Achilles' Heel of Plug-and-Play Software Architectures

Joanna C. S. Santos, Adriana Sejfia, Taylor Corrello, Smruthi Gadenkanahalli and Mehdi Mirakhorli

{jds5109,axs1461,tnc5484,sg1626,mxmvse}@rit.edu

Rochester Institute of Technology

Rochester, NY, USA

ABSTRACT

Through a set of well-defined interfaces, plug-and-play architectures enable additional functionalities to be added or removed from a system at its runtime. However, plug-ins can also increase the application's attack surface or introduce untrusted behavior into the system. In this paper, we ① use a grounded theory-based approach to conduct an empirical study of common vulnerabilities in plug-and-play architectures; ② conduct a systematic literature survey and evaluate the extent that the results of the empirical study are novel or supported by the literature; ③ evaluate the practicality of the findings by interviewing practitioners with several years of experience in plug-and-play systems. By analyzing Chromium, Thunderbird, Firefox, Pidgin, WordPress, Apache OfBiz, and Open-MRS, we found a total of 303 vulnerabilities rooted in extensibility design decisions and observed that these plugin-related vulnerabilities were caused by 16 different types of vulnerabilities. Out of these 16 vulnerability types we identified 19 mitigation procedures for fixing them. The literature review supported **12** vulnerability types and **8** mitigation techniques discovered in our empirical study, and also indicated that **5** mitigation techniques were not covered in our empirical study. Furthermore, it indicated that **4** vulnerability types and **11** mitigation techniques discovered in our empirical study were not covered in the literature. The interviews with practitioners confirmed the relevance of the findings and highlighted ways that the results of this empirical study can have an impact in practice.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; • **Software and its engineering** → *Software design engineering*.

KEYWORDS

Software Security, Plug-and-Play Design, Vulnerabilities

1 INTRODUCTION

Many application domains adopt plug-and-play [1] architectures to enhance their systems' extensibility, reusability and modifiability [8]. For example, the automotive industry is rapidly creating plug-and-play architectures in which software modules can be slotted into the overall electronic architecture without unexpectedly disrupting the other components [20]. In the finance domain, plug-and-play architectures provide universal APIs for in-store point of sale (POS) systems that allow users to plug in a variety of different applications. In medical device development, plug-and-play architectures enhance programs' and medical devices' interoperability, where third-party medical applications are plugged into networked medical devices to provide diagnosis, treatment, research, safety, and quality improvements, as well as equipment management features [3, 4]. 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 runtime infrastructure for plug-ins. *Plug-ins* provide bundled functionalities which can be added at runtime, making the software customizable and extensible. This means that the software product can be released early, and new features can be added later through the plug-ins. Moreover, plug-and-play architectures can enable contributions from third-party vendors because extending the architecture does not require access to the source code; instead, these third-party developers can implement well-defined public interfaces provided by the plug-and-play environment [10].

Although plug-ins are useful for adding new features to the software, they can increase the application's attack surface or introduce untrusted behaviors. 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 system's security and trustworthiness [10, 31]. There are numerous vulnerabilities reported for plug-and-play architectures [28, 32, 33, 38, 45]. A group of researchers 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 [18]. In this case, the plug-in was insecure; however, severe security and privacy issues could also occur when the system accepts malicious plug-ins [27].

Although there are numerous studies in the area of plug-and-play software architectures [1, 12, 16, 24, 57–59] and applications in various domains [5, 13, 20, 26, 47, 49], as well as domain-specific examples of such extensible architectures [7, 11, 46, 48], we currently

---

[1] In this paper, we use the term *plug-and-play* to refer to a wide range of applications that use extensible architectures. Other similar terms used are plugin-based, extension-based, app-based and etc.

lack an empirically grounded work that aims to study vulnerabilities that are prevalent in plug-and-play software architectures as well as mitigation techniques to prevent such vulnerabilities. Furthermore, currently known software vulnerabilities are either generic and not contextualized for specific domains (e.g. PnP) or are well characterized for commercial web-based systems.

The novel contribution of this paper lies in ① an in-depth implementation of grounded theory to *empirically study* vulnerabilities prevalent in PnP architectures and to *characterize* such vulnerabilities based on real data collected from widely used PnP systems. We use *classical grounded theory* [21] as a systematic inductive method for conducting qualitative research of software vulnerabilities in plug-and-play architectures. We chose this approach due to its emphasis on the emergence of concepts [22, 53], 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 is to allow the data to drive our process of discovering classes of plug-and-play vulnerabilities (inductive reasoning) rather than formulating hypotheses at the beginning of the analysis process (deductive reasoning). ② A systematic literature review that aims to map the results of the empirical study to the state-of-the-art and examine to what extent the results are supported by the literature or complements the existing body of work in this domain. This extensive literature review is conducted after examining real projects to prevent the influence of existing concepts on the emerging results. ③ Interviews with six practitioners with work experience of plug-and-play systems from the *health-care*, *business* and *industrial control system (ICS)* domains. The interviews were conducted at the latest stage of our study in order to examine whether the empirical findings can be useful in practice. Our data is released at: https://github.com/SoftwareDesignLab/AchillesHeel.

This paper is organized as follows: Section 2 provides an overview of the methodology used in this empirical study; Section 3 discusses the results of our empirical study; Section 4 compares our results with those from the literature; Section 5 presents our interviews with practitioners; Section 6 describes the threats to the validity of this work; and Section 7 concludes the paper.

2 METHODOLOGY

A *grounded theory* [23, 52] approach is a progressive identification and integration of concepts from data that leads to discoveries directly supported by empirical evidence. The classical grounded theory [21] 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* [22, 53]. Figure 1 shows how we applied the classical grounded theory to our research that was conducted over the period of one year.

2.1 Limiting the Phenomena Under Study

In following a grounded theory approach, researchers are advised against formulating a specific research question upfront; rather, they **define an area of interest** (i.e., the phenomena under observation). As illustrated in Figure 1, the focus of this empirical study is the *"Achilles' heel" (vulnerabilities) in plug-and-play systems*. We

focused on vulnerabilities specific to the plug-and-play architecture itself - i.e., security issues that are enabled due to the extensibility mechanisms provided by plug-and-play environments and are specific to such architecture.

2.2 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. Thus, we targeted data sources that were freely accessible to us. In this context, we focused on *open source systems* with a *plug-and-play software architecture*.

2.2.1 Theoretical Sampling. is the process of jointly gathering and analyzing data in order to decide what data needs to be collected next [21, 23, 53]. Our theoretical sampling started with 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 of 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, more generalizable concepts. 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 empirical study, as per the approach suggested by the classical grounded theory.

2.2.2 Data Sources. We used the *National Vulnerability Database (NVD)* to extract vulnerability meta-data, *Issue Tracking Systems* to obtain further discussions about the problem, *Source Code Repositories* to identify fixes for these vulnerabilities, and *Technical Documents* to 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 [36]. Vulnerabilities disclosed in NVD are assigned a unique Common Vulnerabilities and Exposures Identifier (*CVE ID*). Along with this identifier, NVD also provides a concise *description* of the problem, a list of affected *software releases*, and a list of Web sites (*references*) that contain more details about the problem.
- *Identifying vulnerability details from Issue Tracking Systems*: Although CVE reports provide a 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 or other information. Thus, we identified the URLs to the corresponding bug entry of the issue tracking system of the case study. This way, we read the developers' discussion about the problem and how they developed a solution. We leveraged the list of "references"
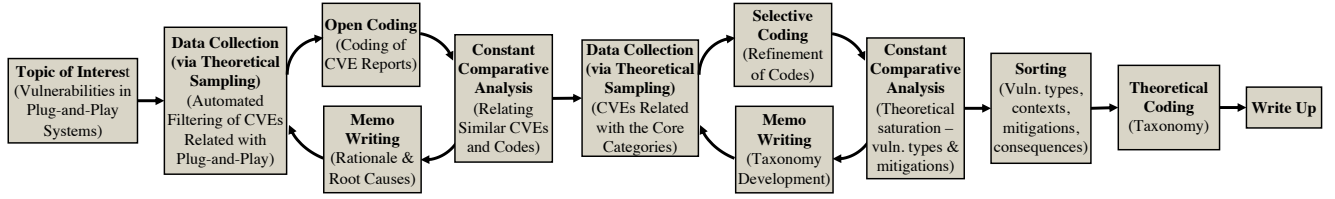
Figure 1: The Grounded Theory Approach Applied to our Work

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 the 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 contained the files that were affected (i.e., modified, added or removed) in the fix. Identifying patches help us to verify the solution applied by developers to repair the software.

- *Identifying 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 features and any security mechanism adopted for protecting their PnP environment. We also reviewed 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, and 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 PnP mechanism is implemented in the source code. Each project's trace matrix of PnP design decisions to source files was peer-reviewed.

2.2.3   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, such as issue tracking system reports and discussions as well as patches involved in the fix. Each of the artifacts contributed to a comprehensive understanding of the CVE under inspection. Figure 2 shows the information model of the vulnerability data we collected after performing the steps enumerated above. We used three complementary approaches to identify the subset of vulnerabilities (CVE instances) associated with the systems' plug-and-play architecture:

- *Component-Based Approach*: The issue tracking entries to fix CVEs often have an attribute indicating the *affected software*
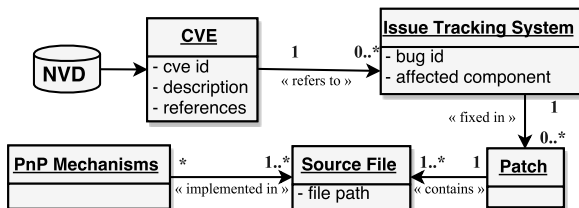
Table 1: Keywords used for automatically filtering CVEs

Case Study	Keywords
Firefox	extension, bundle, theme, add on, add-on, addon, plugin, plug-in, dictionary, xpi, pack
Chromium	extension, plug-in, plugin, app
Thunderbird	extension, bundle, theme, add on, add-on, addon, plugin, plug-in, dictionary, xpi, pack
Wordpress	plug-in, plugin, theme
Pidgin	plug-in, plugin
OfBiz	plug-in, plugin
OpenMRS	plug-in, plugin, add on, add-on, addon

component, which is declared by the original project developers. 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) to identify the subset of CVEs that were potentially related to securing their extensible architecture.

- *Keyword-Based Approach*: we established a list of keywords that reflected the terminology developers used to refer to the plug-and-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 1 enumerates the keywords used per case study.

- *File-Based Approach*: We used the traceability matrix of PnP mechanisms to source files developed during our data collection (Section 2.2.2) 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 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*"). In this manual review, we considered a vulnerability as plugin-related if it was caused by (i) a lack of mitigation procedures in the PnP core to stop misuses from malicious plug-ins or consequences of security bugs in benign plug-ins, (ii) an inappropriate design choice for mitigating the security issues (design-level), or (iii) an incorrect implementation of design decisions in the core (correct design-level choices, but an incorrect implementation in the code).

2.2.4   Data Preprocessing. After collecting, merging and filtering the vulnerability artifacts, we conducted a *preprocessing* step to summarize the data for us to start the *coding* of the data based on the

Figure 2: Information model for the collected data.

Table 2: Statistics of the vulnerability data used in this study.

Case Study	# CVEs	#Analyzed CVEs	#Plugin-related CVEs
Firefox	1396	156	68
Chromium	1252	169	73
Thunderbird	704	85	37
Wordpress	433	221	91
Pidgin	69	34	32
OfBiz	7	1	1
OpenMRS	4	1	1

grounded theory principles. The data preprocessing was performed by the authors, who systematically scrutinized the subset of CVEs (and its associated artifacts - see Figure 2) that were identified using the three complimentary automated approaches described in the previous step. The vulnerability reports were summarized by filling out a form containing specific sections for *context* (the underlying scenario in which the vulnerability occurred), *problem* (fine-grained root cause) and *solution* (how it was fixed).

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

2.3 Open Coding

After preparation of the data, the first step was the *open coding* [22, 23] of the vulnerability summaries. It consists of analyzing each incident (i.e., data point) in order to annotate them with codes (concepts). This coding was performed by the authors of this paper, whose software development experience level varied from 2-10 years. During this process, we analyzed each of the previously collected plugin-related CVE summaries and reviewed its *context*, *problem*, and *solution* (and any other details available in the issue tracking system or other sources as needed). After reviewing the CVE information, we 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.

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; it *"accounts for a large portion of the variation in a pattern of behaviour"* [21]. Please note that in further iterations many of these codes were grouped into core categories. Figure 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.* Each of these were assigned a code, example of **codes** generated for this summary are: *Not showing install warning dialog; Silent install of plug-ins; Block cross-origin install requests.*

2.4 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 a 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, we observed that some key points reoccurred; such key points were used to form the core categories. For instance, in Figure 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". Emerging concepts were then 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. We were either annotating the CVEs with existing tags or creating new ones (i.e., the existing tags are not suitable for the CVE being analyzed). For instance, in the case of CVE-2012-0934 (Figure 3) 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 was the identification **core categories** [21]. In our study, our core categories correspond to the *types of plug-and-play vulnerabilities*.

2.5 Memoing

Throughout the iterative process of *coding* and *constant comparative analysis*, the researchers captured their insights in **memos**. A shared Google Document with a predefined table was used to capture early insights. In this early stage of data analysis, our memos mostly concerned potential core categories (PnP vulnerability types). As the process continued, we finalized them by adding more detailed information about consequences and mitigation techniques. For these potential core categories, the memos would capture a *summary* of the plug-and-play vulnerability type, associated *consequences* and how they could be *mitigated*. Table 3 illustrates a sample memo captured by an analyst during the memoing process.

Table 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 PLUGINSPAGE 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.

2.6 Selective Coding

The **selective coding** of our methodology focused on theoretically saturating the architectural and related concepts. In this step we returned to the CVE instances that were associated with plug-and-play vulnerability types in order to further refine them, capturing all possible consequences observed in the data, their context of occurrence, and how developers mitigated them. In this step and later stages of our analysis, our **memos** encompassed development of a conclusion for the study. To do so, we focused on rearranging (merging or breaking) our core categories for establishing a

CVE-2015-4498	CVE-2011-3055	CVE-2012-0934
Description: 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.	**Description:** 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.	**Description:** A PHP remote file inclusion vulnerability in ajax/savetag.php in the Theme Tuner plugin for WordPress before 0.8 allows remote attackers to execute arbitrary PHP code via a URL in the tt-abspath parameter.
Problem: Normally, Firefox warns the user when trying to install an add-on if this install request was initiated by a Web page. The users must explicitly accept that the add-on 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 redirects 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.	**Problem:** An attacker was able to gain access to the extensions management page and have it 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.	**Problem:** A remote attacker could send a specially-crafted URL request to the savetag.php script using the tt-abspath parameter to specify a malicious file from a remote system. This allows the attacker to execute arbitrary code on the Web server.
Solution: Fix is to block cross-origin add-on install requests.	**Solution:** The fix is to generate the same prompts for packed and unpacked extensions. This also fixes an issue in which we were not prompting for unpacked extensions with plugins at installation time.	**Solution:** Fix was to remove the part of the code that leveraged on user-provided input to include PHP code.
Codes: "Not showing install warning dialog," "Silent install of plug-ins," and "Block cross-origin install requests".	**Codes:** "Not showing install warning dialog," "Silent install of plug-ins," "Consistent generation of install warning prompts".	**Codes:** "Arbitrary code execution," "File path traversal," "Remote code file inclusion".

Figure 3: Examples of open coding of the CVE summaries that are generated after data preprocessing.

cohesive taxonomy of *vulnerability types in PnP architectures*, the *context* in which they occurred, their corresponding *mitigations* and *consequences*.

2.6.1 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 the information retrieved for each vulnerability report (Figure 2) and enables the researcher to annotate the report and tag codes (i.e., concepts) to it.

2.7 Memo Sorting

At the final 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 was based on inter-related concepts. The goal of this step was to look at the data at a higher abstraction level.

2.8 Theoretical Coding

In the later stages of our analysis, we employed theoretical coding to interconnect substantive codes. In other words, we connect the discovered concepts, leading to the development of hypotheses that would shape the results of our empirical study. Theoretical coding involves applying a *coding paradigm* that helps researchers to inter-connect concepts derived from the data [23]. 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*, which are the contributing factors that lead to the vulnerability; *mitigations*, which are techniques to fix these issues; and the *consequences* which result from the vulnerabilities. For instance, *"auto updates"* and *"plug-in update"* codes were merged to form the context *"Plug-in Update"*; *"add checks for plug-in permission during updates"* and *"comparing against initial list of permissions"* formed the mitigation *"Lifetime enforcement of plug-in permission"*; codes *"privilege elevation"*, *"bypass privilege check mechanism"* formed the consequence *"Privilege elevation"*.

3 A TAXONOMY OF VULNERABILITIES IN PLUG-AND-PLAY SYSTEMS

The results of this empirical study are presented in the form of a **taxonomy of vulnerabilities in plug-and-play architectures**,

learned from seven open source systems. This taxonomy characterizes the *context* in which the vulnerability occur, its *mitigation* procedures, as well as its *consequences*.

3.1 Context: 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. In this context, we found the following types of problems (Figure 4):

3.1.1 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 PnP environment. This problem occurs when the PnP environment does not show to the user **all** of the data and/or functionality that will be accessed by the plug-in before the plug-in is installed.
— **Mitigation**: Devising a *central point for installation logic*. All the installation requests are guaranteed to go through this component, which is in charge of (i) consistently generating install warning prompts before any install requests; (ii) showing all the requested permissions.

3.1.2 Bypassing of user notification or restriction checking processes for plug-in installation. Whenever a new install is requested, the PnP environment should ask the user for consent to proceed (or abort) the install. This type of vulnerability is caused when the requirement that the user mediate all install requests is not strictly enforced.
— **Mitigation**: Likewise in "Incorrect user notification of plug-in permissions", the mitigation consists on designing a *Central point for installation logic* (see Section 3.1.1).

3.1.3 Lack of plug-in's configuration file sanitization. It is caused by not validating the plug-in's configuration file to verify whether it is structurally and semantically correct as well as to escape/neutralize any code that is injected in the plug-in's configuration file.
— **Mitigation**: Designing and implementing a *configuration validator* that performs typed parsing and configuration files validation to prevent these files to be used as an attack vector.

3.1.4 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 PnP

Figure 4: Vulnerability types, mitigations and consequences in the context of plug-ins installation

Incorrect user notification of plug-in permissions	Gain privileges, Spoofing, User-assisted attack	Central point for installation logic
Bypassing of user notification or restriction checking processes for plug-in installation	Stealth installation of malicious plug-ins	
Lack of plug-in's configuration file sanitization	Arbitrary code execution, Privilege elevation, Data leakage, Directory path traversal, Application crash	Configuration validator
Extraction/storage of plug-in with world-readable/writable permissions or in unsafe directories	Modify/Erase plug-in data, Replace a benign plug-in with a malicious one, Symlink attack	Dedicated secure Storage
Spoofing the origin of a plug-in install request	Hijack of origin of an install request	Secure exchange of plug-in resources
Improperly checking the origin of an install request	Misleading the user to install a plug-in, Spoofed origin of an install request	Whitelist of install origins

(Plug-ins installation)

environment. When the PnP 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 a potentially external process can alter plug-ins' data or code.
— **Mitigation**: Having a *dedicated secure storage* such that the application extracts the software bundle to the application's dedicated folder, that has restricted admin-only access.

3.1.5 Spoofing the origin of a plug-in install request. It is caused by not enforcing that the install request (followed by a transfer of the plug-in's resources) will be made through a secure communication channel (such as HTTPS). An intruder can conduct a man-in-the-middle attack to spoof the origin of an install request and being able to trick the user into installing a potentially malicious plug-in.
— **Mitigation**: Performing a *secure exchange of plug-in resources* by using a secure communication protocol (e.g. HTTPS).

3.1.6 Improperly checking the origin of an install request. It occurs when the PnP environment accepts install requests initiated either by the user or an external entity (i.e., a remote install), but it incorrectly checks the source (origin) of the install request.
— **Mitigation**: Having a *whitelist of install origins* that specifies safe install points and only allowing these to trigger installation. This avoids giving a malicious code or other attack vectors the ability to silently install a plug-in.

3.2 Context: Plug-in Updates

A PnP system allows that plug-ins are updated whenever a new version is available. In this case, the system applies the needed changes to the plug-in registry. In this context, we found the following vulnerability types (Figure 5):

3.2.1 Elevation of privilege through a plug-in update. It occurs when plug-ins specify a list of privileges upon install and the user accepts these permissions. However, the PnP environment does not check for the changes in privileges of plug-ins after an update, bypassing the user consent. Therefore, a plug-in can elevate its permissions through a plug-in update without user consent.
— **Mitigation**: Performing a *lifetime enforcement of plug-in permissions*. During a plug-in update, the application compares the current request permissions against the previous list of permissions provided by plug-in during install.

3.3 Context: Plug-and-Play Execution Environment

Since plug-ins are not executable (i.e., standalone programs), their execution environment is provided by the host application. The PnP application core is in charge of orchestrating the execution of multiple and concurrent plug-ins. During execution, the PnP application can be prone to the following security problems (Figure 6):

3.3.1 Lack of compartmentalization of plug-ins. It is caused by a lack of a well-defined logical compartment that isolates plug-ins from each other as well as from the PnP environment. In this case, plug-ins are allowed to directly communicate with each other and use the core's resources without appropriate restrictions.
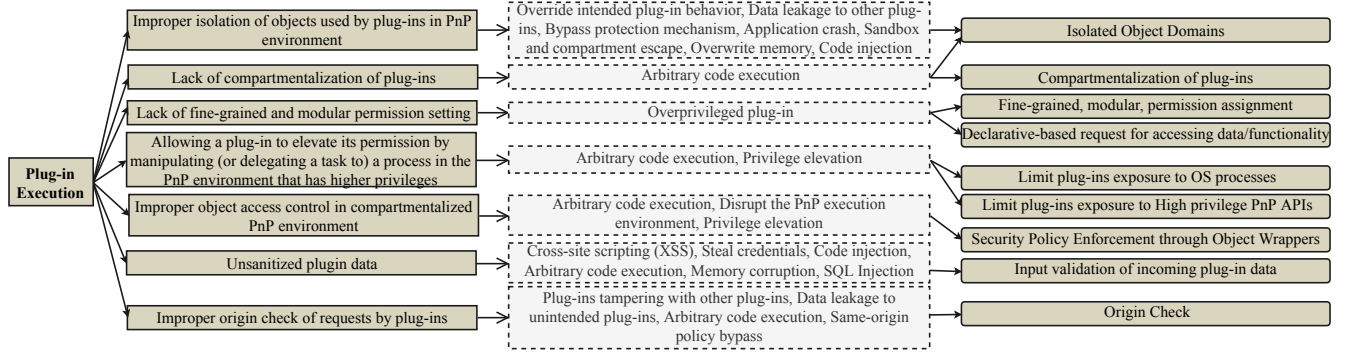— **Mitigation**: There are two complementary approaches to fix this problem. Ensuring a *compartmentalization of plug-ins* in which each plug-in is encapsulated in a separate compartment. Also having *isolated object domains* such that each compartment must have its own copy of objects for communication with the PnP environment. It minimizes the risk of the same object being used by another plug-in. As a result, the PnP environment manages these objects and enforces that these objects are not used as an attack vector.

3.3.2 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, and therefore minimizing the impacts of vulnerabilities in these plug-ins on the application core.
— **Mitigation**: The application host has a *fine-grained, modular, permission assignment* by creating logical groups of functionalities/data that are only available if a plug-in has the necessary permissions. In conjunction, the application implements a *declarative-based request for accessing data/functionality* in which the plug-ins have to explicitly indicate what features/data they intend to utilize such that the user grants them the permission to access those.

3.3.3 Allowing a plug-in to elevate its permission by manipulating (or delegating a task to) a process in the PnP environment that has higher privileges. : This vulnerability arises from the scenario in which a plug-in, executing in an unprivileged process, tampers with a high-privileged process in order to escape its security boundaries.
— **Mitigation**: The application *limits plug-ins exposure to OS processes* by leveraging a mechanism that intermediates any system call between the sandboxed child process and the underlying OS to prevent the low-privileged process to attempt to communicate with other higher-privileged processes. In addition, the application *limits plug-ins exposure to high-privilege PnP APIs*. As a result, the

Figure 5: Vulnerability types, mitigations and consequences in the context of plug-ins update

| Update Plug-ins | → | Elevation of privilege through a plug-in update | → | Privilege elevation | ← | Lifetime enforcement of plug-in permissions |

Figure 6: Vulnerability types, mitigations and consequences in the context of plug-ins execution

Plug-in Execution	Improper isolation of objects used by plug-ins in PnP environment	Override intended plug-in behavior, Data leakage to other plug-ins, Bypass protection mechanism, Application crash, Sandbox and compartment escape, Overwrite memory, Code injection	Isolated Object Domains
	Lack of compartmentalization of plug-ins	Arbitrary code execution	Compartmentalization of plug-ins
	Lack of fine-grained and modular permission setting	Overprivileged plug-in	Fine-grained, modular, permission assignment / Declarative-based request for accessing data/functionality
	Allowing a plug-in to elevate its permission by manipulating (or delegating a task to) a process in the PnP environment that has higher privileges	Arbitrary code execution, Privilege elevation	Limit plug-ins exposure to OS processes / Limit plug-ins exposure to High privilege PnP APIs
	Improper object access control in compartmentalized PnP environment	Arbitrary code execution, Disrupt the PnP execution environment, Privilege elevation	Security Policy Enforcement through Object Wrappers
	Unsanitized plugin data	Cross-site scripting (XSS), Steal credentials, Code injection, Arbitrary code execution, Memory corruption, SQL Injection	Input validation of incoming plug-in data
	Improper origin check of requests by plug-ins	Plug-ins tampering with other plug-ins, Data leakage to unintended plug-ins, Arbitrary code execution, Same-origin policy bypass	Origin Check

access that plug-ins have to higher privileged APIs provided by the core application should be limited according to the permissions they have asked for and the privileges they have.

3.3.4 Improper object access control and compartmentalization enforcement. When plug-ins are isolated in different logical compartments, they communicate with each other through object proxies. Each different type of proxy enforces a set of compartments' access policies. Security issues can occur when the PnP environment uses an incorrect proxy for inter-compartments communication.

— **Mitigation**: Performing a *security policy enforcement through object wrappers*. It adopts different types of object wrappers that act as proxies for a real object residing in a different compartment. These wrappers apply a security policy which enforces what type of properties and operations would get accessed by the callee compartment depending on the relationship between the caller and the callee compartments.

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

— **Mitigation**: Performing an *input validation of incoming plug-in data* such that any data transferred by plug-ins to the PnP environment are sanitized prior use.

3.3.6 Improper origin check of requests by plug-ins. It can result in a security breach when the PnP 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 attack.

— **Mitigation**: Performing an *origin check* to verify request origins and authenticate plug-ins requests against a security policy.

3.3.7 Improper isolation of objects used by plug-ins in the PnP environment. Plug-ins attach to the PnP environment through well-defined public interfaces/APIs provided by the PnP environment. The interaction of Plug-ins and PnP environment is through these APIs. Security problems can occur when plug-ins and the PnP

environment share the same objects or data structures of these APIs. As a result, plug-ins can interfere with the PnP environment or other plug-ins.

— **Mitigation**: Similar to Section 3.3.1, this vulnerability can be mitigated through *isolated object domains*.

3.4 Context: Plug-ins Request Handling

As part of plug-in execution, the application core has to handle plug-ins requests through API calls. In this context, we found the following vulnerability types (Figure 7):

3.4.1 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 a plug-in without checking whether the plug-in is authorized to make such an API call.

— **Mitigation**: Three complementary approaches can be used. The PnP host can *authorize the source of request*: Upon a request, PnP host must authorize the plugin that initiates a request or subscribes to an event. *Decomposing the events* into sensitive and non-sensitive events. Listeners can only subscribe to sensitive events if and only if they have enough permissions. Moreover, *hiding PnP internal events* such that any events and APIs specific to PnP environment must be hidden (inaccessible) from plug-ins.

3.4.2 Reentrant event callbacks. Plug-ins can interrupt the execution of the event-dispatching mechanism before it has finished, resulting in an unpredictable state.

— **Mitigation**: Designing an *atomic event dispatcher*. Given that multiple events may arrive at the application and need to be dispatched to many listener plug-ins, it is important to ensure that the callback mechanism in the PnP environment performs these operations in an atomic fashion.

4 ARE THE FINDINGS SUPPORTED BY THE LITERATURE?

After completion of the grounded theory, we conducted a Systematic Literature Review (SLR) to examine whether the findings of

Figure 7: Vulnerability types, mitigations and consequences in the context of plug-ins requests handling
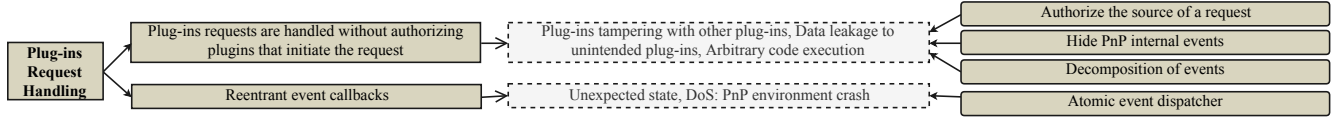
this empirical study are supported by the literature or complement the existing body of work.

4.1 Methodology

The search strategy [62] of our systematic literature review consisted of 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) focused on a research problem outside 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 present), resulting in 205 papers. Then, in this round we applied the inclusion and exclusion criteria by reading the full papers, resulting in a remaining 35 papers. These remaining papers were carefully reviewed, to verify the extent to which the findings from our study were supported by the literature or were complementary.

4.2 Results

Table 4 enumerates the vulnerability types, their mitigations and contexts that were (or were not) discussed in the literature. The columns with a star (★) symbol indicate a concept that has been discussed in the literature but we have not observed from the data we collected in our empirical study. The bullet symbol (●) indicates a finding that has not been previously explored by the literature.

As shown in Table 4, previous works support our empirical findings with regards to the *context* aspects of common vulnerabilities, except in case of issues that occurred during API request handling. We discovered **4** vulnerability types not covered in the literature, which were: *lack of plug-in's configuration file sanitization*, *spoofing the origin of a plug-in install request*, *improperly checking the origin of an install request*, and *plug-ins requests are handled without authorizing plug-ins that initiate the request*. Our taxonomy also covers **11** mitigation techniques that we learned from our data. The literature had covered **5** types of mitigation techniques that we have not observed in our empirical study.

5 PRACTITIONERS PERSPECTIVE ON FINDINGS

We interviewed six practitioners with work experience in design, development or penetration testing of plug-and-play systems (Table 5).

Table 4: Comparison with the literature

Context	Problem	Solution
Plug-ins installation ([39], [40], [10])	Incorrect user notification of plug-in permissions ([56], [2], [43])	Central point for installation logic ●
	Bypassing user notification or restriction checking process for plug-in installation ([17])	Anomaly-based detection method ([17]★)
		Emulation-based mitigation technique ([17]★)
		Central point for installation logic ●
	Lack of plug-in's configuration file sanitization ●	Configuration validator ●
	Extraction/storage of plug-in with world-readable/writable permissions or in unsafe directories ([10], [25])	Dedicated secure storage ([10])
	Spoofing the origin of a plug-in install request ●	Secure communication of plug-in resources ([20])
	Improperly checking the origin of an install request ●	Whitelist of install origins ●
Plug-ins update ([10])	Elevation of privilege through a plug-in update ([45])	Lifetime enforcement of plug-in permissions ●
Plug-and-Play execution environment ([42], [30], [32], [38], [28], [51], [6], [63], [35])	Lack of compartmentalization of plug-ins ([25], [15], [35], [54], [18])	Isolated object domains ([30], [40], [39], [45], [25])
		Compartmentalization of plug-ins ([42], [28], [46])
	Lack of fine-grained and modular permission setting ([28], [45], [6], [18])	Fine-grained, modular, permission assignment ([7])
		Declarative-based request for accessing data/functionality ([30], [51])
	Allowing a plug-in to elevate its permission by manipulating (or delegating a task to) a process in the PnP environment that has higher privileges ([28], [30])	Limit plug-ins exposure to High privilege PnP APIs ●
		Runtime monitoring of plug-in's behavior ([42], [25]★)
		Limit plug-ins exposure to OS processes ([42], [28])
	Improper object access control in compartmentalized PnP environment ([45], [30], [29], [42])	Name-based access control ([30]★)
		Security Policy Enforcement through Object Wrappers ([42], [45])
	Unsanitized plug-in data ([32], [38], [55], [57], [12], [15], [50], [34], [37], [14], [41])	Client-side defense using XSS filters ([34], [37]★)
		Input validation of incoming plug-in data ●
	Improper origin check of requests by plug-ins ([40], [51], [25])	Origin check ●
	Improper isolation of objects used by plug-ins in PnP environment ([28], [46], [45], [25], [40])	Isolated object domains ([30], [40], [39], [45], [25])
Plug-ins Request Handling	Plug-ins requests are handled without authorizing plug-ins that initiate the request ●	Authorize the source of events ●
		Decomposition of events ●
		Hide PnP internal events ●
	Reentrant event callbacks ([19])	Atomic event dispatcher ●

The interview format was semi-structured initiated with three questions and other questions were asked as the discussion evolved. Furthermore, throughout the interview, each vulnerability type and its mitigation technique were discussed. The initiating questions were:

Table 5: Interviewed practitioners

Practitioner	Medium	Background
P1	In-person	Senior developer. Had work experience with plugin-based systems
P2	In-person	Penetration tester for "*IoTcompany*" (anonymized). Has over two years on penetration testing of plug-and-play systems in the Internet of Things (IoT) domain.
P3	Conference call	Principal architect at *anonymized*. Over 15 years of experience and contributor of the OpenMRS medical record system that uses plug-and-play architecture
P4	E-mail	Apache OFBiz long-term developer.
Marc Alff (**P5**)	E-mail	Software Architect (MySQL). Over 20 years of experience in architecture, design and integration of complex, enterprise-scale systems.
P6	E-mail	Architect of a plug-in based enterprise system, with 13 years of professional experience

— **IQ1:** *Would this taxonomy have helped you to discover and/or prevent vulnerabilities in your system associated with the use of plug-ins? Please explain.*
— **IQ2:** *Based on your experience, do you consider that the taxonomy covers all the types of vulnerabilities you have seen? Do you remember of any plug-in-related vulnerability that did not fit the framework?*

— **IQ3:** *Did you consider the mitigations to be complete? Based on your experience, do you consider there were better ways to fix such problems?*

All practitioners were provided ahead of time with a copy of the taxonomy, a detailed explanation of the vulnerability types covered in the taxonomy, consequences, mitigation techniques, and the initiating questions. After interviewing all the practitioners, we performed a qualitative analysis of the interviews. In this process, we performed a line-by-line coding of the transcriptions and iteratively identified common themes across the answers as well as positive and negative feedbacks to our taxonomy.

5.1 Results

All practitioners provided a positive answer to *IQ1*, indicating that this taxonomy would be useful for them to discover and/or prevent vulnerabilities. *P4* asked our permission for the catalog to be distributed among all the members of their security team such that they could implement some of the mitigations. Similarly, *P6* also indicated that this taxonomy gave him/her ideas to be applied in his/her system. Marc Alff (*P5*) provided the following answer:

> *"Some of the security fixes implemented in the past in MySQL are direct implementations of the mitigations listed. Having access to your theory before releasing the code in the first place would have avoided a couple of these vulnerabilities."*

Concerning *IQ2*, all practitioners agreed that, to their knowledge, the taxonomy covered in details the vulnerability types they have seen. Furthermore, they confirmed the correctness of mitigation techniques. *P6*, an architect of a plug-and-play system, offered the following in response to IQ1 and IQ2:

> *"I find them thorough, but they would require a large amount of work and rewriting some parts of the plug-in system in our project to put all the checks and balances in place. This is actually giving me lots of ideas that I was unaware of."*

P2 provided the following answer for *IQ1*:

> *"Some of these are well thought of. A lot of these are good as a checklist for penetration testing. I have seen at least half of these vulnerability types in industrial control systems (ICS) that work with a plug-and-play design model."*

P3 provided the following response to *IQ2*:

> *"The taxonomy looks pretty comprehensive in terms of covering a wide range of issues. This is well beyond the security that we had resources to deal with. We learned that it becomes exceedingly expensive to really limit what is happening within a JVM-based plug-and-play system. We made a decision among the tech-leads of OpenMRS, to accept that every plug-in that someone chooses to install can do anything. Our plug-ins (intentionally) share the same object that core uses and the plug-ins intentionally have a lot of privileges. We tried to offload the responsibility to the system administrator to do their research for*

> *installing the plug-ins. These are the decisions we made for the convenience because we didn't have the resources to implement these mitigations."*

Regarding our **IQ3**, all the practitioners indicated that the mitigations are appropriate. One practitioner (*P5*) explained that:

> *"MySQL implements this mitigation* [Dedicated secure storage]*, with PLUGIN_DIR."*

5.1.1 Follow-up Questions. An emerging question after **IQ1** was concerning the presentation of our findings. We, initially, provided to the individuals our taxonomy as a table (with a summary of the vulnerability types, consequences and mitigations) along with a PDF with a in-depth descriptions of these. When answering IQ1 three of them (*P1*, *P3* and *P2*) hint that our findings could be used as a checklist. We then followed up with this question:

— **FQ1:** *Would you suggest another presentation structure for our findings?*

All of the practitioners suggested representing the findings as a tree-like diagram, but there were discrepancies in terms of what elements should be at the top, and which ones should be leaves. These discrepancies are mostly due to the nature of their job (architect, developer, or pen-tester), as we can observe from the transcripts below. *P3*, a software architect, provided the following answer:

> *"This is useful and good for communicating the problems; I'd like to have a taxonomy like this organized based on architectural choices and the vulnerabilities mitigated with them. Then I can pick up an architectural element to bring into our system and know the vulnerabilities that are covered."*

Similarly, *P1* (a software developer) suggested :

> *"It probably could be presented like a decision tree. When you go down the tree, for example, the context is plug-in installation, then we have the type of vulnerability."*

On the other hand, *P2* (a pen tester) answered that:

> *"The structure of taxonomy starting from context is good. It tells the penetration testers where to starts. The first two columns* [contexts and vulnerability types] *are useful, the consequences for us is not useful as we already know these. Mitigations are up to the company's discretion (these are recommendations that we can make for them)."*

As a result of this follow up question *FQ1*, we revised our taxonomy representation in a tree like diagram (as already shown in Figures 4-7).

Since our taxonomy was derived from Internet-based applications (Web browser, mail client and messaging app), we asked *P2* (a pen-tester of IoT apps) the follow-up question:

— **FQ2:** Have you observed any of these vulnerability types in the IoT apps that you have been testing?

To which case *P2* answered:

> *"I've seen, at least, half of these in devices."*

Such an answer indicates that even though our case studies were in the Internet-based applications, our findings are still applicable to different domains.

5.1.2 Common Themes on the Interviews. We observed two orthogonal ways for incorporating plug-ins into the system at runtime: a "grant all" and a "denial all" approach. In a **"Deny All" integration of plug-ins** approach (observed in Thunderbird, Chrome, and Firefox), plug-ins have limited access to the application's functionalities and data by default. Plug-ins are compartmentalized in different processes, and the host application uses declarative permission in configuration files to restrict plug-ins access. In a **"Grant All" integration of plug-ins**, plug-ins have **unrestricted** access to their internal APIs, and thereby have unrestricted access to functions and data. This approach was implemented in WordPress, Pidgin, OpenMRS, and OfBiz. All of them lack built-in security mechanisms for protecting the application core from plug-ins abuse/misuse (such as compartmentalization and access control). The underlying assumptions of these projects were that users will be careful enough to not install malicious plug-ins as well as that a plug-in developer would follow secure coding guidelines and release bug-free plug-ins [61]. It is important to highlight that we also observed this integration approach in MySQL, a non-studied project, as pointed out by Marc Alff (*P5*): "*There is no "second class citizen" execution environment with constraints in place which would somehow restrict what a plug-in has access to and what it can do.*". The rationale provided was that "*for performance reason, code from a plug-in should be as efficient as code from the server (which excludes constrained environments).*" which is the same mentioned by *P3*'s answer. Besides the performance reason underlying the decision of adopting the "Grant All" approach, another compounding factor is the implementation costs and efforts. Our multi-step study indicates that:

> Addressing security in Plug-and-Play architecture is difficult and expensive; many applications have either neglected security mechanisms or have missed detailed design decisions to address security goals. This empirical study has identified a number of common vulnerabilities prevalence to Plug-and-Play architectures. While these findings may not represent all the possible ways that plug-and-play architectures can be vulnerable, they have been considered important and useful by the practitioners. The results can be generalized to other types of systems. As argued by Wieringa et. al. [60], describing the context of studied cases as we did allows us to "generalize" the findings by analogy (i.e., findings may apply to systems that have used architectural solutions that are partially similar to the cases of this study.) For instance, the findings are generalizable to the automobile industry that adopts PnP design solutions.

6 THREATS TO VALIDITY

In this section we discuss *construct*, *internal* and *external* threats to the validity of this work [44], specifically the research design, 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. While performing our grounded theory-based approach, we rigorously followed the key principles of the classical 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 analysts' hypotheses.

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 analysis 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, our constant comparative analysis and memos helped us to elaborate on the reasoning behind our codes. Moreover, our analysis process included five individuals with security backgrounds.

External validity refers to the extent to which our results are generalizable and applicable to other extensible software. One threat is related to the limited number of applications used in our study. We have not covered applications from energy, medical or automotive domains. However, the results of SLR and interviews indicates that our findings are supported by existing ad-hoc studies and can be expanded to those domains.

7 CONCLUSION

We followed a grounded theory-based approach to study vulnerabilities in plug-and-play software systems. We contributed to an in-depth empirical study of types of vulnerabilities found in plug-and-play architectures based on data from several widely used projects. In our study, we uncovered 16 vulnerability types and 19 mitigation procedures that are presented as a taxonomy. A systematic literature review was conducted to verify to what extent previous studies corroborated with our findings as well as which aspects of our findings complement the state-of-the-art in plug-and-play software systems. Lastly, we interviewed six practitioners to obtain their feedback regarding the value of our findings to the body of knowledge in software design and development.

REFERENCES

[1] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *Softw. Syst. Model.*, 13(4):1367–1394, Oct. 2014.

[2] M. Alam, X. Zhang, M. Nauman, S. Khan, and Q. Alam. Mauth: A fine-grained and user-centric permission delegation framework for multi-mashup web services. In *2010 6th World Congress on Services (SERVICES-1)*, pages 56–63. IEEE, 2010.

[3] D. Arney, S. Fischmeister, J. M. Goldman, I. Lee, and R. Trausmuth. Plug-and-play for medical devices: Experiences from a case study. *Biomedical Instrumentation & Technology*, 43(4):313–317, 2009.

[4] D. Arney, J. Plourde, R. Schrenker, P. Mattegunta, S. F. Whitehead, and J. M. Goldman. Design pillars for medical cyber-physical system middleware. In *OASICs-OpenAccess Series in Informatics*, volume 36. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[5] G. Baldoni, M. Melita, S. Micalizzi, C. Rametta, G. Schembra, and A. Vassallo. A dynamic, plug-and-play and efficient video surveillance platform for smart cities. In *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 611–612, Jan 2017.

[6] S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 54(9):91–99, Sept. 2011.

[7] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[9] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 81–90. ACM, 2009.

[10] D. Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, 2005.

[11] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[12] J. C. C. M. da Fonseca and M. P. A. Vieira. A practical experience on the impact of plugins in web security. In *IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 21–30. IEEE, 2014.

[13] M. M. Danielle Gonzalez, Fawaz Alhenaki. Architectural security weaknesses in industrial control systems (ics): An empirical study based on disclosed software vulnerabilities. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.

[14] S. Das and M. Zulkernine. Cloubex: A cloud-based security analysis framework for browser extensions. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 268–275. IEEE, 2016.

[15] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *2009 Annual Computer Security Applications Conference*, pages 382–391, Dec 2009.

[16] J. Dietrich, J. Hosking, and J. Giles. A formal contract language for plugin-based software engineering. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 175–184, July 2007.

[17] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In J. Camenisch and D. Kesdogan, editors, *iNetSec 2009 – Open Research Problems in Network Security*, pages 52–62, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[18] I. Foster, A. Prudhomme, K. Koscher, and S. Savage. Fast and vulnerable: A story of telematic failures. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, WOOT'15, pages 15–15, Berkeley, CA, USA, 2015. USENIX Association.

[19] J. Frtunikj, V. Rupanov, A. Camek, C. Buckl, and A. Knoll. A safety aware run-time environment for adaptive automotive control systems. In *Embedded real-time software and systems (ERTS2)*, 2014.

[20] D. Gangadharan, J. H. Kim, O. Sokolsky, B. Kim, C.-W. Lin, S. Shiraishi, and I. Lee. Platform-based plug and play of automotive safety features: Challenges and directions. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 76–84. IEEE, 2016.

[21] B. G. Glaser. *Theoretical sensitivity: Advances in the methodology of grounded theory*. Sociology Pr, 1978.

[22] B. G. Glaser. *Basics of grounded theory analysis: Emergence vs forcing*. Sociology press, 1992.

[23] B. G. Glaser and A. L. Strauss. The discovery of grounded theory: Strategies for qualitative research. *New York: Aldlne*, 1967.

[24] M. Greiler, A. v. Deursen, and M.-A. Storey. Test confessions: a study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*, pages 244–254. IEEE Press, 2012.

[25] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. In *SP 2008. IEEE Symposium on Security and Privacy*, pages 402–416. IEEE, 2008.

[26] J. Himmelspach and A. M. Uhrmacher. Plug'n simulate. In *Simulation Symposium, 2007. ANSS '07. 40th Annual*, pages 137–143, March 2007.

[27] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX Security Symposium*, pages 579–593, 2015.

[28] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan. An analysis of the mozilla jetpack extension framework. In *European Conference on Object-Oriented Programming*, pages 333–355. Springer, 2012.

[29] T. Kwon and Z. Su. Detecting and analyzing insecure component usage. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 5:1–5:11, New York, NY, USA, 2012. ACM.

[30] P. W. l. Fong and S. A. Orr. A module system for isolating untrusted software extensions. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 203–212, Dec 2006.

[31] G. McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[32] I. Medeiros, N. Neves, and M. Correia. Equipping wap with weapons to detect vulnerabilities: Practical experience report. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 630–637. IEEE, 2016.

[33] O. Mesa, R. Vieira, M. Viana, V. H. Durelli, E. Cirilo, M. Kalinowski, and C. Lucena. Understanding vulnerabilities in plugin-based web systems: an exploratory study of wordpress. In *Proceedings of the 22nd International Conference on Systems and Software Product Line*, pages 149–159. ACM, 2018.

[34] B. Mewara, S. Bairwa, and J. Gajrani. Browser's defenses against reflected cross-site scripting attacks. In *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT)*, pages 662–667. IEEE, 2014.

[35] J. W. Min, S. M. Jung, and T. M. Chung. Filtering malicious routines in web browsers using dynamic binary instrumentation. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 554–557. IEEE, 2012.

[36] National Vulnerability Database. NVD Data feeds. https://nvd.nist.gov/vuln/data-feeds, 2017. (Accessed on 04/31/2016).

[37] F. B. M. Nor, K. A. Jalil, and J. l. Ab Manan. An enhanced remote authentication scheme to mitigate man-in-the-browser attacks. In *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, pages 271–276, June 2012.

[38] P. J. C. Nunes, J. Fonseca, and M. Vieira. phpsafe: A security analysis tool for oop web application plugins. In *2015 45th Annual IEEE/IFIP International Conference onDependable Systems and Networks (DSN)*, pages 299–306. IEEE, 2015.

[39] D. Oliveira, J. Navarro, N. Wetzel, and M. Bucci. Ianus: Secure and holistic coexistence with kernel extensions - a immune system-inspired approach. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1672–1679, New York, NY, USA, 2014. ACM.

[40] D. Oliveira, N. Wetzel, M. Bucci, J. Navarro, D. Sullivan, and Y. Jin. Hardware-software collaboration for secure coexistence with kernel extensions. *SIGAPP Appl. Comput. Rev.*, 14(3):22–35, Sept. 2014.

[41] J. Pan and X. Mao. Detecting dom-sourced cross-site scripting in browser extensions. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 24–34. IEEE, 2017.

[42] R. Rajkumar, A. Wang, J. D. Hiser, A. Nguyen-Tuong, J. W. Davidson, and J. C. Knight. Component-oriented monitoring of binaries for security. In *2011 44th Hawaii International Conference onSystem Sciences (HICSS)*, pages 1–10. IEEE, 2011.

[43] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.

[44] P. Runeson and M. Hoest. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, 2009.

[45] A. Saini, M. S. Gaur, and V. Laxmi. The darker side of firefox extension. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 316–320. ACM, 2013.

[46] A. Saini, M. S. Gaur, V. Laxmi, and P. Nanda. sandfox: Secure sandboxed and isolated environment for firefox browser. In *Proceedings of the 8th International Conference on Security of Information and Networks*, SIN '15, pages 20–27, New York, NY, USA, 2015. ACM.

[47] J. C. Santos, A. Peruma, M. Mirakhorli, M. Galster, J. V. Vidal, and A. Sejfia. Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 69–78. IEEE, 2017.

[48] J. C. Santos, K. Tarrit, A. Sejfia, M. Mirakhorli, and M. Galster. An empirical study of tactical vulnerabilities. *Journal of Systems and Software*, 149:263 – 284, 2019.

[49] J. C. S. Santos, K. Tarrit, and M. Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223, April 2017.

[50] H. Shahriar, K. Weldemariam, T. Lutellier, and M. Zulkernine. A model-based detection of vulnerable and malicious browser extensions. In *2013 IEEE 7th International Conference on Software Security and Reliability*, pages 198–207, June 2013.

[51] B. Shand and J. Rashbass. Security for middleware extensions: Event meta-data for enforcing security policy. In *Proceedings of the 2008 Workshop on Middleware Security*, MidSec '08, pages 31–33, New York, NY, USA, 2008. ACM.

[52] L. Sousa, A. Oliveira, W. Oizumi, S. Barbosa, A. Garcia, J. Lee, M. Kalinowski, R. de Mello, B. Fonseca, R. Oliveira, et al. Identifying design problems in the source code: a grounded theory. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 921–931. IEEE, 2018.

[53] K. J. Stol, P. Ralph, and B. Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 120–131, May 2016.

[54] M. Sun, D. Gu, J. Li, and B. Li. Pyxhon: Dynamic detection of security vulnerabilities in python extensions. In *2012 International Conference on Information Science and Technology (ICIST)*, pages 461–466. IEEE, 2012.

[55] H. Trunde and E. Weippl. Wordpress security: an analysis based on publicly available exploits. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, page 81. ACM, 2015.

[56] G. Varshney, M. Misra, and P. Atrey. Browshing a new way of phishing using a malicious browser extension. In *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, pages 1–5, April 2017.

[57] J. Walden, M. Doyle, R. Lenhof, J. Murray, and A. Plunkett. Impact of plugins on the security of web applications. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, page 1. ACM, 2010.

[58] S. Wang, G. S. Avrunin, and L. A. Clarke. Architectural building blocks for plug-and-play system design. In I. Gorton, G. T. Heineman, I. Crnković, H. W. Schmidt, J. A. Stafford, C. Szyperski, and K. Wallnau, editors, *Component-Based Software Engineering*, pages 98–113, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[59] S. Wang, G. S. Avrunin, and L. A. Clarke. *Plug-and-Play Architectural Design and Verification*, pages 273–297. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[60] R. J. Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[61] WordPress.org. Hardening wordpress. https://codex.wordpress.org/Hardening_WordPress#Themes_.2F_Plugins, 2018. (Accessed on 03/01/2018).

[62] H. Zhang, M. A. Babar, and P. Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, 2011.

[63] R. Zhao, C. Yue, and Q. Yi. Automatic detection of information leakage vulnerabilities in browser extensions. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1384–1394. International World Wide Web Conferences Steering Committee, 2015.

gangadharan2016platform [20]
rajkumar2011component [42]
alam2010mauth [2]
fong2006module [30]
medeiros2016equipping [32]
nunes2015phpsafe [38]
rieck2010cujo [43]
oliveira2014ianus [39]
karim2012analysis [28]
oliveira2014hardware [40]
saini2015sandfox [46]
saini2013darker [45]
shand2008security [51]
bandhakavi2011vetting [6]
zhao2015automatic [63]
bezemer2009automated [9]
trunde2015wordpress [55]
walden2010impact [57]

grier2008secure [25]
da2014practical [12]
dhawan2009analyzing [15]
min2012filtering [35]
shahriar2013model [50]
mewara2014browser [34]
nor2012enhanced [37]
das2016cloubex [14]
varshney2017browshing [56]
pan2017detecting [41]
sun2012pyxhon [54]
manuel2009mitigating [17]
birsan2005plug [10]
foster2015hackingCars [18]
frtunikj2014safety [19]
kwon2012detecting [29]
barth2010protecting [7]