texttt{"mandatory"

In case of a contradiction between two rules, the last encountered rule overrides the one encountered earlier. On the other hand, policies with the same attribute are applied according to their order of apparition in the file. User can leverage this sequencing to override a former rule with a new one.

Each \textit{<allow>/deny>} tag contains parameters to do the filtering. \textit{<deny>} tags are used to prohibit an action whereas \textit{<allow>} ones bypass the prohibition of previous \textit{<deny>} tags depending on their attributes. The available attributes of these tags are the followings

- \textit{send\_interface/receive\_interface} = \texttt{“interface\_name”}
- \textit{send\_member/receive\_member} = \texttt{“method\_or\_signal\_name”}
- \textit{send\_error/receive\_error} = \texttt{“error\_name”}
- \textit{send\_destination/receive\_sender} = \texttt{“name”}
- \textit{send\_type/receive\_type} = \texttt{“method\_call”|“method\_return”|“signal”|“error”}
- \textit{send\_path/receive\_path} = \texttt{“/path/name”}
- \textit{send\_requested\_reply} = \texttt{“true”|“false”}
- \textit{receive\_requested\_reply} = \texttt{“true”|“false”}
- \textit{eavesdrop} = \texttt{“true”|“false”}
- \textit{own} = \texttt{“name”}
- \textit{own\_prefix} = \texttt{“name”}
- \textit{user} = \texttt{“username”}
- \textit{group} = \texttt{“groupname”}

The \textit{send\_destination} and \textit{receive\_sender} attributes filter the owner of the given name, it does not apply the filter to that \texttt{“name”}. The other \textit{receive\_} and \textit{send\_} attributes are only textual filters which are applied to the message header. When a rule is written, \textit{receive\_} and \textit{send\_} attributes cannot be put together in the same rule.

The \textit{own} attribute enables administrator to authorise or not a connection to own a well-known name. It is also possible to use \textit{own\_prefix} which only filters the prefix of the well-known name. For instance, with the rule \textit{<allow own\_prefix=“com.myCompany”>}, we allow the connection to own

The user and group attributes enable administrator to authorise or not a user or a group to connect to the bus. These attributes can only be used in the rules included inside context or at_console policies. It is not meaningful to include them inside a user or group policy.

A rule matches if all the combined attributes match the related parameters of the messages, i.e., the combined attributes are related with a logical AND. To get a logical OR, several independent rules should be written.

We did not investigate the case of the other attributes send_requested_reply, receive_requested_reply, and eavesdrop. The last one is often used for the case of debugging. Therefore, we leverage the most stringent parameters which does not block the system from working.

7.4 Use Case: Implementing a policy

From the reading of the documentation, we thought that the implementation of a D-Bus policy consists in blocking everything and then listing one-by-one the different messages and connections which should be authorised. This approach allows to respect our guidelines about minimising the access and using whitelisting. However, by implementing a use case, we saw that the implementation of the D-Bus policy was not so straightforward.

To implement a policy, we developed a small use case which represents, in a simplified way, some features of the real architecture of the project on which we performed our security analysis.

7.4.1 Client’s Project Background

D-Bus is used in the project to allow the processes to communicate. Nine processes are connected to the session instance of D-Bus. The configuration file which is used for this instance of D-Bus is the default session.conf file provided with D-Bus. The processes use both methods and signals. Some relationships could be represented as Client-Server relations: a client calls a remote method from the server and wait for the output of the computation. Although the project uses both synchronous and asynchronous calls, we limited ourselves to the case of synchronous calls. In addition, some processes use signals.

As far as the implementation is concerned, the project uses a C++ library API for D-Bus: DBus-c++ 0.9.0. This installation could be done via the package libdbus-c++-dev.

7.4.2 Use case implementation

To represent the real architecture without having an architecture too complex, we worked with the following system:

- **Watchdog**: It emits a signal called heartbeat at a periodic time.
- **Logger**: This process offers a method called saveLog which allows other processes to write to a log file which is managed by this process.
- **Server**: It provides two methods: add and mul, which take two integers and return respectively the sum and the product of the two integers. Server also emits a signal nbOfCount which gives the number of computations done since Server has been started. This signal is emitted
every time Server has made ten operations. Server also listens to the signal heartbeat which is emitted by Watchdog. Moreover, it logs all its operation by calling the saveLog method exposed by Logger.

- **Client**: It does not provide any service to the bus. However, it calls alternatively the add and mul methods of Server and it listens to the signal nbOfCount of Server. As Server, Client logs all its operations by calling the saveLog method exposed by Logger.

This system is quite interesting since it allows us to represent several features of the real architecture. In the project, one of the process must send a signal at a periodic time with the state of one of the component of the system. This behaviour is represented through Watchdog and Server. In addition, a lot of the relations between the processes are similar to a client–server architecture. Therefore, our processes Client and Server, with the methods add and mul, represent these relations. Finally, we added Logger which manages the log because the real architecture has a similar process which writes in the filesystem.

However, the system we described before is a perfect system with no attacker. Since our goal is to check the enforcement of our policy, we added to this system malicious processes. We added the two following processes:

- **SpyA**: This process tries to own a name and it provides a method called methodNoise which takes an integer as input and returns this integer incremented by one. It also emits a signal called signalNoise which carries a string. This process also tries to use the add and mul methods exposed by Server and it listens to the signal nbOfCount of the Server.

- **SpyB**: This process tries to also own a name. It tries to call the method of SpyA and to listen to the signal signalNoise of SpyA.

This architecture allows us to look at different attacks. SpyA tries to integrate itself in the system, use some methods, and listen to a signal which are exposed on this D-Bus instance. On the other hand, the client–server architecture between SpyB and SpyA allows us to investigate the possibility of using the bus without authorisation. This kind of use could introduce an overhead leading to decrease the available resources for the primary system (i.e., it is a kind of DoS attacks). We did not investigate spoofing attacks since the specification underlines the unicity of the names. However, it could be possible to do some spoofing by killing a process and reusing its well-known name.

Figure 18 summarizes the different processes, their member, and their relations. As far as the naming of their service, their interface, and their object path is concerned, we used the following template where $ should be replaced by the name of the process (i.e., Logger, Watchdog, Server, Client, SpyA, and SpyB):

```
SERVICE_NAME = "com.thales.SimpleSystem.$"
OBJECT_PATH = "/com/thales/SimpleSystem/$/$Object"
INTERFACE = "com.thales.SimpleSystem.$.ServiceInterface"
```
7.4.3 Policy Definition

7.4.3.1 Starting from Default Policies

In the case of the project, they use the default `session.conf` configuration file with its policy. The content of the file is presented in Appendix C. This policy allows everything and it is not secure at all. In order to secure our system, we decided to look at the policy provided by the default `system.conf` and to use it as a starting point to write our own policy. The policy could be found in Appendix C. We will go through the different rules to analyse them and see if we can harden the policy.

Before looking to the different rules, we can see that the system configuration file leverages the policy attributes `context="default"` and `user="root"` to give more privilege to the root user.

<allow user="*"/>

In the case of our system, since we know the specific set of users, we could directly deny all users except the one which will run the application. In addition, for monitoring purpose, the user `root` could be authorised.

<deny own="*"/>

This rule blocks any process to claim a well-known name and to register a service. Then, this rule cannot be hardened. However, we will have to add later...
a policy which lists, as a whitelist, the well-known names which could be registered.

<deny send_type="method_call">
This rule blocks any method calls. As the previous rule, it cannot be hardened but we will have to add additional rules for the method calls we will authorise.

<allow send_type="signal">
Since signals are unidirectional, the policy of the system configuration file does not see the point in blocking them. However, we decided to harden this rule by denying signals and only allowing the signals we defined in our architecture.

<allow send_requested_reply="true" send_type="method_return"/>
The default configuration of the system bus allows this kind of message since there is no point in filtering them. These messages are replies to method calls, the messages carrying the method calls have already gone through the filter. So, if there is a method return, it means that the method was authorised. Therefore, we did not harden this rule.

<allow send_requested_reply="true" send_type="error"/>
Error messages are used as replies if an exception is raised. Therefore, as previously, it is better to allow this kind of message. In addition, these messages are managed by the bus itself. Denying them could lead to some problems for the bus.

<-- All messages may be received by default --> (4 allow rules)
At the beginning, we tried to harden the rules related to the received method call and signal. However, we encountered some problems due to D-Bus implementation which cannot allow us to define rules with the receive_* attribute. Therefore, these four rules were unmodified. We will discuss later about this limitation with the receive_* attributes.

<-- Allow anyone to talk to the message bus --> (2 allow rules)
<-- But disallow some specific bus services --> (3 deny rules)
With the two allows rules, the context="default" policy allows applications to access to some D-Bus methods but the three deny rules block the most critical part. Since our system does not need to communicate with the message bus itself during its execution (apart from the registration phase), we denied the access to these interfaces except the one required for the registration. However, their access could be allowed in later rules specific to the root user for monitoring purpose.

<policy user="root"> (3 allow rules)
Then, the policy defines three additional rules specific to the user root which are related to monitoring or reporting. Since our system is planned to
run as a simple user, we removed all these rules. However, we created a new configuration policy file called `monitorPolicy.conf` with the following content:

```
<!-- Only root can talk to the message bus -->
<policy user="root">
  <allow send_destination="org.freedesktop.DBus" send_interface="org.freedesktop.DBus"/>
  <allow send_destination="org.freedesktop.DBus" send_interface="org.freedesktop.DBus.Introspectable"/>
  <allow send_destination="org.freedesktop.DBus" send_interface="org.freedesktop.systemd1.Activator"/>
  <allow send_destination="org.freedesktop.DBus" send_interface="org.freedesktop.DBus.Monitoring"/>
  <allow send_destination="org.freedesktop.DBus" send_interface="org.freedesktop.DBus.Debug.Stats"/>
</policy>
```

This policy is defined for the user `root`. Its goal is to provide this user an access to the monitoring tool `dbus-monitor`. We integrated this policy in the main configuration file with the tag:

```
<include>monitorPolicy.conf</include>
```

This approach allows developers to remove or integrate a specific policy easily.

### 7.4.3.2 Enhancing the Policy

If we launch our system with the previous policy, it will not work due to the `<deny>` rules. Therefore, the next step was to add rules which allow our system to work but which are not too permissive.

Firstly, we had to define our policy for the user who will run the application and which was authorised in the default policy. In our case, the user is named `clement`. Therefore, we created a policy specific to this user:

```
<policy user="clement">
  <allow own="com.thales.SimpleSystem.Logger"/>
  <allow own="com.thales.SimpleSystem.Server"/>
  <allow own="com.thales.SimpleSystem.Client"/>
  <allow own="com.thales.SimpleSystem.Watchdog"/>
</policy>
```

When the process wants to connect and register to D-Bus, it will send a method call for the method `RequestName` and `AddMacth` provided by the interface `"org.freedesktop.Dbus"`. To allow this registration step, we must add the two following rules:

```
<allow send_destination="org.freedesktop.DBus" send_interface="org.freedesktop.DBus">
  <send_member="RequestName" send_type="method_call"/>
</allow>
<allow send_destination="org.freedesktop.DBus" send_interface="org.freedesktop.DBus">
  <send_member="AddMatch" send_type="method_call"/>
</allow>
```

Then, we should allow our system to own a well-known name. Based on the parameters defined in our code, it leads us to add the following rules:

```
<allow own="com.thales.SimpleSystem.Logger"/>
<allow own="com.thales.SimpleSystem.Server"/>
<allow own="com.thales.SimpleSystem.Client"/>
<allow own="com.thales.SimpleSystem.Watchdog"/>
```

At this step, we thought about leveraging the attribute `own_prefix` since all of our services start with `"com.thales.SimpleSystem"`. Thanks to this attribute, we could merge our four previous rules into one rule. However, this approach does not respect our guideline about minimising the authorisations. By using the `own_prefix` attribute, `SpyA` could register its service to D-Bus by using the well-known name `"com.thales.SimpleSystem.SpyA"`. Therefore, the `own_prefix` attribute should not be used due to its permissiveness.
Since we denied the method call, we must write <allow> rules for messages which call a method. In our system, we should only be able to call the **add** and **mul** methods of **Server**, and the **saveLog** method of **Logger**. Therefore, it leads to the following rules:

```
<allow>
  send_destination="com.thales.SimpleSystem.Server"
  send_path="/com/thales/SimpleSystem/Server/ServerObject"
  send_member="add"
  send_type="method_call"
</allow>

<allow>
  send_destination="com.thales.SimpleSystem.Server"
  send_path="/com/thales/SimpleSystem/Server/ServerObject"
  send_member="mul"
  send_type="method_call"
</allow>

<allow>
  send_destination="com.thales.SimpleSystem.Logger"
  send_path="/com/thales/SimpleSystem/Logger/LoggerObject"
  send_interface="com.thales.SimpleSystem.Logger.ServiceInterface"
  send_member="saveLog"
  send_type="method_call"
</allow>
```

As we underlined in the summary of the documentation of the <policy> tag, combined attributes in a same rule use logical AND. This point should be used to harden the filtering. The more explicit we are about the parameters, the more restrictive our system is.

Signals are different from method calls because they are unidirectional and they do not have a destination. To understand the structure of these messages, we used the monitoring tool of D-Bus called **dbus-monitor**. This command allows users to see the messages which are exchanged on a specific D-Bus instance. Thanks to this tool, we could see that the destination of a signal is “(null destination)”. However, the path, interface and member are non-null. Therefore, to apply a filter on signal, we should not use the send_destination attributes, leading to the following rules:

```
<allow>
  send_path="/com/thales/SimpleSystem/Server/ServerObject"
  send_member="signalNbOp"
  send_type="signal"
</allow>

<allow>
  send_path="/com/thales/SimpleSystem/Watchdog/WatchdogObject"
  send_member="heartbeat"
  send_type="signal"
</allow>
```

As we underlined before, we did not write <deny> rules with receive_ attributes. At the first glance, we thought that it was sufficient to use the same
approach as the send_* attributes. However, by using dbus-monitor, we saw that our messages were blocked. By looking at the messages of the method calls, we saw that these messages, as far as the sender’s side is concerned, only contain the sender’s name. In addition, the sender name is the unique name which is assigned by the daemon, not the well-known name.

We looked at the C++ binding to see if it was possible to assign by ourselves this unique ID. The binding was using the functions of the low-level C API for D-Bus. In the documentation of these functions, it was recommended not to use them since they require some synchronisation inside a single threat due to the need of assigning a unique ID. Therefore, we decided not to modify the bindings, firstly, due to the problems that could appear, but also that modifying library/COTS without retesting them could compromise the security of the system as we pointed out in the guidelines.

A first idea was to look at a first execution of our system, retrieve the unique name, and update the policy at run-time. However, when a D-Bus instance is created, it loads at the beginning a configuration file which will be used for the whole life cycle of this instance. Therefore, it is useless to update the configuration file after the start of the D-Bus instance. Some low-level functions of the C API of D-Bus could modify the policy. However, the development team of D-Bus do not recommend relying on this API to use D-Bus due to its complexity. Our idea of looking at several executions of the systems led us to see that the allocation of the IDs by the daemon was done deterministically by incrementing a counter. All the IDs have the form :1.X where X is an integer which is incremented by one for each new connection attempt. By launching the different processes in a specific order with a sleep time between each of them, it was possible to guess the value of the IDs which were assigned by the daemon. The registrations should work at the first try. In case of failure, the daemon also increases the counter.

Therefore, due to the dependency, we launched our system in the following order: Logger, Server, Client, Watchdog, SpyA, and SpyB. It led to the allocation of the following unique names: Logger(:1.0), Server(:1.1), Client (:1.2), Watchdog (:1.3). As far as our spies are concerned, it is a bit different because they try in the first case to request a well-known name which is not authorised by the policy, which leads to increase the counter. Then, SpyA is assigned to the unique name :1.5 (:1.4 failed), and SpyB is assigned to the unique name :1.7 (:1.6 failed).

Thanks to this approach, we could rewrite a policy which takes into account the sender. We updated the default policy with the following rule:

```
<deny receive_type="method_call"/>
```

And we added to the policy linked to the user `client` the following rules:

```
<allow receive_sender=":1.1" receive_type="method_call"/>
<allow receive_sender=":1.2" receive_type="method_call"/>
```

In the case of the previous unique name allocation, the first rule allows Server to make method calls, whereas the second rule authorises Client to make method calls. No other process can make method calls.

This point led us to think about the case of the signals. The part of the signal messages related to the sender are also limited to its unique name.
Therefore, the same approach could be used. However, adding this rule does not seem very useful since the send_* attributes of the signal (i.e., send_interface, send_path, and send_member) are related the attributes of the sender because signals do not have a destination. As we underlined previously, adding <deny> rules is better for the security of the system. However, it requires the administrator to know in advance the unique names of the processes.

As we underlined in the part about the documentation, rules cannot combine receive_* and send_* attributes since they are processed independently. Therefore, it prevents an administrator from setting up a policy which filters point-to-point communication. If we have two instances A and B which expose methods, and an instance C which must call some methods of A and whose we know the unique name, then, in the policy, we will allow:

- Method calls to the destination A (with its related path, interfaces, and members).
- Method calls to the destination B (with its related path, interfaces, and members).
- Method calls from the sender C.

Nevertheless, this policy would not block method calls from C to B. There is not an isolation of the point-to-point communications.

7.4.3.3 Policy Test

In order to test our policy, we run our system with the different policies: default session policy, improved policy which allows receive_type="method_call", and the same improved policy but which uses the unique names to filter the receive_type="method_call". The processes were executed in the same order as we described previously to guess their unique name.

For the three policies, the core system (i.e., Logger, Server, Client, and Watchdog) worked properly, we did not receive any error. Therefore, we did not restrict too much our system. The differences can be seen with the malicious system (i.e., SpyA and SpyB). Firstly, thanks to the filter on the owning names, we encountered the following messages in the logs:

```
SpyA: Name Request and Connection to DBus Failure
Error name: org.freedesktop.DBus.Error.AccessDenied
Error message: org.freedesktop.DBus.Error.AccessDenied
What: Connection ":1.4" is not allowed to own the service "com.thales.SimpleSystem.SpyA" due to security policies in the configuration file
```

```
SpyB: Name Request and Connection to DBus Failure
Error name: org.freedesktop.DBus.Error.AccessDenied
Error message: org.freedesktop.DBus.Error.AccessDenied
What: Connection ":1.6" is not allowed to own the service "com.thales.SimpleSystem.SpyB" due to security policies in the configuration file
```

These messages prove that we blocked the request, but not the connection.
Then, thanks to the filter on the service registration, SpyB was not able to access to the method of SpyA. We got the following error message from the process SpyB:

```
SpyB: Ask SpyA for methodNoise: 7
SpyB: Method call failure
Error name: org.freedesktop.DBus.Error.ServiceUnknown
Error message: org.freedesktop.DBus.Error.ServiceUnknown
What: The name com.thales.SimpleSystem.SpyA was not provided by any .service files
```

In addition, the rules related to the signals allowed us to block the emission of the signalNoise of SpyA. However, we did not get an error neither in the process SpyA nor in the process SpyB like the previous DBusError. Since our filter was applied to the send_* attributes, the error was only visible in the bus itself. By using `dbus-monitor`, we saw the following sequences of messages:

```
signal time=1499064549.459196 sender=:1.5 -> destination=(null destination) serial=14
path=/com/thales/SimpleSystem/SpyA/SpyAObject;
interface=com.thales.SimpleSystem.SpyA.ServiceInterface; member=signalNoise
  string "SpyA sends signal after 1"
error time=1499064549.459337 sender=org.freedesktop.DBus-> destination=:1.5
error name=org.freedesktop.DBus.Error.AccessDenied reply_serial=14
  string "Rejected send message, 1 matched rules; type="signal", sender=":1.5"(uid=1000 pid=4921
                comm="SpyA/a.out")interface="com.thales.SimpleSystem.SpyA.ServiceInterface"
  member="signalNoise" error name= "(unset)" requested_reply="0"
  destination="org.freedesktop.DBus"(uid=1000 pid=4925 comm="SpyB/a.out")"
```

This point is quite interesting since the malicious process is not informed of this error: we minimise the quantity of information returned.

The use of the most advanced policy allowed us to block the method calls emerging from SpyA, which was not the case with the other policies. We saw the following messages in SpyA:

```
SpyA: Ask Server for: 3*3
SpyA: Calling Server Failure:Ask Server for: 3*3
Error name: org.freedesktop.DBus.Error.AccessDenied
Error message: org.freedesktop.DBus.Error.AccessDenied
What: Rejected receive message, 1 matched rules; type="method_call", sender=":1.5"(uid=1000
            member="mul" error name= "(unset)" requested_reply="0"
            destination="com.thales.SimpleSystem.Server"(uid=1000 pid=3137 comm="Binaries/Server.out")
```

### 7.5 Performance Analysis

Since D-Bus is run on an embedded device, we decided to analyse the performance of the system on a board to look at the computational overhead. Embedded systems differ from general-purpose computers by these kinds of constraints. In addition, we built our own Linux operating system to focus only on D-Bus.

#### 7.5.1 Methodology

##### 7.5.1.1 System

To analyse the computational overhead, we only launched the core system of our use case, we did not launch the two spies. We did not integrate the malicious part of the system because it would lead to a bias when comparing the default session policy with our own policy. Since the default policy does not
block the messages from the spies, it would lead to additional computations in the other processes, such as Server. In the case of the default session policy, Server would have to do the computations asked by SpyA, whereas in the case of our own policy, Server would not have to do the computations required by SpyA since D-Bus would have blocked the method call beforehand.

7.5.1.2 Policy

We worked with the default session policy and the policy we implemented which does not integrate the filtering on the receive_* attributes. We did not integrate the one with the filtering on the receive_* attributes due to its precariousness.

To examine how the complexity of the policy could increase the overhead, we generated additional policies. These policies were based on our policy. We just appended at the end of the rules which are inside the tags <policy user="clement"> </policy> additional rules to simulate more detailed policies. We added the following rules:

```
<deny send_destination="com.thales.SimpleSystem.X"
send_interface="com.thales.SimpleSystem.X.ServiceInterface"/>
```

Where X is a number between 1 and 1000 leading to have distinct rules.

We generated five policies with 50, 100, 250, 500, and 1000 additional rules where X goes from 1 to the number of additional rules. For instance, the policy 250 consists of our own policy and a list of the previous rules where X goes from 1 to 250.

Therefore, we run our analysis on seven policies.

7.5.1.3 Board

The analysis was done on an Atmel SAMA5D3 Xplained evaluation kit we used for the AES Analysis. The board specifications are presented in Table 9 of Appendix B. We built our own Linux operating system to only focus on D-Bus and limit the noise that could come from other processes such as the daemon launched by the operating system at the start. To build it, we used Buildroot [50]. We based our operating system on the default configuration files provided by Buildroot for this board called atmel_sama5d3_xplained_mmc_defconfig. We modified the default configuration file sama5_defconfig of the kernel provided by the Buildroot default configuration files, which is linux4sama5_5, to allow perf events thanks to the parameter:

```
CONFIG_PERF_EVENT=y
```

As far as the Buildroot configuration file is concerned, we activated the perf tool of the kernel, dbus, and dbus-c++. We also enabled WCHAR support and C++ support as recommended by the tutorial of free-electrons [51].

After having built our embedded distribution, we created, in addition to the already existing root user, a user called clement with his own work space.
7.5.1.4 Performance Tool

As underlined in the board configuration, we relied on the perf tool [52] provided by the Linux Kernel. We launched our core system with its own D-Bus instance thanks to the command dbus-run-session. We leveraged the command line parameter “--config-file” of this command to parametrise the policy of this D-Bus instance. After having launched our system, we launched the following command:

$ perf record -o PerfRecord/perf.data -a sleep 600

It corresponds to a system-wide collection from all CPUs (in the case of our board, there is only one CPU) during ten minutes inside the file perf.data. The header of the perf report which gives the different parameters of the data collection is presented in Appendix D. This way, we were able through the command perf report to analyse the results on our computer.

The perf report command gives access to the value of a metric called Overhead. This metric is obtained by summing all the values of the periods of one entry. In the case of our analysis, the entries were the command names. We retrieved them by using the command line option “--sort=comm” of the perf report command. This sorting parameter fetches the name of the command of the related task by reading it via /proc/<pid>/comm. The sum of the different overheads is equal to 100%. Nevertheless, when analysing the call chains and considering the child functions, the sum of the overhead could exceed 100% because of the dependency. In the case of our analysis, we did not consider the call chains.

Then, the value of the overhead is not meaningful as such. It is a relative metric. It becomes valuable by comparing it to the overheads of similar systems.

As far as the possible bias of the perf tool is concerned, we leveraged the tool itself to check that it was not introducing a significant overhead.

7.5.2 Results

The results of the different executions are gathered in Table 8. As expected, the more rules a policy has, the bigger the overhead is. We can see that our own policy led to a small overhead in comparison to the default session policy. Figure 19 represents the relation between the number of additional rules and the resulting overhead (the default session policy is not represented). We can see that the dependency seems to be linear.
Table 8. Computational overhead measure by *perf* tool during the execution of the core system depending on the policy

<table>
<thead>
<tr>
<th>Policy</th>
<th>Computational Overhead of D-Bus Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Session Policy</td>
<td>33.51%</td>
</tr>
<tr>
<td>Own Policy (0 additional rule)</td>
<td>34.68%</td>
</tr>
<tr>
<td>Own Policy (50 additional rules)</td>
<td>36.54%</td>
</tr>
<tr>
<td>Own Policy (100 additional rules)</td>
<td>38.26%</td>
</tr>
<tr>
<td>Own Policy (250 additional rules)</td>
<td>44.52%</td>
</tr>
<tr>
<td>Own Policy (500 additional rules)</td>
<td>50.87%</td>
</tr>
<tr>
<td>Own Policy (1000 additional rules)</td>
<td>61.38%</td>
</tr>
</tbody>
</table>

As far as the overhead of the *perf* tool is concerned, the reports measured that its overhead was between 0.24% and 0.35%. Therefore, we can conclude that it did not impact too much our results.

### 7.5.3 Discussion

This linear dependency could have been expected by looking at the core implementation of the D-Bus daemon. In the C implementation, in the `policy.c` file, it can be seen that the rules are checked by doing a loop over the list of rules for each message.

We could see that the relation is not totally proportional and, for a bigger number of rules, the overhead increases less than the coefficient of proportionality of the policy with a smaller number of rules (less than 250). This point could be due to several factors.
• It could be explained by some cache or compiler optimisations for the loop.
• It could come from a saturation of D-Bus due to the frequency of the messages.
• Another possible factor is the deviation. We executed our system only once for each policy. However, the Linux kernel is non-deterministic. Then two executions of the same application could lead D-Bus to process the messages in different orders. In addition, although we built a restricted Linux distribution, other processes are also working in background and they could not be controlled such as mmeqod (kernel process responsible for the I/O queue) or kworker (kernel worker process). On the other hand, it was not possible to run several times the experiment due to the time it could require.

As far as the number of rules is concerned, we could see that in the case of the real system of the project, we could be around 100 rules related to the sending parameters (around 10 processes with 10 methods/signals each). However, we have to think that each of this rule will have several send_* attributes (from three to four), whereas in the case of our proof-of-concept, we added rules with only two attributes. Moreover, since the destination and interface did not exist, the D-Bus daemon was only checking the first attribute. Therefore, the overhead could be bigger than expected due to the differences in the content of the rules.

Although a more complex and detailed policy leads to an overhead on every board, the benchmark should be redone due to the difference of architecture of the targeted board. Even if the board of the client’s project also contain an Arm 32-bits processor, the board architecture, the other hardware components, and the Linux distribution (built with Yocto and based on a commercial embedded Linux distribution) differ from our board. These differences could lead to different proportionality coefficients.

In addition, by investigating the source code of D-Bus, more particularly the policy.c files, we saw that developers implemented optimisations with the method bus_client_policy_optimize which deletes the rules such as <allow send_interface="hello.world"/> which are followed by a rule <deny send_interface="*"/>. In the case of our proof-of-concept, we did not implement this kind of rules to prevent D-Bus from doing this kind of optimisations. However, developers should keep in mind this fact which could reduce the overhead in comparison to the number of written rules.

7.6 Conclusion

7.6.1 Security Features

Thanks to our simple system, we succeeded in preventing malicious processes from owning well-known name, exposing methods and sending signals. This point is quite useful to prevent a malicious system to use D-Bus as their own mean of communication, reducing the risk of DoS attacks. However, when the attacker can access to the user session, malicious process can connect to the bus.
Without rules about the `receive_*` attributes, attackers can call the methods exposed on D-Bus. We were not able to block them from calling exposed methods or listening to signals if the attacker knows them. However, we reduced the risk of knowing these names by blocking the calls to the D-Bus methods which allow Introspection (i.e., looking at the exposed methods and signals). In our proof-of-concept, we limited the access to the introspection to the monitoring mode for the user `root`. Although this technique relies on the “obscurity principle” to avoid the access to the methods, it is a first level of defence.

On the other hand, by being able to know in advance the unique name of an application on D-Bus by defining the order of connection of the processes, we implemented rules to do filtering on the `receive_*` attributes. However, this solution remains precarious.

We pointed out the limitations of D-Bus which processes separately the `receive_*` and `send_*` attributes. It prevents developers from filtering a specific connection. It only provides filtering on one end of the communication.

Through our analysis, we also underlined the importance of being explicit and preferring a whitelisting approach than a blacklisting one. As a result, the `own_prefix` attribute should not be used due to its permissiveness. In addition, the administrators should state all the possible parameters for a rule. Although it could introduce an overhead, it provides a higher level of security.

As underlined in the description of the configuration file of D-Bus, it is possible to include inside a configuration file other files thanks to the `<include>` tag. This feature could be used to provide modularity for configuring the policy. However, this kind of approach could create security problems related to the linking (e.g., behaviour when file is missing or replacement by another file).

### 7.6.2 Performance

Our performance analysis showed the apparition of an overhead for more complex and detailed policies which is proportional to the number of rules due to the `for` loop used in the C implementation of D-Bus for going through the rules.

However, this overhead could be more difficult to quantify in the case of more complex system. Therefore, engineers could reuse our approach to estimate it.

### 7.6.3 Future Investigation

Firstly, we saw that the writing of the rules was quite straightforward. It does not introduce a significant implementation overhead. Thanks to the D-Bus XML introspection files which list the exposed methods and signals, we could easily generate the rules. It would consist in translating the methods and signals tags into the corresponding rules. These introspection files are already used to automatically generate the proxy and adapter interfaces files thanks to the `dbusxx-xml2cpp` command from the `libdbus-c++` package. Nevertheless, it could lead to long configuration files. Relying on the modularity aspect of these files could be a solution to make them more human readable. For
instance, each process could have its own policy file which will be called from the `<include>` tag of the main configuration file.

During our examination of the monitoring messages, we saw that D-Bus was also providing error messages. This kind of message could be a starting point to develop a process in charge of detecting intrusions or strange behaviours such as an Intrusion Detection System (IDS) of a firewall. This process could be a process with different privileges which could eavesdrop error messages of D-Bus and, depending on the error messages, it could react. A possible use case is the sending of a signal. We were able to prevent SpyA from sending signals and we did not inform it. Only the bus itself was displaying an error message. Therefore, a process which would have access to this kind of messages would be able to detect this unexpected behaviour.

We saw in the description of the configuration file that it could be possible to include SELinux and AppArmor policies. These policies could be the solutions to some of our previous problems. SELinux, which stands for Security-Enhanced Linux, provides a variety of security policies such as the U.S. Department of Defence (DoD) Mandatory Access Controls (MAC). AppArmor is an alternative to SELinux by being easier to configure because SELinux is often considered as difficult to set up. However, due to their complexity, we decided in the first step not to investigate them in the case of our analysis. It could be a possible future work to look at the features which are provided by these policies, their computational overhead, but also their implementation overhead. The use of these policies could also require the modification of the built Linux distribution to integrate the Linux Security Modules (LSM).

Moreover, we investigated a subset of possible attacks. A future work could be to look at other attacks. For instance, we spoke about the possibility of impersonating a process by being able to kill it and reuse its well-known name. This attack would require to know the methods and signals of the XML introspection files, but it could be theoretically feasible.
8 Conclusion on Thesis Work

This chapter summarizes the entire work which was carried out during this internship. It also discusses some aspects and branches out on possible future works.

8.1 Deliveries

At the end of the thesis, in addition to this report, we delivered the following documents:

- Guidelines for Secure Embedded Software Development
- Guidelines references
- Security Analysis of Client’s Project
- D-Bus Security Analysis
- OS-Virtualisation solutions for Embedded Systems Security
- Static Metrics Monitoring Tool
- Dynamic Metrics Analyser

8.1.1 Guidelines for Secure Embedded Software Development

This document represents one of the main tasks of this thesis which was presented in Chapter 5: writing a set of guidelines to higher the level of security of embedded systems during their development.

8.1.2 Guidelines references

This document is an appendix to the guidelines document. It contains a complete list of all the references used to build this document. In addition, this document contains a table which links each guideline with the related subset of references.

8.1.3 Security Analysis of Client’s Project

This document consists of the entire analysis of the specification of a client’s project in regards with our guidelines. This document gives recommendations at the end of the analysis. Some of them were presented in Chapter 6. In addition, it provides in appendices some examples of analysis or proof-of-concepts which could be done to meet some of the recommendation such as the AES analysis we presented in Chapter 6.

8.1.4 D-Bus Security Analysis

This analysis, which was presented in Chapter 7, was put in a specific document. Moreover, the implemented use cases and the benchmarking solution were attached to this document.

8.1.5 OS-Virtualisation solutions for Embedded Systems Security

At the end of the literature review of embedded systems vulnerabilities, we looked at OS-Virtualisation solutions for securing embedded systems and wrote a state of the art about these technologies and their possibilities. However, since it was not identified as a useful solution for the client’s project, we decided not to include this state of the art in our thesis. Then, we put the
review in a separate document which could be useful for other projects as a starting point.

8.1.6 Static Metrics Monitoring Tool

The static metrics tool that we developed and which was presented in Chapter 6 was directly integrated in the Jenkins server. The tool consists of a Jenkins job with a shell script and a Python module. We also generated a documentation for the Python module.

8.1.7 Dynamic Metrics Analyser

The dynamic metrics tool which was presented in Chapter 6 was delivered to the company. It consists of a shell script, a Python package, and a README which explains all the steps to follow to set up the environment and run the analysis.

8.2 Conclusion and Evaluation

Our thesis was aimed to answer to the research questions related to embedded software security: “What are the main and most critical software vulnerabilities that can be encountered in the area of embedded software? How is it possible to address these vulnerabilities and what are the issues which could appear due to the specificity of embedded systems?” We answered to this question by realising different tasks which dealt with different aspects of embedded software security.

Firstly, through this thesis, we were able to deepen the knowledge of the engineers of the company about embedded software vulnerabilities. We wrote a set of guidelines related to secure development of embedded software, and more particularly, we described the different steps which should be followed to design this kind of document.

Although we would have liked to perform more interviews, embedded engineers were quite busy on other projects. Despite having done only one structured interview, we also got a constant feedback on the modifications we did from the supervisor at the company. Moreover, the publication of the guidelines by Bureau Veritas led us to update our way of improving the guidelines. However, we succeeded in taking advantage of this unexpected event to enhance our document and analyse it.

We succeeded in proving the consistency of our guidelines by applying them in the case of a real project. This concrete example allowed us to work on several proof-of-concepts to tackle different aspects of embedded software security.

We carried out an in-depth analysis of an inter-process communication mean: D-Bus. We analysed the different security parameters of the policy and pointed out its strengths and weaknesses. The implementation of a use case demonstrated this point. In addition, we were able to look at the specificities of embedded systems by looking at the performance of the system. We did our analysis by defining protocol and developing benchmarking solutions. We highlighted the performance impact of more detailed security policies.
8.3 Personal Learning and Limitations

Behind the work presented in the thesis, we also widened our personal knowledge on different topics. We learnt how to carry out both quantitative and qualitative analyses. We followed trainings to build our own embedded Linux distribution. We deepened our knowledge about the application domain of the client’s project. We enhanced our skills related to writing code but also about requirement analysis.

One of the main limitations about working with cybersecurity is to find use cases. The literature contains a lot of descriptions of mechanisms and proof-of-concepts related to security. However, finding a use case which could underlines at the same time the advantages and disadvantages of these techniques was more complex than expected. This issue led us to follow an agile approach to do our thesis. We had to permanently analyse if the work we were doing could be applied in the case of the concrete example. This difficulty was underlined by the writing of the analysis of OS-virtualisation solutions for securing embedded applications which did not lead to the implementation of a use case.

The delay that we had encountered to find use cases prevented us from completely diving into the client project and using the complete set of tools. These operations could have required a significant amount of time to understand all the structure of the project and to master the tools such as Yocto. In addition, the device of the client’s project was quite expensive. Not all developers and testers could use this device. Then, it could have been difficult to get one for the internship.

8.4 Future Work

In the Chapter 7 about securing D-Bus, we presented possible future works regarding the use of two Linux Security Modules (LSMs): SELinux and AppArmor.

Moreover, at the beginning of the internship, we spoke about possible aspects of security which could be analysed during the thesis. Due to the client’s project and the time frame of the project, we focused our work on a subset of solutions. However, it could have been interesting to investigate the strengths and weaknesses of library signature to dynamically authenticate these calls. In addition, although we did a literature review of the ways to use OS-virtualisation solutions to harden the security of embedded applications, a future work could be to implement Docker containers on an embedded board and evaluate the security aspects and performance overhead.

At the end of the thesis, some managers of the company were interested to share the good practices guide we had written to other entities of the company. Therefore, the improvement of this document, such as the formalism of the referencing and numbering of the guidelines, could be a future work.

As we underlined in the delimitations of Chapter 1, another intern was working on a hardware security mechanism, the ARM TrustZone. Applying the result of our thesis on this technology could be a starting point for new investigations.
References


Appendix A. Interview Structure

This appendix presents the structure of the interview that we used to improve the guidelines document.

Interviewee’s Background
What is your current position in the company? (e.g., Business Analyst, System Architect, and Developer)
How long have you been working with the development of software product?
How long have you been working with the development of embedded systems?
Do you have any security background? (e.g., company or school training, MOOCs, and personal reading).

General Impression
Before starting, can you give describe, in one sentence/word, your impression after going through this document?

Layout/Size
How did you read it? (e.g., computer screen, phone, tablet, or printed version).
What was your impression of the general layout? (e.g., table of contents, preface, content, sections, and footnote).
How long did it take you to go through this document?
Did the document seem you too short or too fast? (Rate on a scale from 1: too short to 10: too long).

Introduction
Did you read the table of content?
How did it help you to understand/read the document?
Did you read the preface?
How did it help you to understand/read the document?
What do you think about this table of content and preface?

Structure of the document
Rate the structure of the document: From 1 (Not organised/Unclear) to 10 (Well organised/Clear structure). You can give some comments.
Rate the language of the document: From 1 (Not adapted) to 10 (Adapted). You can give some comments (e.g., formal/informal and immature/mature).

Understanding
Rate your understanding of the guidelines: From 1 (understood nothing) to 10 (crystal clear).
What did you not understand?
As far as the misunderstandings are concerned, why did you not understand them? (e.g., unclear speech, unclear idea, too specific, or bad grammar).
Was some background information about cybersecurity missing to help your understanding?
Did you search some pieces of information by yourself?
Learnings
Did you learn something?
What did you learn?

Interest/Specialisation
Which part were you more interested in? Why?
Have you already applied some of these guidelines on your project? Which one?
Could some of these techniques be applied to your project? How?

Improvement (Open Discussion)
What improvements could be added to this document to lead you to integrate it in your projects?
Do you have any additional remarks?
Depending on the rating, more or less in-depth discussions will be done.

Interview Aspect
What did you think about the interview? (e.g., structure, questions, and duration).
Appendix B. Hardware Specifications

Table 9 gathers the specifications of the board which was used to implement our different use cases: a SAMA5D3 Xplained, produced by Atmel. A full presentation of the board could be retrieved in [53]. We used this board to build our custom Linux Distribution thanks to Free Electrons' tutorials [51].

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>SAMA5D36 (324-ball BGA package) ARM Cortex-A5 Processor with ARM v7-A Thumb2® instruction set, core frequency up to 536 MHz.</td>
</tr>
</tbody>
</table>
| Processor Clock Sources | 12-MHz crystal oscillator  
32,768-kHz crystal oscillator |
| Memory                | 2 x 1 Gb DDR2 (16M x 16 bits x 8 banks)  
1 x 2 Gb SLC NAND Flash (256M x 8 bits) |
| Board Supply Voltage  | 5V from USB or power jack or Arduino shield  
On-board power regulation is performed by a Power Management Unit (PMU) |
| Battery               | On-board optional power Cap for CMOS backup |
| OS                    | Custom Embedded Linux built with Build Root based on Linux Kernel 4.4.26-linux4sam_5.5 |
Appendix C. Default Policies

The policy of the default session configuration file is presented in Table 10, whereas the one of the system configuration file is presented in Table 11.

Table 10. Policy of the session configuration file

```
<policy context="default">
  <!-- Allow everything to be sent -->
  <allow send_destination="*"></allow>
  <!-- Allow everything to be received -->
  <allow receive_type="signal">eavesdrop="true"></allow>
  <!-- Allow anyone to own anything -->
  <allow own="*"></allow>
</policy>
```

Table 11. Policy of the system configuration file

```
<policy context="default">
  <!-- All users can connect to system bus -->
  <allow user="*"></allow>
  <!-- Holes must be punched in service configuration files for name ownership and sending method calls -->
  <deny own="*"></deny>
  <deny send_type="method_call"></deny>
  <!-- Signals and reply messages (method returns, errors) are allowed by default -->
  <allow send_type="signal"></allow>
  <allow send_requested_reply="true" send_type="method_return"></allow>
  <allow send_requested_reply="true" send_type="error"></allow>
  <!-- All messages may be received by default -->
  <allow receive_type="method_call"></allow>
  <allow receive_type="method_return"></allow>
  <allow receive_type="error"></allow>
  <allow receive_type="signal"></allow>
  <!-- Allow anyone to talk to the message bus -->
  <allow send_destination="org.freedesktop.Dbus" send_interface="org.freedesktop.DBus"></allow>
  <!-- But disallow some specific bus services -->
  <deny send_destination="org.freedesktop.Dbus" send_interface="org.freedesktop.DBus.Introspectable"></deny>
  <deny send_destination="org.freedesktop.Dbus" send_interface="org.freedesktop.systemd1.Activator"></deny>
</policy>
```

```
<policy context="default">
  <!-- Only systemd, which runs as root, may report activation failures. -->
  <policy user="root">
    <allow send_destination="org.freedesktop.Dbus" send_interface="org.freedesktop.systemd1.Activator"></allow>
  </policy>
</policy>
```

```
<policy context="default">
  <!-- root may monitor the system bus. -->
  <policy user="root">
    <allow send_destination="org.freedesktop.Dbus" send_interface="org.freedesktop.DBus.Monitoring"></allow>
  </policy>
</policy>
```

```
<policy context="default">
  <!-- If the Stats interface was enabled at compile-time, root may use it. Copy this into system.local.conf or system.d/*.conf if you want to enable other privileged users to view statistics and debug info -->
  <policy user="root">
    <allow send_destination="org.freedesktop.Dbus" send_interface="org.freedesktop.DBus.Debug.Stats"></allow>
  </policy>
</policy>
```
Appendix D. Measurement Parameters

Table 12 gives the headers from the perf reports for the policy file where perf_X.data is the name of the recording for the policy with X additional rules.

Table 12. Header of the performance report

```
# ========
# captured on: Fri Jul  7 14:03:10 2017
# hostname : buildroot
# os release : 4.4.26-linux4sam_5.5
# perf version : 4.4.26-linux4sam_5.5
# arch : armv7l
# nrcpus online : 1
# nrcpus avail : 1
# cpudesc : ARMv7 Processor rev 1 (v7l)
# total memory : 251988 kB
# cmdline : /usr/bin/perf record -o PerfRecord/perf_X.data -a -c 1000 sleep 600
# event : name = cycles:ppp, size = 112, { sample_period, sample_freq } = 1000, sample_type = IP|TID|TIME|CPU|PERIOD, disabled = 1, inherit = 1, mmap = 1, comm = 1, freq = 1, task = 1, precise_ip = 3, sample_id_all = 1, exclude_guest = 1, mmap2 = 1, comm_exec = 1
# HEADER_CPU_TOPOLOGY info available, use -I to display
# pmu mappings: not available
# ========
```