Architectural Vulnerabilities in Plug-and-Play Systems

Taylor Corrello

tnc5484@rit.edu
Architectural Vulnerabilities in Plug-and-Play Systems

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Mehdi Mirakhorli, Supervisor

Dr. Pradeep Murukannaiah, Reader

Dr. Scott Hawker, Graduate Program Director
Architectural Vulnerabilities in Plug-and-Play Systems

by

Taylor Corrello

THESIS
Presented to the Faculty of the Department of Software Engineering
Golisano College of Computer and Information Sciences
Rochester Institute of Technology

in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in Software Engineering

Rochester Institute of Technology
May 2018
Acknowledgments

I would first and foremost like to thank my thesis advisor, Dr. Mehdi Mirakhorli of Rochester Institute of Technology. He has consistently provided me with the resources I needed to succeed.

I would also like to recognize Joanna Cecilia Santos for her guidance throughout my research. She dependably allowed the work to be my own, but was a remarkable resource to me in my times of need.

I would also like to thank the experts who were involved in the analysis of CVEs: Joanna Cecilia Santos, Adriana Seja, Tejal Vishnoi, Smruthi Gadenkanahall.

I would also like to acknowledge Dr. Pradeep Murukannaiah as the second reader of this thesis; I am gratefully indebted to him for his very valuable comments on this thesis.

Finally, I must express my very profound gratitude to someone who has never given up on me, someone who has been a big influence on making my dreams a reality, someone who has provided continuous encouragement and unfailing support not only throughout my journey here at Rochester Institute of Technology but throughout my 23 years of life: my mom, Tami Corrello. This accomplishment would have not been possible without her.

Thank you.

iii
Abstract

Architectural Vulnerabilities in Plug-and-Play Systems

Taylor Corrello, M.S.
Rochester Institute of Technology, 2018

Supervisor: Dr. Mehdi Mirakhorli

Plug-and-play architectures enhance systems’ extensibility by providing a framework that enables additional functionalities to be added or removed from the system at their runtime. Such frameworks are often implemented through a set of well-defined interfaces that form the extension points for the pluggable functionalities. However, the plug-ins can increase the applications attack surface or introduce untrusted behavior into the system. Designing a secure plug-and-play architecture is critical and non-trivial as the features provided by plug-ins are not known in advance. In this paper, we conduct an in-depth study of seven systems with plug-and-play architectures. In total, we have analyzed 3,183 vulnerabilities from Chromium, Thunderbird, Firefox, Pidgin, WordPress, Apache OfBiz, and OpenMRS whose core architecture is
based on a plug-and-play approach. We have also identified the common security vulnerabilities related to the plug-and-play architectures, and mechanisms to mitigate them by following a grounded theory approach. We found a total of 303 vulnerabilities that are rooted in extensibility design decisions. We also observed that these plugin-related vulnerabilities were caused by 15 different types of problems. We present these 15 types of security issues observed in the case studies and the design mechanisms that could prevent such vulnerabilities. Finally, as a result of this study, we have used formal modeling in order to guide developers of plug and play systems in verifying that their architectures are free of many of these types of security issues.
Table of Contents

Acknowledgments iii
Abstract iv
List of Tables ix
List of Figures x
Chapter 1. Introduction 1
Chapter 2. Background 5
Chapter 3. Vulnerability Analysis 9
  3.0.1 Limiting the Phenomena Under Study ............... 10
  3.0.2 Data Collection: Theoretical Sampling ............... 11
    3.0.2.1 Theoretical Sampling ...................... 11
    3.0.2.2 Data Sources ............................ 12
    3.0.2.3 Data Fusion ............................... 14
    3.0.2.4 Data Preprocessing ....................... 16
  3.0.3 Open Coding .................................. 17
  3.0.4 Constant Comparison Method ....................... 19
  3.0.5 Memoing ...................................... 20
  3.0.6 Selecting Coding ................................ 21
    3.0.6.1 Data Analysis Instrument ................. 22
  3.0.7 Memo Sorting .................................. 22
  3.0.8 Theoretical Coding ................................ 22
  3.0.9 Literature Review & Write Up ..................... 23
Chapter 4. Vulnerability Patterns Found

4.0.1 Plug-in Install
- 4.0.1.1 Incorrect user notification of plug-in permissions
- 4.0.1.2 Bypassing user confirmation for plug-in installation
- 4.0.1.3 Lack of plug-in’s configuration file sanitization
- 4.0.1.4 Improperly checking the origin of an install request

4.0.2 Plug-in Updates
- 4.0.2.1 Elevation of privilege through a plug-in update

4.0.3 Plug-in Registry Management
- 4.0.3.1 Extraction/Storage of Plug-in with world readable/writable permissions or in unsafe directories

4.0.4 Plug-and-Play Execution Environment
- 4.0.4.1 Lack of compartmentalization of plug-ins
- 4.0.4.2 Lack of fine-grained and modular permission setting
- 4.0.4.3 Allowing a plug-in to elevate its permission by manipulating (or delegating a task to) a process in the plug-and-play environment that has higher privileges
- 4.0.4.4 Improper object access control and compartmentalization enforcement
- 4.0.4.5 Unsanitized plug-in data
- 4.0.4.6 Improper origin check of requests by plug-ins
- 4.0.4.7 Improper isolation of objects used by plug-ins in the plug-and-play environment

4.0.5 Plug-ins Request Handling
- 4.0.5.1 Reentrant event callbacks
- 4.0.5.2 Plug-ins requests are handled without authorizing plug-ins that initiate the request

Chapter 5. Evaluation of Data Collection / Analysis
## Chapter 6. Formal Modeling

6.1 Overview of Approach .................................................. 43  
   6.1.1 AADL ................................................................. 43  
   6.1.2 AGREE ............................................................... 44  
6.2 Basic Example ............................................................... 45  
6.3 Step 1: Model the Plug-in .................................................. 46  
6.4 Step 2: Model the Core .................................................... 48  
6.5 Verification ................................................................. 50  
   6.5.1 Plug-ins are kept up to date .......................................... 51  
   6.5.2 Plug-in Request Handling ............................................. 52  
6.6 Case Study: Modeling a Full System ...................................... 53  

## Chapter 7. Threats to Validity

## Chapter 8. Conclusion and Future Work

8.0.1 Future Work .............................................................. 59  
8.0.2 Conclusion ............................................................... 59  

## Appendices

Appendix A. Filtered CVE Appendix .......................................... 61  
   A.0.1 Model Declarations .................................................. 61  

## Bibliography

## Vita
List of Tables

3.1 Keywords used for automatically filtering CVEs . . . . . . . . . . 15
3.2 Statistics of the vulnerability data used in this study. . . . . . . . . 16
3.3 Sample memo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

4.1 Findings overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
List of Figures

3.1 The Grounded Theory Approach Applied to our Work . . . . 9
3.2 Information model for the collected data. . . . . . . . . . . 10
3.3 Examples of open coding of CVEs . . . . . . . . . . . . . 18
3.4 Codes that resulted from our open coding process . . . . . 20

6.1 Basic system correctly connected . . . . . . . . . . . . . . 46
6.2 Basic system correctly connected and verified . . . . . . . . 47
6.3 Basic System incorrectly connected . . . . . . . . . . . . . 48
6.4 Basic System incorrectly connected and verified . . . . . . . 49
6.5 Plug-in AADL Model . . . . . . . . . . . . . . . . . . . . . . 50
6.6 Core AADL Model . . . . . . . . . . . . . . . . . . . . . . . 51
6.7 Verification for KarzInc Base Car . . . . . . . . . . . . . . 54
6.8 Verification for a custom car that does not take into account all mitigations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
Chapter 1

Introduction

Plug-and-play architectures are widely adopted in many application domains to enhance systems’ extensibility, reusability and modifiability [12]. For instance the automotive industry is rapidly creating plug-and-play architectures where software modules slot into the overall electronic architecture without unexpectedly disrupting other modules [28]. In the finance domain, plug-and-play architectures provide universal APIs for in-store point of sale (POS) systems and enable plugging a variety of different applications into the POS system at the merchant. In medical device development, the plug-and-play architectures are used to enhance programs and medical device interoperability, where third-party medical applications are plugged into networked medical devices to provide diagnosis, treatment, research, safety and quality improvements, and equipment management features[6, 7].

In plug-and-play architectures, the software is decomposed into a “core” component representing the plug-and-play environment of the host application and a set of bundles representing “plug-ins”. The plug-and-play environment provides the software’s main functionalities and a runtime infrastructure for plug-ins. Plug-ins provide bundled functionalities which can be added at run-
time, making the software customizable and extensible. This means that the software product can be released early, and new features can be added later through plug-ins. It also means that the software can be customized to address specific needs of specific instances. Moreover, plug-and-play architectures can enable contributions from third-party vendors because extending the architecture does not require access to the source code, but instead, these third-party developers can implement well-defined public interfaces provided by the plug-and-play environment.

Although plug-ins are useful for adding new features to the software, they can increase the application’s attack surface or introduce untrusted behavior. Designing a secure plug-and-play architecture is critical and non-trivial as the features provided by plug-ins are not known in advance and inclusion of the third party functions can negatively affect the systems security and trustworthiness [14, 43]. There are numerous vulnerabilities reported for plug-and-play architectures [38, 44, 49, 56]. For instance, a group of researchers have demonstrated how hackers can wirelessly access the critical driving functions of a vehicle through an entire industry of Internet-enabled gadgets plugged directly into cars’ dashboards to monitor vehicles’ location, speed and efficiency [26]. In this case, the plug-in was insecure, however, severe security and privacy issues could also occur when the system accepts malicious plug-ins [37].

Although there are numerous studies in the area of plug-and-play software architectures [67, 68], their applications in various domains [8, 28, 36] and
securing domain-specific examples of such extensible architectures [11, 15], we
currently lack an empirically grounded work that aims to understand common
types of vulnerabilities that are associated with plug-and-play software archi-
tectures as well as novel mitigation techniques to prevent such vulnerabilities.
Therefore, in this paper, we follow a grounded theory approach to derive a the-
ory around the domain of vulnerabilities in plug-and-play software systems.
We strictly followed the classical version of the grounded theory, and ana-
lyzed vulnerability reports of several open source systems with plug-and-play
architecture.

The contributions of our work are:

- An in-depth discussion of common types of vulnerabilities in plug-and-
  play software architectures based on data from real, widely used plug-
  and-play projects.

- An empirically grounded presentation of architectural mitigations that
could prevent or minimize the impacts of common types of security is-
  sues.

- A reference security architecture for plug-and-play software systems ver-
  ified by formal modeling.

This paper is organized as follows: Chapter 2 presents background on
grounded theory. Chapter 3 provides an overview of our study’s methodol-

---

1Our data is released at: https://drive.google.com/drive/folders/1LmZuuRaEtSbpE\_TYr0rrAl1Q-cF63-\_B
ogy. Chapter 4 provides the common types of plug-and-play vulnerabilities we observed in the case studies. Chapter 5 provides an evaluation of our data collection approach. Chapter 6 provides the formal modeling of some of our mitigation strategies. Chapter 7 presents the threats to the validity of this work. Lastly, Chapter 8 concludes this paper.
Chapter 2

Background

A grounded theory approach is a progressive identification and integration of concepts from data that leads to the construction of theories directly supported by empirical data. Grounded Theory is well-known for having different variants. These differences lie on the role of the literature and the data analysis process. The classical grounded theory encompasses the following activities: identification of topic of interest, theoretical sampling, data coding (through open, selective, and theoretical coding), constant comparative analysis, memo writing, memo sorting and write up & literature review.

Defining the Phenomena Under Study.
Researchers are advised against formulating a specific research question upfront, but rather, define an area of interest (i.e., the phenomena under observation). The idea is that the research question should also emerge from the data at hand, such that it minimizes the potential biases incurred by having a pre-defined research question.

Theoretical sampling
Researchers perform their data collection procedures via theoretical sampling, which is the process of jointly collecting and analyzing data in order to de-
cide what data needs to be collected next [29]. In conventional sampling approaches, the data collection is preplanned with the aims of increasing the statistical generalizability of the results [62]. However, in theoretical sampling, the data collection process is guided by the earlier rounds of sampled data, which identify knowledge gaps that need further explanation, thereby requiring more rounds of data collection [31].

Coding of Data
As data is collected, researchers start their data coding processes. In classical grounded theory there are two types of codes that are resultant of these coding processes: substantive codes and theoretical codes. Substantive codes are concepts and properties that emerge from the data and reflect the nature of the phenomena under study. Theoretical codes, on the other hand, integrate the substantive codes by indicating how they relate with each other, forming the hypotheses to be integrated into the theory. While substantive codes are the outcomes of open coding and selective coding, theoretical codes emerge from the theoretical coding process.

In the initial phases of data analysis, researchers perform open coding which consists on analyzing each of the incidents (i.e., data points) in order to annotate them with codes (concepts). The open coding can be performed at different levels of granularity (line-by-line, paragraph-by-paragraph, etc) [30, 31]. These codes are constantly refined throughout the open coding process, leading to the emergence of a core category and its associated concepts. The core category is the main concern or problem observed in the phenomena
under study [29]. As stated by Glaser in his seminal work, the core category “accounts for a large portion of the variation in a pattern of behaviour” [29].

Once the core category is identified, researchers perform the selective coding of data, in which further data collection and analysis are delimited and focused in saturating the core category. This means that the coding continues until the theory’s concepts are fully supported by the data, and no new concepts are observed (theoretical saturation is achieved).

The last phase of coding is the theoretical coding phase which involves establishing the relationships between the substantive codes, leading to the development of hypotheses that would integrate to our theory. Theoretical coding involves the application of a coding paradigm [31]. Glaser [29] defined a list of coding families that could help researchers to interconnect concepts derived from the data during their theoretical coding process.

Memo Writing (or “Memoing”) Throughout the data analysis, researchers take notes in “memos”. These memos capture any insights of the researcher and consists of “logs” of the process of developing the theory.

Constant Comparative Analysis One crucial aspect in developing a solid theory is to constantly compare incidents and concepts with each other in order to identify similarities and differences between emerging concepts. This process of comparing back and forth between codes and incidents is crucial to ensure same level of granularity of
concepts.

**Write Up & Literature Review**

The last stage occurs when we achieve a *theoretical saturation*, in which all the concepts that are part of the theory are fully supported by the data, and there is no knowledge gap (i.e., hypotheses that need further clarification through an additional round of data collection and analysis). At this last stage, the researcher writes up his/her theory. Subsequently, the researcher performs a literature review with the aim to compare and augment the originated theory against the theories described in the literature. The literature review is delayed until the theory is fully developed to prevent any biases during the analysis [29].

The output of this process is a **theory**. *A theory consists of a series of interconnected concepts that offers an explanation about a problem (core category) in a given context and how that main concern is handled* [31]. *In other words, it is a narrative based on the evidence from the data that can explain the aspects of a certain phenomena.*
Chapter 3

Vulnerability Analysis

Figure 3.1: The Grounded Theory Approach Applied to our Work

We used the classical grounded theory [29] as a systematic inductive method for conducting qualitative research of software vulnerabilities in plug-and-play architectures, aimed toward theory development for the common vulnerabilities of plug-and-play architectures, as well as their mitigation techniques. We chose this approach due to its emphasis on the emergence of concepts [30, 62], i.e., high emphasis on an inductive rather than a deductive data analysis. Since we do not know in advance the nature of the vulnerabilities (except a high-level knowledge that they are rooted in plug-and-play systems), our goal was to allow the data to drive our process of discovering classes of plug-and-play vulnerabilities (inductive reasoning) rather than formulating hypotheses throughout the analysis process (deductive reasoning). Figure 3.1 shows how we applied the classical grounded theory to our research. We explain this process in the subsections that follow.
3.0.1 Limiting the Phenomena Under Study

In following a grounded theory approach, instead of forming initial research questions we define an area of interest. As illustrated in Figure 3.1, the focus of this study is vulnerabilities in plug-and-play software systems. We focused on vulnerabilities specific to the plug-and-play architecture - i.e., security issues that are enabled due to the extensibility mechanisms provided by plug-and-play environments and are specific to such architecture.

These vulnerabilities are a subset of the total vulnerabilities that can be exploited in the system, because we are only focusing on the ones that result from including plug-in mechanisms in the architecture. These plug-in related vulnerabilities can be caused by malicious or benign-but-buggy\textsuperscript{1} plug-ins.

\textsuperscript{1}Benign-but-buggy plug-ins refers to the plug-ins that have a benign behavior that contains a security defect that can be used by attackers to exploit the application core. [11]
3.0.2 Data Collection: Theoretical Sampling

Given the topic of interest of this work, we needed access to software vulnerability reports, the description of these vulnerabilities, in-depth discussion about how they occurred and were fixed, as well as information about the architectural decisions of the projects affected by these security problems. Therefore, we targeted data sources that are freely accessible to us. In this context, we focused on open source systems with a plug-and-play software architecture.

3.0.2.1 Theoretical Sampling

We began by sampling two open source projects, Chromium, and Thunderbird to extract and analyze their vulnerability reports. From an initial analysis of these reports, we observed that Thunderbird and Chromium had overlapping concepts since they were from a similar domain. Therefore, we included more projects in which we extracted and analyzed their vulnerabilities. The additional projects were Firefox, WordPress, and Pidgin. Although Firefox and Chromium were from the same domain, adding them could help us pinpoint problems that are only applicable to Web browsers and other concepts that are more generalizable. In the later stages (after we identified our core categories), we sampled more open source projects from different domains. We included OpenMRS and Apache OfBiz for further analysis and to support the findings of our theory.
3.0.2.2 Data Sources

We used the National Vulnerability Database (NVD) to extract vulnerability meta-data, the Issue Tracking Systems to obtain further discussions about the problem, the Source Code Repositories to identify fixes for these vulnerabilities and Technical Documents that explain the underlying plug-and-play mechanisms of the affected software project. The process of extracting data from these sources is described below:

- **Retrieving vulnerabilities from NVD**: We obtained the vulnerability reports from the National Vulnerability Database through parsing their public data feeds [47]. Vulnerabilities disclosed in NVD are assigned a unique Common Vulnerabilities and Exposures Identifier (CVE ID). Along with this identifier, vulnerabilities tracked by NVD contain a concise description of the problem, a list of affected software releases, and a list of Web sites that can be used as references and can contain more details about the problem [47].

- **Identifying vulnerability details from Issue Tracking Systems**: Although CVE reports provide a brief description of the security problem, they do not contain a detailed discussion about the vulnerability such that we could verify its underlying root cause, consequences and other information. Therefore, we also identified URLs to the corresponding bug entry of the issue tracking system of the case study. This way, we could read the developers’ discussion about the problem and the process they used...
to come up with a solution. To do so, we leveraged the list of “references” for the CVE and identified which of these links referred to the issue tracking system of the corresponding case study.

- **Collecting vulnerability patches from Source Code Repositories:** To retrieve patches that fixed vulnerabilities, we extracted the commits that referred to the corresponding bug entry in the issue tracking systems (i.e., commits whose message explicitly mentions the bug id). These patches contain the files that were affected (i.e., modified, added or removed) in the fix. The identification of the patches helps us to verify what solution was applied by developers to repair the software.

- **Identification of design decisions for enabling Plug-and-Play:** We reviewed available literature, existing technical documentation, posts in the projects issue tracking systems and existing architectural diagrams of each case study in order to identify their design decisions for supporting plug-and-play feature and any security mechanism adopted for protecting their plug-and-play environment. We also review their source code to understand the structure of the application and technical decisions. This review was conducted using a keyword search, manually browsing the source code, reading any code comments or “readme” files, as well as the release reports. We compiled our findings in a trace matrix which enumerates where each plug-and-play mechanism is implemented in the source code. Each project’s trace matrix of plug-and-play design decisions to source files was peer reviewed.
3.0.2.3 Data Fusion

By the end of an iterative data collection and theoretical sampling process, we collected a total of 3,183 vulnerability reports (CVEs) and associated data. Figure 3.2 shows the information model of the vulnerability data we have collected after performing the steps enumerated above. We used three complementary approaches to identify the subset of vulnerabilities (CVE instances) that were associated with the plug-and-play architecture of these systems. These approaches were:

- **Component-Based Approach**: The issue tracking entries to fix CVEs often have an attribute indicating the affected software component, which is declared by the original developers of the project. Thus, we leveraged this component tag to identify CVEs that are potentially related to their extensible architecture. To do so, we defined a list of component tags that are associated with the plug-and-play architecture of each case study. This list was established after a careful review of the projects’ technical documents and source code. Next, we filtered all the issue tracking reports whose bug matched the component tag of our subset. Lastly, we traced the bug ids of these entries back to their associated vulnerabilities (CVE instances) in order to identify the subset of CVEs that are potentially related to securing their extensible architecture.

- **Keyword-Based Approach**: A list of keywords was established that reflected the terminology used by the developers to refer to the plug-and-
Table 3.1: Keywords used for automatically filtering CVEs

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox</td>
<td>extension, bundle, theme, add on, add-on, addon, plugin, plug-in, dictionary, xpi, pack</td>
</tr>
<tr>
<td>Chromium</td>
<td>extension, plug-in, plugin, app</td>
</tr>
<tr>
<td>Thunderbird</td>
<td>extension, bundle, theme, add on, add-on, addon, plugin, plug-in, dictionary, xpi, pack</td>
</tr>
<tr>
<td>Wordpress</td>
<td>plug-in, plugin, theme</td>
</tr>
<tr>
<td>Pidgin</td>
<td>plug-in, plugin</td>
</tr>
<tr>
<td>OfBiz</td>
<td>plug-in, plugin</td>
</tr>
<tr>
<td>OpenMRS</td>
<td>plug-in, plugin, add on, add-on, addon</td>
</tr>
</tbody>
</table>

play architecture of each case study. These keywords were searched on the descriptions of the retrieved CVEs to identify those related to plug-ins. Table 3.1 enumerates the keywords used per case study.

- **File-Based Approach**: The traceability matrix of plug-and-play mechanisms to source files, developed during our data collection (Section 3.0.2.2) was used to locate plug-in related source files. The plug-in related CVEs were identified by mapping the files in the trace matrix to the source files affected by CVEs.

The goal of these three complementary approaches was to maximize the recall of all CVEs related to the plug-and-play architecture. Table 3.2 shows the total number of retrieved CVEs (column “# CVEs”), how many of these CVEs were selected after applying the three previous approaches (column “# Analyzed CVEs”), and lastly the total number of CVEs related to plug-and-play architecture after a manual review ( “# Plugin-related CVEs”).
Table 3.2: Statistics of the vulnerability data used in this study.

<table>
<thead>
<tr>
<th>Case Study</th>
<th># CVEs</th>
<th># Analyzed CVEs</th>
<th># Plugin-related CVEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox</td>
<td>1396</td>
<td>156</td>
<td>68</td>
</tr>
<tr>
<td>Chromium</td>
<td>1252</td>
<td>169</td>
<td>73</td>
</tr>
<tr>
<td>Thunderbird</td>
<td>704</td>
<td>85</td>
<td>37</td>
</tr>
<tr>
<td>Wordpress</td>
<td>433</td>
<td>221</td>
<td>91</td>
</tr>
<tr>
<td>Pidgin</td>
<td>69</td>
<td>34</td>
<td>32</td>
</tr>
<tr>
<td>OfBiz</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OpenMRS</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3.0.2.4 Data Preprocessing

After collection, merging and filtering of the CVE reports, preprocessing was conducted in order to summarize the data for the analysts to start the coding of the data based on grounded theory. The data preprocessing was performed by five individuals with security background, who systematically scrutinized the subset of CVEs that were identified using the three complimentary automated approaches described in the previous step. These individuals summarized the vulnerability reports by filling out a form containing specific sections for the:

- **Context**: underlying scenario in which the vulnerability occurred;
- **Problem**: why it occurred (fine-grained root cause);
- **Solution**: how it was fixed;

These summaries are important for us to minimize the information load when coding and constantly comparing a large amount of data. All summaries are
also released through the link to study package.

### 3.0.3 Open Coding

After preparation of the data, the first step was the *open coding* of the vulnerability summaries. Five security researchers with an average of two years experience in the security domain conducted the open coding practice. During this process, they analyzed each of the plugin-related CVE summaries, reviewed its *context, problem,* and *solution* that were collected previously (and any other details available in the issue tracking system or other sources as needed). After reviewing the CVE information, researchers collaboratively highlighted the *key points* in the summaries, then based on these key points they assigned codes to the vulnerability. The codes were used as delegates for concepts and key points involved in vulnerability. Figure 3.3 lists all the initial codes generated by the analysts through the open coding process and their frequencies (number of CVEs presented with the code). Please note, that many of these codes in further iterations have been grouped into core categories.

Figure 3.3 shows the summary report collected for three CVEs. For instance, in case of CVE-2015-4498, the *key points* are highlighted in red color: *add-on installation, allows remote attackers to bypass an intended user-confirmation, warns the user, bypass this install warning dialog, installation of the add-on will start without the dialog, and block cross-origin add-on install request.* Then each of which are assigned a code, example of codes generated
<table>
<thead>
<tr>
<th>CVE-2015-4498</th>
<th>CVE-2011-3055</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong> The add-on installation feature in Firefox before 40.0.3 allows remote attackers to bypass an intended user-confirmation requirement by constructing a crafted data: URL and triggering navigation to an arbitrary http: or https: URL.</td>
<td><strong>Description:</strong> The browser native UI in Google Chrome before 17.0.963.83 does not require user confirmation before an unpacked extension installation, which allows user-assisted remote attackers to have an unspecified impact via a crafted extension.</td>
</tr>
<tr>
<td><strong>Problem:</strong> Normally, Firefox warns the user when trying to install an add-on if this install request was initiated by a Web page. This warning needs to be explicitly accepted for the add-on to continue installing. However, there is one exception in which the dialog will not be shown, which is when the user pastes the direct link in the URL bar. An attacker could leverage this exception scenario to bypass this install warning dialog. Basically, an attacker could create links to Web pages that redirect to the location of the add-on’s bundle (XPI file). When the user clicks on the link, the Web browser will follow the chain of redirects, and the installation of the add-on will start without the dialog.</td>
<td><strong>Problem:</strong> An attacker was able to gain access to the extensions management page and get it to load an unpacked extension with an NPAPI plugin (see also bug 117715) without generating a prompt. Looking at the code in UnpackedInstaller::OnLoaded, it looks like it should generate a prompt in all cases unless the extension is disabled.</td>
</tr>
<tr>
<td><strong>Solution:</strong> Fix is to block cross-origin add-on install requests.</td>
<td><strong>Solution:</strong> The fix is to generate the same prompts for packed and unpacked extensions. This also fixes an issue where we were not prompting for unpacked extensions with plugins at installation time.</td>
</tr>
<tr>
<td><strong>Codes:</strong> “Not showing install warning dialog”, “Silent install of plug-ins”, and “Block cross-origin install requests”.</td>
<td><strong>Codes:</strong> “Not showing install warning dialog”, “Silent install of plug-ins”, “Consistent generation of install warning prompts”.</td>
</tr>
</tbody>
</table>

---

Figure 3.3: Examples of open coding of CVEs
for this summary are: *Not showing install warning dialog; Silent install of plug-ins; Block cross-origin install requests.*

### 3.0.4 Constant Comparison Method

The codes emerging from each CVE summary were constantly compared against the existing codes to observe commonalities and differences (which could result in further break down of these codes into more fine-grained levels). Emerging codes were compared against other vulnerability reports in order to observe their properties (such as potential mitigations and types of consequences). Furthermore, CVE instances were compared against other vulnerability reports to establish uniformity of concepts and identify variations. Through constant comparison, security researchers observed that some key points reoccurred, then such key points were used to form the core categories. For instance, in Figure 3.3 the key points for CVE-2015-4498 and CVE-2011-3055 are similar, and they have been assigned codes such as “Not showing install warning dialog”, or “Silent install of plug-ins”.

Then, emerging concepts are compared to more incidents to generate new theoretical properties of the concepts and more hypotheses. The goal of the constant comparative method is to ensure that all the concepts are supported by the data and at the same level of granularity. As the analysts performed the analysis, they were either annotating the CVEs with existing tags or creating new codes that emerged (i.e., the existing tags are not suitable for the CVE being analyzed). For instance, in case of CVE-2012-0934 (Fig-
None of existing codes for CVE-2015-4498 and CVE-2011-3055 could represent it, therefore, we created new codes for it.

The result of this open coding and constant comparative analysis iteration is the identification core categories [29]. In our study, our core categories correspond to the types of plug-and-play vulnerabilities and their corresponding mitigations that we found from the observations.

Figure 3.4: Codes that resulted from our open coding process

3.0.5 Memoing

Throughout the iterative process of open coding and constant comparative analysis, the researchers captured their insights in memos. A shared Google Document with predefined tables was used to capture early insights. In these early stages of data analysis, our memos mostly concerned potential core categories (plug-and-play vulnerabilities) and as the process continued we finalized them by adding more detailed information about consequences and
mitigation techniques. For these potential core categories, these memos would capture a summary of the type of architectural violation, associated consequences and how it can be mitigated. Table 3.3 illustrates a sample memo captured by an analyst during the memoing process.

Table 3.3: Sample memo

**Memo#19: Unsanitized plugin data**

**Problem:** The core application interacts with data from the plugins. The problem arises when this data is not properly sanitized. The application host trusts data from the plug-in when it shouldn’t because this data is crossing boundaries.

**Mitigation:** Introduce mechanisms that sanitize the data flowing from plugins to the core application.

**Consequence(s):** Arbitrary code execution, Denial of service

**Some observed examples:**
- CVE-2005-0752 [Firefox]: The Plugin Finder Service (PFS) in Firefox before 1.0.3 allows remote attackers to execute arbitrary code via a javascript: URL in the PLUGINPAGE attribute of an EMBED tag.
- CVE-2013-0896 [Chrome]: BrowserPluginGuest trusts the shared memory region sizes passed in messages from renderers. When the browser attaches to these regions it does not sanity check the region sizes and can be made to write beyond the end of the mapped region.
- CVE-2012-5328 [WordPress]: Multiple SQL injection vulnerabilities in the Mingle Forum plugin 1.0.32.1 and other versions before 1.0.33 for WordPress might allow remote authenticated users to execute arbitrary SQL commands.

### 3.0.6 Selecting Coding

The selective coding of our methodology focused on theoretically saturating the architectural and related concepts. In this step, we go back to the CVE instances that were associated with these architectural violations in order to further refine these violations, capturing all possible consequences observed in the data, and how developers mitigated them. In this and later
stages of our analysis, our memos encompassed theory development, in which we focused on rearranging our core categories for establishing our cohesive theory.

3.0.6.1 Data Analysis Instrument

It is important to highlight that we used a custom-built Web-based tool to support our activities of coding the data. This Web tool presents to the researcher the information retrieved for each vulnerability report (Figure 3.2), and enables the researcher to annotate the report, and tag codes (i.e., concepts) to the report.

3.0.7 Memo Sorting

At the ending stages of our data analysis, we conceptually sorted our memos. By sorting we do not imply a chronological order, instead, the sorting of our notes based on inter-related concepts. The goal of this sorting is to look at the data at a higher-level of abstraction.

3.0.8 Theoretical Coding

In the later stages of our analysis, we employed theoretical coding in order to interconnect substantive codes, leading to the development of hypotheses that would integrate to our theory. As explained in Section 2, the theoretical coding involves the application of a coding paradigm [31]. In this coding process, we integrated our concepts and structured them into contexts,
which are the underlying scenario of the plug-and-play vulnerability, *causes*, that are the contributing factors that lead to the vulnerability, and the *consequences* of the vulnerabilities.

3.0.9 Literature Review & Write Up

The last step of our methodology is writing our theory. In this writing process, we also performed a Systematic Literature Review (SLR) to compare and cross-reference findings from our theory with respect previous work.

The search strategy [72] of our literature review consisted in a manual search for works from four sources: the ACM Digital Library, IEEE Explore Library, ScienceDirect, and Springer Link. Our inclusion criteria were as follows: the work was (i) a full paper; and (ii) focused on discussing security problems on plug-and-play software architectures. Exclusion criteria were (i) position papers, short papers, tool demo papers, keynotes, reviews, tutorial summaries, and panel discussions; (ii) not fully written in English; (iii) duplicated study and (iv) focusing on a research problem not in the domain of plug-and-play software architectures. In our manual search, we used the following search query: *(plug-in OR plugin OR extension) AND (security OR vulnerability OR vulnerabilities)*.

From our manual search we collected a total of 11,053 papers. We applied our inclusion and exclusion criteria through reading the paper’s title, abstract and keywords (if existent), resulting in a remainder of 33 papers. These remaining papers were carefully reviewed, where we focused on verifying
to what extent the findings from our theory were new (the novelty of our theory). Our theory is presented in Section 4 along with comparisons with the literature.
## Chapter 4

### Vulnerability Patterns Found

Through this in-depth study, we identified 15 common types of security problems that occur in plug-and-play systems. Table 4.1 shows a concise view of the findings organized with the types of plug-and-play vulnerabilities, their context and consequences.

In the following subsections, we describe all the results within their identified context (e.g. plug-and-play functional context).

<table>
<thead>
<tr>
<th>Types of Plug-and-Play Vulnerabilities</th>
<th>Context</th>
<th>Consequences</th>
<th>Mitigation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insecure or modifications of plug in components</td>
<td>Arbitrary code execution, Privileged execution</td>
<td>Configuration checks performed on the plug-in configuration file to ensure it is not malicious.</td>
<td>Configuration checks performed on the plug-in configuration file to ensure it is not malicious.</td>
</tr>
<tr>
<td>Improper input validation of plug in data</td>
<td>Arbitrary code execution, Privileged execution, Data injection, SQL Injection, Code injection, Privilege escalation</td>
<td>Input validation of incoming plug-in data is performed to ensure it is not malicious.</td>
<td>Input validation of incoming plug-in data is performed to ensure it is not malicious.</td>
</tr>
<tr>
<td>Improper object access control in compartmentalized Plug-in environment</td>
<td>Arbitrary code execution, Change the Plug-in execution environment, Privileged execution</td>
<td>Configuration checks performed on the plug-in configuration file to ensure it is not malicious.</td>
<td>Configuration checks performed on the plug-in configuration file to ensure it is not malicious.</td>
</tr>
<tr>
<td>Improper origin check of requests by plug-in</td>
<td>Arbitrary code execution, Change the Plug-in execution environment, Privileged execution</td>
<td>Access control checks performed on incoming requests to ensure they are not malicious.</td>
<td>Access control checks performed on incoming requests to ensure they are not malicious.</td>
</tr>
<tr>
<td>Improper operation of objects used by plug in Plug-in environment</td>
<td>Arbitrary code execution, Change the Plug-in execution environment, Privileged execution</td>
<td>Authorization checks performed on objects used by the plug-in to ensure they are not malicious.</td>
<td>Authorization checks performed on objects used by the plug-in to ensure they are not malicious.</td>
</tr>
<tr>
<td>Improper handling of plug in requests</td>
<td>Arbitrary code execution, Change the Plug-in execution environment, Privileged execution</td>
<td>Access control checks performed on incoming requests to ensure they are not malicious.</td>
<td>Access control checks performed on incoming requests to ensure they are not malicious.</td>
</tr>
</tbody>
</table>

### Table 4.1: Findings overview
4.0.1 Plug-in Install

One of the most basic features in a plug-and-play system is to load and install new plug-ins to the application at runtime. In this context, we found the following types of problems:

4.0.1.1 Incorrect user notification of plug-in permissions

When a new plug-in is added to the system, it can request access to certain data/functionality provided by the plug-and-play environment. This problem occurs when the plug-and-play environment does not (or incorrectly) shows the list of data and/or functionality that will be accessed by the plug-in before the plug-in is installed.

- **Consequences**: It opens space to user-assisted attacks, in which the user is misled to trust and consequently accept the install of a potentially over-privileged plug-in. This results in plug-ins being able to gain privileges, and perform unintended activities, such as access sensitive data (data leakage).

- **Mitigation**: Similar to the vulnerability “Bypassing user confirmation for plug-in installation” (Section 4.0.1.2) the mitigation is to adopt a Central Install Point that would display a list of all the permissions being requested by the plug-in (besides consistently generating for mitigating the problem discussed in Section 4.0.1.2).

- **Evidence**: We found two cases of this vulnerability in our data analy-
sis. In our SLR we found further evidence [65] of this vulnerability, in which the authors experimented with malicious extensions in Chrome and developed a “browshing” (browser + phishing) attack.

4.0.1.2 Bypassing user confirmation for plug-in installation

Whenever a new install is requested, the plug-and-play environment should ask the user for consent to proceed (or abort) the install. This type of vulnerability is caused by not strictly enforcing a requirement to have all install requests mediated by the user.

- **Consequences**: It can result in stealth installation of malicious plug-ins. Since these plug-ins were silently installed, an attacker could leverage this to perform arbitrary malicious activities, such as stealing data.

- **Mitigation**: It consists of having a *Central Install Point* in which all the installation requests, regardless of how initiated, are guaranteed to go through this central installation component. This component ensures that the install process of a plug-in is designed to prompt the user for consent before proceeding with the install.

- **Evidence**: We found six CVEs that were caused by this type of problem. Our SLR did not yield any results in this category. As an example, by design, Firefox does not ask the user to confirm the install of an add-on if the URL to the add-on was copied and pasted directly into the browser’s address bar. The underlying assumption is that it did
not represent a risk because the end-user was the one who typed the URL. Although this design helps to improve usability, attackers could leverage this design decision to silently install their malicious extensions, as reported in CVE-2015-4498. They would create a Web page that has a direct link to the malicious add-on bundle (XPI file), naming it with any well-known benign extension. Once the user clicks on the link, Firefox will download and install without any confirmation.

4.0.1.3 Lack of plug-in’s configuration file sanitization

This problem is caused by not validating the plug-in’s configuration file in order to verify whether it is structurally correct, and also escape/neutralize any code that is injected into the plug-in’s configuration file.

- **Consequences**: A malicious extension or a faulty benign plug-in may contain a malformed configuration file (e.g. omitting required fields or injecting code in the fields). Such issue could result in various security risks such as crash of plug-and-play environment, a directory path traversal, or data leakage. Malicious extensions could leverage this vulnerability to inject arbitrary code and perform numerous unintended activities (such as installing another plug-in).

- **Mitigation**: Validating mechanisms must be established and applied before the plug-in is added to the registry. For instance, the plug-and-play environment can define a typed data structure that specifies the expected
data type in each field within the configuration file. Each raw field in the configuration file is converted to the corresponding field in the typed data structure. This typed data structure is validated, in which a validator properly escapes and neutralizes any code injected in one of the configuration file’s fields. Then, if the validation was successful, this typed data structure is passed to the initialization routine that will perform the necessary tasks for adding the plug-in to the plug-in’s registry.

– **Evidence**: We observed six CVEs that were caused by this problem. Through our SLR, we did not discover any research paper that identified this type of vulnerability.

4.0.1.4 Improperly checking the origin of an install request

This vulnerability occurs when the plug-and-play environment accepts install requests initiated either by the user or an external entity (i.e., a remote install), but it does not check (or incorrectly checks) the source of an install request.

– **Consequences**: A malicious plug-in can be installed in the plug-and-play environment.

– **Mitigation**: It can be fixed through defining a white list of the trusted remote sources that are allowed to trigger an install.

– **Evidence** We identified five CVEs that are a result of this plug-and-play
vulnerability type. Through our SLR we did not find any research paper that identified this type of vulnerability.

4.0.2 Plug-in Updates

*Updating plug-ins* is a basic feature in a plug-and-play system. It allows third party developers to provide new features and bug fixes for their plug-ins. In this context, we found the following types of problems:

4.0.2.1 Elevation of privilege through a plug-in update

This vulnerability occurs when plug-ins specify a list of privileges upon install and the user accepts these permissions. However, the plug-and-play environment does not check for the changes in privileges of plug-ins after an update, thus bypassing the user’s consent. Therefore, a plug-in can elevate its permissions through a plug-in update, and without user consent.

- **Consequences**: This vulnerability allows users to accept the installation of a seemingly innocuous extension. However, after an update the permissions are elevated, and this can be leveraged to perform malicious activities.

- **Mitigation**: Permissions need to be enforced at lifetime: once a user confirms a set of permissions during install, these cannot be elevated at any period of time.

- **Evidence**: Although we found only one case in our analysis, similar prob-
lems related to updates have been discussed by other authors in the literature. In a study of Firefox extensions, the authors point to the fact that Firefox did not previously perform integrity checks on the extensions after they had been installed, including after updates where these extensions could elevate their privileges or become malicious in other ways [56].

4.0.3 Plug-in Registry Management

It is important for a plug-and-play system to keep a registry of installed plug-ins. However, with this registry, the following types of problems were found to occur:

4.0.3.1 Extraction/Storage of Plug-in with world readable/writable permissions or in unsafe directories

In general, plug-ins are released as software bundles (e.g., zip files) that are extracted by the plug-and-play environment. When the plug-and-play environment extracts and/or stores these bundles using world-readable (or writable) permissions (e.g. 777 permissions in Unix-based operating systems), any other plug-in or potentially external process can alter plug-ins’ data or code.

- Consequences: An insecure storage of plug-ins means that trojans could modify plug-in data, alter plug-in functionality, or execute unauthorized code.
- **Mitigation**: Each plug-in’s bundles (configurations, scripts, binaries and other related artifacts) must be stored in a read-only storage dedicated to that plug-in.

- **Evidence**: In our analysis we found three cases of unsafe extraction/storage of plug-ins. Through our SLR we found a work by Birsan *et al* (pg. 5) [14], that argues that defining a secure shared location can be particularly challenging in a multi-user environment. “The new plug-ins, private to that user, cannot be installed in the read-only, shared install location, so the product should allow users to install and configure extra plug-ins in a location where they have more privileges.”

### 4.0.4 Plug-and-Play Execution Environment

In regards to the plug-and-play execution environment, we found the following types of problems:

#### 4.0.4.1 Lack of compartmentalization of plug-ins

This vulnerability type is caused by a lack of a well-defined logical compartment to isolate plug-ins from each other and from the plug-and-play environment.

- **Mitigation**: There are two complementary ways of fixing this vulnerability. The first one is to create logical compartments and make operations outside that compartment limited and intermediated by the plug-and-
play environment. The second approach is to create isolated object domains, as discussed in Section 4.0.4.7.

- Evidence: This vulnerability type appeared in 23 vulnerability reports. In our SLR, we found studies that reflected issues related to the lack of compartmentalization in Telematic Control Units in automobiles[26] and in extensions of Python [63]. Others [13] developed a tool to detect such issue.

4.0.4.2 Lack of fine-grained and modular permission setting

Many vulnerabilities observed in our analysis were due to benign plug-ins that had more privileges than needed to implement their features. A fine-grained and modular permission setting could have limited the access of such plug-ins.

- Consequences: Over-privileged plug-ins can expose the plug-and-play environment to a full compromise. Through a vulnerability in the plug-in, attackers would have access to any functionality and data of the plug-and-play environment.

- Mitigation: Modularize the plug-and-play environment into different fine-grained related functions with specific privileges. Plug-ins will gain access to specific functions, but not all.

- Evidence: Twenty four vulnerabilities were caused by this lack of a modular permission setting. In our SLR we found a study of over-privileged
extensions in Firefox [10, 38, 56]. The lack of fine-grained permissions rendered the Telematic Control Unit vulnerable to full control from all attackers that can obtain access to the USB port [26].

4.0.4.3 Allowing a plug-in to elevate its permission by manipulating (or delegating a task to) a process in the plug-and-play environment that has higher privileges

The vulnerability arises from the scenario in which a plug-in, executing in a unprivileged process, tampers with a high-privileged process in order to escape its security boundaries.

- **Consequences**: A malicious plug-in could execute arbitrary code in higher privileged context.

- **Mitigation**: This problem can be mitigated by limiting plug-ins’ exposure to high privilege plug-and-play APIs. Furthermore, OS system calls to other processes running in the underlying operating systems must be prevented. This means leveraging a mechanism that intermediates any system call between the plug-in and the underlying OS. The second mitigation technique is to limit the access to higher-privileged APIs.

- **Evidence**: There were three instances of CVEs in our dataset. In our SLR we found one paper discussing this issue [38].
4.0.4.4 Improper object access control and compartmentalization enforcement

When plug-ins are isolated in different logical compartments, they communicate with each other through object proxies that enforce compartment’s access policy. Security issues can occur when the plug-and-play environment uses an incorrect proxy for the inter-compartments communication.

- **Consequences:** Each plug-in may have a different set of permissions to use certain functionality; therefore, when their object proxies are incorrect, a lower privileged plug-in may leverage the proxy of a higher-privileged plug-in to elevate its privileges and execute arbitrary code. It can also be used as a mechanism to disrupt the plug-and-play execution environment.

- **Mitigation:** The plug-and-play environment *enforces security policies through object wrappers*, which act as proxies for a real object residing in a different compartment. These wrappers are instantiated and used according to the relationship between the caller and the callee compartments. Each type of object wrapper enforces a different type of security policy, which indicates the properties and operations would get accessed by the callee compartment.

- **Evidence:** We have observed five CVEs caused by the incorrect usage of object proxies. In our SLR we found a study that used crafted malicious Firefox extensions [56], to show how they can gain access to any DOM
structure. Authors recommended limiting plug-ins’ DOM access. A similar solution relying on named-based access control is recommended in [40] to keep untrusted software, i.e. extensions, from tampering with the plug-and-play environment. We also encountered studies that analyzed detection of insecure components in general [39], and studies that dealt with fine-grained security policy that would limit the access control of plug-ins specifically [53].

4.0.4.5 Unsanitized plug-in data

The core application interacts with data from the plug-ins. Security problems arise when the plug-and-play environment trusts data from the plug-in and, therefore, it does not properly sanitize the data.

- Consequences: this vulnerability results in a number of issues that are resultant of improper validation of inputs, such as cross-site scripting (XSS), stealing credentials, code injections (e.g. SQL injection), arbitrary code execution, memory corruption, and crashes.

- Mitigation: Adoption of an input validation mechanism that intercepts and sanitizes the data flowing from plug-ins to the core application.

- Evidence: We found 56 CVEs in this category. In our SLR, we found several papers that studied various individual plug-ins of WordPress. Several plug-ins have been found to be vulnerable to XSS and SQL injection
attacks [19, 20, 22, 44, 49, 51, 60, 64, 66]. These papers also discussed ap-
proaches to make each individual plug-in more secure. In contrast, our
approach identified mechanisms to secure the plug-and-play environment
and to lessen the impact of such vulnerable plug-ins.

4.0.4.6 Improper origin check of requests by plug-ins

This problem can result in a security breach when the plug-and-play
environment fails to correctly check the origin of requests (i.e., who was the
plug-in that initiated a call), therefore allowing the elevation of privilege at-
tack.

- **Consequences**: Since the application failed to verify which plug-in made
  a high-privileged API call, the plug-in would be able to perform actions
  beyond what should have been allowed. This can result in data leakage to
  unintended plug-ins, arbitrary code execution, same-origin policy bypass.

- **Mitigation**: Each plug-in must be assigned a unique identifier that acts
  as an origin identifier. Then, the plug-and-play environment must check
  the origin of requests against a security policy whenever a new incoming
  request is made to the plug-and-play execution environment.

- **Evidence**: There were 13 CVEs caused by this issue. The importance of
  checking the origin request is emphasized in [50], as a pre-condition to
  keeping the OS kernel secure from malicious extensions as well.
4.0.4.7 Improper isolation of objects used by plug-ins in the plug-and-play environment

plug-ins attach to the plug-and-play environment through well-defined public interfaces/APIs provided by the plug-and-play environment. The interaction of Plug-ins and the plug-and-play environment is through these APIs. Security problems can occur when plug-ins and the plug-and-play environment share the same objects or data structures of these APIs. As a results, plug-ins can interfere with the plug-and-play environment or other plug-ins.

− Consequences: Since multiple plug-ins can change the properties of these objects/data structures, it can negatively interfere with other plug-ins or with the plug-and-play environment. As a result, these shared objects create a channel that can be used to leak data or to alter the execution logic of other plug-ins. They can also introduce race condition problems leading to crashes of the plug-and-play environment.

− Mitigation: The plug-and-play environment can implement an isolated object domain solution. Each plug-in (and so the plug-and-play environment) must have its own copies of objects that are passed to or returned by an API call. Then, the plug-and-play environment manages these objects, ensuring that pointers or “object references” used by a plug-in, are not pointing to objects in another compartment (plug-and-play environment or other plug-ins).

− Evidence: We found twelve cases of shared objects being used as an
attack vector. In the literature, we found papers that discuss using this type of isolated object domain to prevent this vulnerability [33, 50, 56, 57] or that the lack of it causes this vulnerability [38]. In [50], authors discussed that the solution to isolate OS kernel extensions is to have them communicate with the kernel through only a set of defined, exported functions and to keep them from direct access to kernel data and code.

4.0.5 Plug-ins Request Handling

We found the following types of problems relating to plug-in request handling:

4.0.5.1 Reentrant event callbacks

Plug-ins can interrupt the execution of the event dispatching mechanism before it has finished, resulting in an unpredictable state. Given that multiple events may arrive and need to be dispatched to many plug-ins, it is important to ensure that the callback mechanism in the plug-and-play environment performs these operations in an atomic fashion.

- *Consequences*: The plug-and-play environment can crash or be left in an unexpected state.

- *Mitigation*: Dispatch of the event to plug-ins must be atomic such that it guarantees the integrity of the plug-and-play environment (avoid leaving the plug-and-play environment in an invalid state).
Evidence: We found five vulnerability reports in this category. In our SLR, we found a plug-and-play environment for automotive control systems that [27] used atomic operations for reading and writing data in order to avoid re-entry issues.

4.0.5.2 Plug-ins requests are handled without authorizing plug-ins that initiate the request

Vulnerabilities arise when the plug-and-play environment accepts any call from plug-ins without checking whether the plug-in is authorized to make such an API call.

Mitigation: There are various mitigation techniques. First, upon a request, the plug-and-play environment must authorize the plug-in that initiates a request or subscribes to an event. Second, events must be decomposed into sensitive and non-sensitive events. Listeners can subscribe to sensitive events if and only if they have enough permissions. Third, events and APIs specific to the plug-and-play environment must be hidden from plug-ins.

Evidence: Five CVEs were traced to this issue. In our SLR we did not find any research paper that identified this type of vulnerability.
Chapter 5

Evaluation of Data Collection / Analysis

The process under which the theory has been developed can evaluate the quality of a theory [17]. In Section 3, we described our research process and sample outcome of each step in details. We rigorously followed the classical version of the grounded theory. In each step, we reviewed our process to assess any deviation. A web-based toolkit with advanced search capability was developed for highlighting the key-concepts, conducting the open and selective coding and generating categories. Memos were shared through a separate Google document to facilitate concurrent feedback and revision of ideas. All steps were conducted iteratively: the theoretical sampling of the data proceeded as we created new catalogs or developed new memos. Additional data points were collected to help explore the analyst’s hypotheses.

Corbin and Strauss in their seminal article [17] discussed a set of scientific canons that are particularly important for qualitative research. We discuss the fitness and reliability of our results under these scientific canons: **Validity, Reliability, and Credibility** of the data. The data collected for the study included actual vulnerabilities and fixes to those vulnerabilities from large-scale popular open source projects. Furthermore, we used several projects from six
software domains to conduct our study. **Plausibility** of theory: In this work theory is defined as a set of vulnerability concepts (plug-and-play vulnerability, context, causes, consequences and mitigation techniques). Since the findings are driven from actual systems, the plausibility of theory is accurate, reflecting the underlying data. **Value** of the theory: To the best of our knowledge, there has not been a previous study of plug-and-play vulnerabilities. Plug-and-play systems are taking over various application domains, such as Web browsers, Internet-based apps, operating systems, middlewares, integrated medical systems, automotive systems, video surveillance and so on [6–8, 27, 28, 35, 50, 61]. The findings of this paper can help developers in these domains better design and implement plug-and-play systems and avoid common critical vulnerabilities. **Reproducibility** of the theory: We have released all our data, including the intermediary key points, codes, categories, memos, and their association with the underlying CVE data. This will enable the reproducibility of our findings, and we believe by following the same general rules for data collection and analysis, plus similar conditions, another investigator should be able to arrive at the same general scheme. **Empirical grounding** of the research findings: The results of this study are tied to empirical observations and concrete data from several case studies.
Chapter 6

Formal Modeling

6.1 Overview of Approach

We used formal modeling in order to verify that our proposed mitigation techniques to common vulnerabilities in plug-and-play architectures were indeed secure. Architectural description languages can be used to model architectural components of a software system. Tools have also been developed in order to verify whether the provided architecture meets a specific style. [52] We used Architecture Analysis and Design Language (AADL) to model our architecture. Then, we used the Assume Guarantee REasoning Environment (AGREE) to verify the behavior of our models. The declarations of all of our models can be found in Appendix A.

6.1.1 AADL

We used Architecture Analysis and Design Language (AADL) to model our architecture. AADL is an architectural description language allows for analysis to be performed on the models created. AADL uses component types to specify a static representation of the system architecture. This can be used to model ”software functionality, software runtime specifications, execution hardware, hardware and protocols used for connections, and related compo-
ponents such as sensors and actuators”. [25] OSATE is a tool used to create AADL and perform analysis. AADL can be used to describe a system at many levels, all the way down to the hardware if necessary. Each layer is made up of subcomponents from lower levels. This allows us to describe our architecture in greater detail where necessary, and leave other sections more abstract to keep them generalizable.

6.1.2 AGREE

We used the Assume Guarantee REasoning Environment (AGREE) to verify the behavior of our models. AGREE is compositional, attempting to prove that properties about one layer of the architecture are true based on properties provided by subcomponents. First, assumptions are written about each component. These are the things that the component is expecting from the environment. For example, a plug-in may expect that some data is sent to it from the core.

Next, guarantees are written, which provide information about what a component promises to do, given that the assumptions have been met. For example, our simple plug-in may return the data given to it by the core.

These assumptions and guarantees are added to each component in the form of an annex. When the top level system is analyzed, the subcomponents are checked to ensure that the assumptions and guarantees about each subcomponent have been met.

This allows us to create a generalizable model where plug-ins and the
core are explained, and third party developers can configure their top level systems as they please. They can then analyze their top level system, and confirm that the subcomponents used are secure with their configurations.

During the verification phase, AGREE verifies that none of the assumptions contradict one another, and that given the assumptions are met, the guarantee statements can also be met.

6.2 Basic Example

In the following basic example, we demonstrate how AGREE can validate the assume guarantee statements on the diagram. We model a system with a very basic core and two plug-ins. We guarantee that the output of pluginA will be the same as the input of pluginA. In the core, we assume that the event dispatched to plug-in1 is going to be received in the callback from pluginA. When we build the top level system, we initially properly connect the components so that the input and output for the plug-ins match. Figure 6.1 shows the model of the properly connect system and Figure 6.2 shows the AGREE verification.

However, in the second example, we improperly connect the components so that the input from pluginA and pluginB are mixed. In this case, we see that our AGREE verification has failed, because the input from pluginA was not the output that was sent to pluginA. Figure 6.3 shows the model of the improperly connect system and Figure 6.4 shows the AGREE verification.
6.3 Step 1: Model the Plug-in

We began our modeling by creating the components required for our reference plug-in. From our architecturally significant requirements, we found that plug-ins are composed of the following basic components: Initializer,
Updater, Uninstaller, and Business Logic. The below figures show the code used to generate the model of the plug-in component, as well as the visual representation of that model.

These components serve as the base components that we believe should be present in a plug-in. However, individual developers are welcome to add separate components to the plug-ins as well. This allows for maximum flexibility so that our method will work in the general case, allowing for use in multiple domains and implementations. Figure 6.5 shows the completed plug-in model.
6.4 Step 2: Model the Core

Next, we created a model for our core. Using our architecturally significant requirements we determined that the main components of the core
Figure 6.4: Basic System incorrectly connected and verified

are the Plug-in Registry, Plug-in Manager, and Business Logic. We use these components to our implementation of the core. However, external developers can adapt our model to fit their desired needs. We provide each component of the core that we believe to be necessary for our implementation of a secure plug and play system. However, in other domains, or for specific projects, a developer may need to provide more details about their core, or may need to make modifications. This is permitted through extending our components or through the addition of more components types. The model generated for our core is shown in Figure 6.6

<table>
<thead>
<tr>
<th>Property</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>▼ Verification for Core_TO_B</td>
<td>2 Invalid, 5 Valid</td>
</tr>
<tr>
<td>▼ Contract Guarantees</td>
<td>2 Invalid</td>
</tr>
<tr>
<td>▶ Core assume: C output range</td>
<td>Invalid (0s)</td>
</tr>
<tr>
<td>▶ Subcomponent Assumptions</td>
<td>Invalid (0s)</td>
</tr>
<tr>
<td>▶ ✓ This component consistent</td>
<td>1 Valid</td>
</tr>
<tr>
<td>▶ ✓ PluginA consistent</td>
<td>1 Valid</td>
</tr>
<tr>
<td>▶ ✓ PluginB consistent</td>
<td>1 Valid</td>
</tr>
<tr>
<td>▶ ✓ Core consistent</td>
<td>1 Valid</td>
</tr>
<tr>
<td>▶ ✓ Component composition consistency</td>
<td>1 Valid</td>
</tr>
</tbody>
</table>
6.5 Verification

Central Install Point: In order to address the vulnerabilities related to incorrect user notification of plug-in permissions and bypassing user notifications for plug-in installation, which could result in the consequences of elevation of privileges, spoofing, user assisted attacks, and stealth installation of malicious plug-ins, we previously discussed creating a centralized installation point for all plug-ins.
By design, our plug-in implementation contains a central point of installation. All plug-ins that want to be added to the plug-in registry pass through our initialization component. In order for events to be dispatched to the plug-in, they must be registered in the core registry component, so that the core's business logic component knows to dispatch the event.

6.5.1 Plug-ins are kept up to date

Ensuring that the most up to date versions of plug-ins are installed can help keep the core secure. This can be done by having the plug-in manager be responsible for checking for installed plug-ins with available updates.

This begins by ensuring that the most up to date version of a plug-in is installed. We verify this information by providing the assumption the
plug-in manager will find version 3 of the given plug-in. We then provide the assumption in our plug-in that the version is 4. Then, we see in the AGREE verification that this is permitted.

Next, we provide a plug-in where the version is only 2. In this case we are not able to guarantee that the plug-in is up to date and we see that the verification fails.

We also provide verification that the current plug-in is up to date. The core registry confirms the proper plug-in version number and ensures that the plug-in updater also reflects the same number. We also verify that the plug-ins are always updated rather than downgraded and that an older, potentially vulnerable extension is not used to replace a current version.

### 6.5.2 Plug-in Request Handling

In order the address the vulnerability category of plug-in requests being handled without authorizing plug-ins that initiate the request, which has been shown to lead to plug-ins tampering with other plug-ins, data leakage to unintended plug-ins, and arbitrary code execution, we previously recommended three approaches.

The first approach is authorizing the source of the request. Upon request, a plug and play core must authorize the plug-in that initiates a request or subscribes to an event. This is made possible through our connection between the business logic of the core and the core registry. Only installed plug-ins that have been validated against the plug-in manager will be allowed
to subscribe to events, as the list of plug-ins subscribed to an event is also contained within the registry.

The seconds mitigation proposed is decomposing events into sensitive and non-sensitive events. Listeners should be allowed to subscribe to sensitive events if and only if they have enough permissions. This is made simple because the plug-in registry is responsible for both managing the permissions that plug-ins have and also the events that they are subscribed to.

The third part of the mitigation supplied for the category of vulnerabilities is to hide plug and play internal events from the plug-ins. This is the accomplished by only providing data to plug-ins that involve the events they are subscribed to.

We are able to verify the first approach by assuming that a token is passed along with the business logics event dispatch and is returned by the plug-in during the event callback. This allows us to assume in the core that the plug-in will return the same token in the callback that it was passed during the event dispatch. In our plug-in, we guarantee that the token sent in the dispatch is the same as the one in the callback.

### 6.6 Case Study: Modeling a Full System

To model the full system, a third party developer could plug and play our existing components.

Developers can adapt this model in order to meet their own specific
needs, allowing our work to apply to the general case, and thus assist developers of plug-and-play systems in any domain. Some examples of places this could be used are the automotive industry when developing self driving cars, or the medical domain such as with implantable medical devices and their ecosystems.

Automotive systems are frequently being built so that telematic control units (TCUs) can be added to the electronic control units in order to add additional functionality after market. Some examples of these types of systems are GM’s Onstar and Progressive Snapshots. In this case study, we will use the hypothetical system Offstar. [26]

![Verification for KarzInc Base Car](image)

Figure 6.7: Verification for KarzInc Base Car

In the first example, a developer might want to model a base car using
our provided core and plug-in models. This example makes use of the components that had been defined previously in order to build a top level system. Figure 6.7 shows that the verification completes with no errors.

In addition, a developer could also modify our core and plug-in architectures in order to have more fine grained control over the architecture of their system. Our verifications will still work, provided the developer ensures that they provide the appropriate guarantees in their components. This allows developers to have control over their architecture, while also forcing developers to think about the requirements for security and the ways that they will mitigate potential vulnerabilities.

Figure 6.8 shows how the a car might be connected to a telemetry device. This example makes use of the components that had been defined previously in order to build a top level system.

The custom car cannot be verified that the plug-in is returning the same token that was sent to it during the event dispatch. This indicates to the developer that they will need to do additional work to ensure that their system is secure. This can be done by providing guarantees within the custom models that ensure that the plug-ins will be properly authenticated.
Figure 6.8: Verification for a custom car that does not take into account all mitigations
Chapter 7

Threats to Validity

In this section we discuss construct, internal and external threats to the validity of this work [55] and how we have mitigated them. **Construct validity** is concerned with the degree to which the measurements support the findings and results of what we were investigating. In our context, this type of threat is related to whether the measures we have taken for identifying plug-in based vulnerabilities were accurate enough to back up our findings. The first threat in this category relates to the quality of the data we collected. In this study, we used reports from the NVD, which is a well-known and widely used repository of vulnerabilities. Moreover, we also collected information directly from issue tracking systems and source code repositories, which represent the actual developers’ insights and mindset when fixing problems. Therefore, we consider that these data sources correspond to the highest quality we could achieve for this type of work. The second threat concerns our automated filtering approach could miss true positives. To mitigate this threat we applied three complementary filtering approaches in order to increase our recall. **Internal validity** reflects the extent to which a study minimizes systematic error or bias so that a causal conclusion can be drawn. One of the main threats to the internal validity of the research is the extensive manual analy-
ysis of CVE reports to observe patterns of incidence of vulnerabilities in these plug-and-play systems. Such manual analysis can be prone to biases. However, to mitigate this threat, our constant comparative analysis and memos helped us to elaborate on the reasonings behind our codes. Moreover, our analysis process encompassed five individuals with security background. **External validity** refers to the extent to which our results are generalizable and applicable to other extensible software. One threat is related to limited number of applications used in our study. In this study we have not covered applications from energy, medical or automotive domains, however, the results of SLR confirms that our findings are supported by existing ad-hock studies and can be expanded to those domains.
Chapter 8

Conclusion and Future Work

8.0.1 Future Work

Our study could be expanded by providing more verification in our formal modeling. Currently, we demonstrate the benefit of using formal modeling in this situation by showing how we can verify a few specific properties. However, this could be expanded to verify many more properties about the models so that the system is more secure.

This work could also built upon by analyzing the effectiveness of using formal modeling to see whether developers were more likely to implement security requirements that were discovered by formal modeling.

8.0.2 Conclusion

We contributed an in-depth discussion of the common types of vulnerabilities found in plug-and-play architectures based on data from several widely used projects. In addition, we provide an empirically grounded presentation of architectural mitigations that can minimize the impacts of common security issues. Furthermore, we used formal modeling to verify whether an plug-and-play system is secure against commons types of vulnerabilities.
Appendices
Appendix A

Filtered CVE Appendix

A.0.1 Model Declarations
package pluginSystemExample
public
  with Base_Types;

system plugin_initializer
features
  Version: out data port Base_Types::Integer;
  annex agree{**
    guarantee "Version is the installed version": Version = 3;
  **};
end plugin_initializer;

system plugin_updater
features
  Version: out data port Base_Types::Integer;
  annex agree{**
    guarantee "Version is up to date": Version = 3;
  **};
end plugin_updater;

system plugin_uninstaller
features
  Version: out data port Base_Types::Integer;
end plugin_uninstaller;

system plugin_BusLogic
features
  TokenReceived: in data port Base_Types::Integer;
  TokenReturned: out data port Base_Types::Integer;
  annex agree{**
    guarantee "TokenReturned is the same as Input": TokenReturned =
    TokenReceived;
  **};
end plugin_BusLogic;

system plugin
features
  OutputInit: out data port Base_Types::Integer; --init can contain the id of the plugin, which is looked up by the core to make sure it is not blacklisted
  OutputUpdate: out data port Base_Types::Integer;
  OutputUninstall: out data port Base_Types::Integer;
  BusinessIn: in data port Base_Types::Integer;
  BusinessOut: out data port Base_Types::Integer;
  annex agree {**
    guarantee "Installed version is the most up to date version":
    OutputInit=3 and OutputUpdate=3;
    guarantee "the plugin will return the token in the callback that it receives in the dispatch": BusinessIn = BusinessOut;
    guarantee "Install the most up to date plugin": OutputInit =
    OutputUpdate;
  **};
end plugin;

system implementation plugin.Impl
subcomponents
  initializer: system plugin_initializer;
  updater: system plugin_updater;
  uninstaller: system plugin_uninstaller;
  businessLogic: system plugin_busLogic;

connections
  Init: port initializer.Version -> OutputInit;
  Update: port updater.Version -> OutputUpdate;
  Uninstall: port uninstaller.Version -> OutputUninstall;
  BusIn: port BusinessIn -> businessLogic.TokenReceived;
  BusOut: port businessLogic.TokenReturned -> BusinessOut;

end plugin.Impl;

system core_registry
  features
    input_init: in data port Base_Types::Integer;
    input_updater: in data port Base_Types::Integer;
    input_uninstaller: in data port Base_Types::Integer;
    outputManager: out data port Base_Types::Integer;
    inputManager: in data port Base_Types::Integer;
    outputBus: out data port Base_Types::Integer;

annex agree{**
  assume "up to date plugin is installed": input_init <= input_updater;
  guarantee "most recent plugin is installed during init": outputManager =
  inputManager;
**};
end core_registry;

system core_pluginManager
  features
    input: in data port Base_Types::Integer;
    output: out data port Base_Types::Integer;
    PluginID :out data port Base_Types::Integer;
    PluginVersion: in data port Base_Types::Integer;
end core_pluginManager;

system core_BusinessLogic
  features
    eventCallbacks: in data port Base_Types::Integer;
    eventDispatches: out data port Base_Types::Integer;
    inPluginList: in data port Base_Types::Integer;

annex agree{**
  assume "Plugins send the same id back that you sent them": eventCallbacks
  = eventDispatches;
**};
end core_BusinessLogic;

system CustomCar_BusinessLogic
  features
    eventCallbacks: in data port Base_Types::Integer;
    eventDispatches: out data port Base_Types::Integer;
    inPluginList: in data port Base_Types::Integer;

annex agree{**
assume "Plugins do not send the same id back that you sent them":
eventCallbacks + 1 = eventDispatches;
**);
end CustomCar_BusinessLogic;

system Core
features
  InputInit: in data port Base_Types::Integer;
  InputUpdate: in data port Base_Types::Integer;
  InputUninstaller: in data port Base_Types::Integer;
  OutputEventDispatch: out data port Base_Types::Integer;
  InputEventCallbacks: in data port Base_Types::Integer;
  InputCheckVersion: in data port Base_Types::Integer;
  OutputPluginID: out data port Base_Types::Integer;
annex agree {**
  assume "Plugins send the same id back that you sent them":
  InputEventCallbacks = OutputEventDispatch;
  assume "plugins installed are up to date":
  InputCheckVersion = InputInit;
  assume "plugins updated when needed": InputCheckVersion =
  InputUpdate;
  **};
end Core;

system implementation Core.Impl
subcomponents
  coreRegistry: system core_registry;
  core_BusinessLogic: system core_BusinessLogic;
  coreManager: system core_pluginManager;
connections
  EventDispatch: port core_BusinessLogic.eventDispatches ->
  OutputEventDispatch;
  EventCallback: port InputEventCallbacks ->
  core_BusinessLogic.eventCallbacks;
  Uninstaller: port InputUninstaller -> coreRegistry.input_uninstaller;
  Updater: port InputUpdate -> coreRegistry.input_updater;
 Initializer: port InputInit -> coreRegistry.input_init;
  plug2bus: port coreRegistry.outputBus -> core_BusinessLogic.inPluginList;
  plugm2r: port coreRegistry.outputManager -> coreManager.input;
  plugr2m: port coreManager.output -> coreRegistry.inputManager;
  checkPluginID: port coreManager.PluginID -> OutputPluginID;
  checkPluginVersion: port InputCheckVersion -> coreManager.PluginVersion;
end Core.Impl;

system CustomCore
features
  InputInit: in data port Base_Types::Integer;
  InputUpdate: in data port Base_Types::Integer;
  InputUninstaller: in data port Base_Types::Integer;
  OutputEventDispatch: out data port Base_Types::Integer;
  InputEventCallbacks: in data port Base_Types::Integer;
  InputCheckVersion: in data port Base_Types::Integer;
  OutputPluginID: out data port Base_Types::Integer;
annex agree {**
assume "Plugins send the same a different id back that you sent them": InputEventCallbacks + 1 = OutputEventDispatch;
assume "plugins installed are up to date": InputCheckVersion = InputInit;
assume "plugins updated when needed": InputCheckVersion = InputUpdate;
**};
end CustomCore;

system implementation CustomCore.Impl
subcomponents
  coreRegistry: system core_registry;
  core_BusinessLogic: system CustomCar_BusinessLogic;
  coreManager: system core_pluginManager;
connections
  EventDispatch: port core_BusinessLogic.eventDispatches -> OutputEventDispatch;
  EventCallback: port InputEventCallbacks -> core_BusinessLogic.eventCallbacks;
  Uninstaller: port InputUninstaller -> coreRegistry.input_uninstaller;
  Updater: port InputUpdate -> coreRegistry.input_updater;
  Initializer: port InputInit -> coreRegistry.input_init;
  plug2bus: port coreRegistry.outputBus -> core_BusinessLogic.inPluginList;
  plugm2r: port coreRegistry.outputManager -> coreManager.input;
  plugr2m: port coreManager.output -> coreRegistry.inputManager;
  checkPluginID: port coreManager.PluginID -> OutputPluginID;
  checkPluginVersion: port InputCheckVersion -> coreManager.PluginVersion;
end CustomCore.Impl;

system KarzInc
features
  Input: in data port Base_Types::Integer;
  Output: out data port Base_Types::Integer;
annex agree {**
  assume "System input range": Input = 3;
  **};
end KarzInc;

system implementation KarzInc.Impl
subcomponents
  Offstar: system plugin.Impl;
  BaseCar: system Core.Impl;
connections
  InputToCheckVersion: port Input -> BaseCar.InputCheckVersion
  {Communication_Properties::Timing => immediate;};
  Core_TO_AEvent: port BaseCar.OutputEventDispatch -> Offstar.BusinessIn
  {Communication_Properties::Timing => immediate;};
  A_TO_CoreInit: port Offstar.OutputInit -> BaseCar.InputInit
  {Communication_Properties::Timing => immediate;};
  A_TO_CoreUpdate: port Offstar.OutputUpdate -> BaseCar.InputUpdate
  {Communication_Properties::Timing => immediate;};
  A_TO_CoreUninstall: port Offstar.OutputUninstall ->
  BaseCar.InputUninstaller
  {Communication_Properties::Timing => immediate;};
A_TO_CoreEvent: port Offstar.BusinessOut -> BaseCar.InputEventCallbacks
{Communication_Properties::Timing => immediate;};
CheckVersionToOutput: port BaseCar.OutputPluginID -> Output
{Communication_Properties::Timing => immediate;};
end KarzInc.Impl;

system CarCo
features
  Input: in data port Base_Types::Integer;
  Output: out data port Base_Types::Integer;
  annex agree {{
    assume "System input range": Input = 3;
    **};
end CarCo;

system implementation CarCo.Impl
subcomponents
  Offstar: system plugin.Impl;
  CustomCar: system CustomCore.Impl;
connections
  InputToCheckVersion: port Input -> CustomCar.InputCheckVersion
  {Communication_Properties::Timing => immediate;};
  Core_TO_AEvent: port CustomCar.OutputEventDispatch -> Offstar.BusinessIn
  {Communication_Properties::Timing => immediate;};
  A_TO_CoreInit: port Offstar.OutputInit -> CustomCar.InputInit
  {Communication_Properties::Timing => immediate;};
  A_TO_CoreUpdate: port Offstar.OutputUpdate -> CustomCar.InputUpdate
  {Communication_Properties::Timing => immediate;};
  A_TO_CoreUninstall: port Offstar.OutputUninstall ->
  CustomCar.InputUninstaller
  {Communication_Properties::Timing => immediate;};
  A_TO_CoreEvent: port Offstar.BusinessOut -> CustomCar.InputEventCallbacks
  {Communication_Properties::Timing => immediate;};
  CheckVersionToOutput: port CustomCar.OutputPluginID -> Output
  {Communication_Properties::Timing => immediate;};
end CarCo.Impl;

end pluginSystemExample;
Bibliography


Vita

Taylor Noelle Corrello was born in New Jersey on May 5th, 1995, the daughter of Tami Ann Corrello. She is currently pursuing her Bachelor and Master of Science in Software Engineering at the Rochester Institute of Technology.

Permanent address: 813 Devon Estate Ave
Myrtle Beach, South Caroline 29588

This thesis was typeset with \LaTeX\ by the author.

\LaTeX\ is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth’s TeX Program.