Understanding Software Vulnerabilities Related to Architectural Security Tactics

An Empirical Investigation of Chromium, PHP and Thunderbird

Joanna C. S. Santos*, Anthony Peruma*, Mehdi Mirakhorli*, Matthias Galster†, Jairo Veloz Vidal*, Adriana Sejfia*
*Rochester Institute of Technology, USA
†University of Canterbury, New Zealand
jds5109@rit.edu, axp6201@rit.edu, mxmvse@rit.edu, mgalster@ieee.org, jpv8322@rit.edu, axs1461@rit.edu

***Abstract*—To satisfy security requirements, software architects often adopt security tactics. These architectural tactics provide mechanisms for resisting, detecting, reacting to and recovering from attacks. Consequently, flaws in the implementation of security tactics or their deterioration during software evolution and maintenance can introduce severe vulnerabilities that could be exploited by attackers. However, we currently lack an in-depth understanding of the types and impact of vulnerabilities related to security tactics. Therefore, in this paper, we conduct a first-of-its-kind in-depth case study involving three large-scale open-source systems. We investigate the most common types of vulnerabilities associated with security tactics, how frequently they may occur over time, and how fixing them differs from fixing vulnerabilities that are not related to security tactics. Key findings are (i) most tactic-related vulnerabilities were related to the tactics "Validate Inputs" and "Authorize Actors", (ii) vulnerabilities related to tactics have a similar distribution over time and software releases as vulnerabilities that are not related to tactics, (iii) fixing tactic-related vulnerabilities is not necessarily more complex than fixing vulnerabilities that are not related to security tactics. This study highlights the importance of ensuring an appropriate implementation of security-related design decisions in code to avoid vulnerabilities rooted in the architecture.**

I. Introduction

Software engineers face a constantly growing pressure to build secure software by design [9], where systems have to be designed from the ground up to be secure and resistant to attacks. To achieve this goal, software architects work with various stakeholders to identify security requirements and adopt appropriate architectural solutions to address these requirements. These architectural solutions are often based on security tactics [10]. Bass et al. [5] provide a comprehensive list of such tactics and define security tactics as reusable solutions to satisfy security quality attributes regarding resisting attacks (e.g., tactic "Authenticate Actors"), detecting attacks (e.g., tactic "Detect Intrusion"), reacting to attacks (e.g., tactic "Revoke Access"), and recovering from attacks (e.g., tactic "Audit"). As argued by Cervantes et al. [8], strategic, system-wide architectural approaches result in the highest security and lowest maintenance costs.

The importance of implementing architectural tactics correctly in the source code was highlighted in our earlier work [17]. We found that even when suitable architectural tactics are chosen upfront, developers—especially less experienced ones—often struggle when implementing these tactics in the code and implement tactics incorrectly, causing software bugs. In particular, incorrect implementation of security tactics or the deterioration of security tactics during coding and maintenance activities [14], [17] can result in vulnerabilities in the security architecture of the system, thus compromising key security requirements. We refer to these vulnerabilities as *tactical vulnerabilities*.

This means, despite significant efforts that go into designing secure systems, security can slowly erode because of ongoing maintenance activities. Even seemingly innocuous changes made to the design can lead to degradation of security [23].

The code snippet in Listing 1 from a J2EE web application shows an incorrect implementation of the "Manage User Sessions" tactic [15]. The correct implementation of this tactic in a web application would allow the system to keep track of users that are currently authenticated (including permissions they hold). However, in the given code snippet, the application authenticates users with `LoginContext.login()` without first calling `HttpSession.invalidate()` to invalidate any existing session. This enables attackers to fixate (i.e., find or set) another user's session identifier (e.g., by inducing a user to initiate a session using the session identifier provided by the attacker). Once the user authenticates him/herself with this forged session identifier, the attacker would be able to hijack or steal his/her authenticated session. Although architects have used the "Manage User Sessions" tactic in the architecture design of the web application, the developers have failed to implement it correctly, resulting in a *tactical vulnerability* that can be exploited for session fixation attacks [25].

Recent empirical studies of security vulnerabilities have neglected the architectural context, including design decisions such as tactics and patterns [7], [11], [16]. They mostly focus on studying and understanding coding issues related to the management of data structures and variables (e.g., buffer overflow/over-read).

Therefore, this paper reports an in-depth case study of software vulnerabilities associated with architectural security tactics across three large-scale open-source systems (Chromium, PHP and Thunderbird). These three systems are among the

Listing 1 Incorrect implementation of the tactic "Manage User Sessions" in a J2EE web application resulting in session fixation vulnerability

```
1  private void auth(LoginContext lc, HttpSession
       session, final PrivilegedAction<T> action)
       throws LoginException {
2  ...
3  lc.login();
4  Subject subject = lc.getSubject();
5  session.setAttribute("Privileged User", subject)
       ; //Store the Subject in HttpSession
6  Subject.doAs(subject,action); //Run privileged
       action after authenticated
7  ...
8  }
```

top 50 vulnerable software products [1]. The study took over 18 months, in which we collected and analyzed, for each system, their source code, version control data, and bug data. We also identified the security tactics used in these systems and traced them to the source code. Furthermore, we extracted and reviewed the complete list of vulnerabilities reported for these three systems from the National Vulnerability Database (NVD)[1] and mapped these vulnerabilities to security tactics to identify "tactical" and "non-tactical" vulnerabilities. A rigorous process was used to collect and analyze the data and to mitigate potential threats to the validity of our work, and several vetting and peer-review mechanisms were implemented. Using this data, we investigated the following research questions towards understanding the relationship between security tactics and vulnerabilities in the source code:

- **RQ1:** *What are the most common tactical vulnerabilities in real software systems?* We found that "Improper Input Validation" is by far the most common vulnerability type (or *root cause* of vulnerability) across Chromium, PHP and Thunderbird. Moreover, security tactics of "Validate Inputs" and "Authorize Actors" were the most affected tactics with vulnerabilities.
- **RQ2:** *How does the number of security vulnerabilities related to architectural tactics evolve over time?* We found that even though the absolute number of tactical and non-tactical vulnerabilities differs, their numbers evolve similarly over years and releases.
- **RQ3:** *Are vulnerabilities related to tactics more complex to fix compared to non-tactical vulnerabilities?* We estimated "complexity to fix" as the code churn and the total number of affected files when fixing a vulnerability. We found that, with the exception of Thunderbird, code churn and number of affected files is not higher nor lower for tactical vulnerabilities. For tactical vulnerabilities in Thunderbird, code churn is slightly higher while the number of affected files is slightly lower.

The **contribution** of our work is two-fold:

- An empirically grounded mapping of known vulnerabilities to architectural tactics. This mapping allows us to differentiate vulnerabilities based on their root cause, e.g., vulnerabilities because of manipulation of data in the code (e.g., buffer overflow/overread), or due to an incorrect implementation of an architectural design decision in form of a security tactic. This paper presents the *first* approach to identify vulnerabilities associated with architectural tactics and to assess the existence of such vulnerabilities in three large systems.
- An investigation on how tactical vulnerabilities differ from other types of vulnerabilities (non-tactical), in terms of root causes, complexity to fix and how frequently they occur over the time.

The remainder of this paper is organized as follows: In Section II we briefly introduce various concepts and terms related to vulnerabilities to ensure that the essence of the paper can be understood by a broader audience. Section III discusses the methodology followed to conduct this study. Section IV presents the results achieved. Section V discusses the results observed. Section VI elaborates on threats to the validity of this work. Section VIII concludes this paper.

II. Software Vulnerabilities

Vulnerabilities are security-related software defects that violate the system's security requirements. Software can become vulnerable as a result of a defect that leads to a variety of consequences, such as leakage of data and the modification of data by unauthorized users. Software vulnerabilities are often publicly disclosed and tracked in databases. The **National Vulnerability Database (NVD)**[1] is an example of such databases. It currently contains more than 80,000 vulnerabilities. Vulnerabilities recorded in the NVD are assigned a **Common Vulnerabilities and Exposures (CVE)** identifier. Each CVE instance provides details about the vulnerability. The following excerpt retrieved from NVD shows a vulnerability in the PHP project.

> **CVE ID**: CVE-2011-3189
> **Overview**: The crypt function in PHP 5.3.7, when the MD5 hash type is used, returns the value of the salt argument instead of the hashed string, which might allow remote attackers to bypass authentication via an arbitrary password, a different vulnerability than CVE-2011-2483.
> **References**: https://bugs.php.net/bug.php?id=55439, **[...]**
> **Affected Versions**: PHP 5.3.7
> **Vulnerability Type** Cryptographic Issues (CWE-310)
> **[...]**

The **vulnerability type** (or **root cause**) indicates the cause of security issues. It refers to an entry from a list of known common software issues documented in the **Common Weakness Enumeration (CWE)** catalog [2]. This catalog is supported by the U.S. Department of Homeland Security (DHS) and maintained by the MITRE corporation. The CWE catalog is a community-developed catalog of common types of weaknesses in software that could introduce a breach that can be exploited by attackers. These weaknesses can be introduced during design, implementation, configuration, or other phases of the software development cycle. Many of these weaknesses, such

[1]https://web.nvd.nist.gov/

[2]https://cwe.mitre.org

as *buffer overflow* or *buffer overread*, are mainly due to mistakes in manipulating data structures. However, there are many weaknesses in this catalog which are associated with security tactics. These weaknesses are tactical vulnerabilities, as they are either due to an incorrect adoption of security tactics or violation of the tactic's key principle in the source code. An example of a tactical vulnerability was given in Listing 1. Another example from the CWE catalog is "Exposure of Data Element to Wrong Session" [3] which occurs when a product does not sufficiently enforce boundaries between the states of different sessions, causing data to be provided to or used by the wrong session.

Using the information from the NVD, the CWE catalog, software repositories, and the link between these sources of information, it is possible to study the root causes of software vulnerabilities in software systems. Furthermore, it is possible to distinguish tactical vulnerabilities from purely coding bugs (i.e., non-tactical vulnerabilities).

III. Case Study Setup

To answer our research questions from Section I, we conducted an in-depth case study with three cases [20] based on guidelines for industrially-based multiple-case studies [24]. The unit of analysis in our study was a software project.

A. Case Selection

We selected the following three open-source projects:

- **Chromium** [4] is the project of Google Chrome (web browser). It is built on top of existing projects, e.g. WebKit [5]. It is ranked 4th among software products with the most vulnerability disclosures [1].
- **PHP** [6] is the interpreter of the PHP programming language. It is ranked 23rd among software products with the most vulnerability disclosures [1].
- **Thunderbird** [7] is a cross-platform email, news, calendar, and chat client. It is actively maintained by the Mozilla foundation and ranked 15th among software products with the most vulnerability disclosures [1].

These projects were selected because (i) they are widely adopted by a large number of users, (ii) they are among the top 50 software projects with the highest number of vulnerabilities [1], (iii) they have adopted a wide range of architectural tactics, and (iv) their development team uses an issue tracking system for managing and fixing the defects. Therefore, they provide a rich set of artifacts regarding the software development activities conducted, security tactics used, vulnerabilities discovered, and fixes to vulnerabilities. Table I provides additional details about each system. The three systems are rather diverse in size, age, and their application domain. They are similar in that they are open-source software systems and mostly implemented in C/C++.

[3]https://cwe.mitre.org/data/definitions/488.html
[4]http://www.chromium.org/
[5]http://webkit.org/
[6]http://php.net/
[7]http://mozilla.org/thunderbird/

Table I
Details about the Studied Systems

	Chromium	PHP	Thunderbird
Size (LOC)	>1,000K	>4,000K	>14,000K
Number of major releases	56	18	22
Total contributors	5,223	423	889
Core contributors	1904	114	83
Age	9 years started in 2008	22 years started in 1994	18 years started in 1998
Release cycle (major releases)	6 weeks	Yearly	6 weeks
Domain	Web browser	Script language for web apps	Email, calendar, chat client
Language(s)	Mostly C++	Mostly C	Mostly C++
Vulnerabilities	705	531	1,380
Number of users	~1 billion	~244 millions	~9 millions

B. Data Collection

Since vulnerability data changes continuously, it is important to note that our data collection process covers all the data from the studied cases until the beginning of 2016. To obtain the data required to answer our research questions, we performed the following high-level steps:

1) Identification of security tactics for each of the three projects (Section III-C)
2) Extraction of disclosed vulnerabilities for each of the three projects from the NVD (Section III-D)
3) Classification of tactical and non-tactical vulnerabilities following two complementary approaches to ensure accuracy and completeness of the classification:
   - Bottom-up (Section III-E)
   - Top-down (Section III-F)

These steps relied on a broad range of data elements and software development artifacts. Therefore, to help the reader understand how we utilized these data elements and artifacts, we present in Figure 1 an information model. This information model shows each data element used in our study and the relationships between data elements and to various software development artifact repositories.

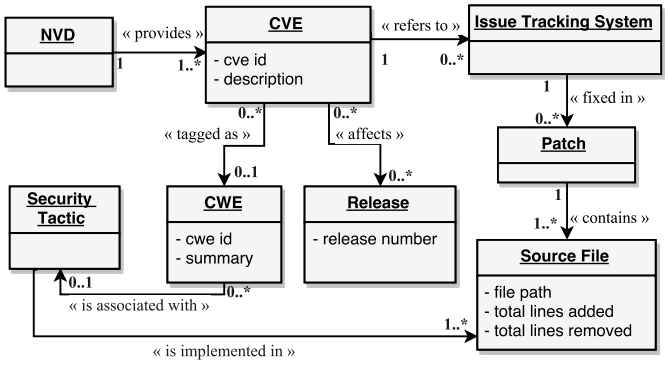


Figure 1. Data Extraction Information Model

C. Identifying Security Tactics in each Project

The first step of our study involved identifying the security tactics used in the three projects. We performed several complementary activities to ensure the accuracy of the extracted tactics:

Table II
SECURITY TACTICS IN CHROMIUM, PHP AND THUNDERBIRD

	Identify Actors	Validate Inputs	Manage User Sessions	Authenticate Actors	Authorize Actors	Limit Access	Limit Exposure	Encrypt Data	Separate Entities	Change Default Settings	Inform Actors	Detect Denial of Service Attack	Detect Intrusion	Verify Message Integrity	Audit
Chromium	*	*	*	*		*	*		*	*	*	*	*	*	*
PHP		*	*	*	*	*		*						*	*
Thunderbird	*	*	*	*	*			*			*		*	*	*

1) We reviewed the available literature and technical documentations for each project (e.g., [4] for Chromium) to look for any references to specific security tactics. We then manually checked if these tactics occurred in the code.
2) We manually browsed through the source files in each project to identify tactic-related files.
3) We used search features of IDE's to search through the code using tactic-related keywords (e.g. "authenticate").
4) We used a previously developed technique that automatically reverse-engineers architectural tactics from source code [15], [18].

The results from the above four activities were merged to document the set of tactics used in each project. We then obtained feedback from developers if they agree with the identified tactics: For Chromium we received feedback from the lead of the security team, and for PHP and Thunderbird, we obtained feedback from two developers who contributed to the implementations of the security tactics. The list of identified security tactics for each project is shown in Table II.

D. Extracting Disclosed Vulnerabilities for each Project

Next, we retrieved CVEs for the three systems from the NVD. Typically, each CVE entry records releases of the software that were impacted by the vulnerability and provides a link to the project's *Issue Tracking System*. Furthermore, as shown in the data extraction information model (Figure 1), each CVE instance has a CWE tag, which identifies the root cause of the vulnerability (see also example of a vulnerability in PHP in Section II).

Completeness check: Not every CVE entry in NVD is complete, i.e., it does not provide all the data we need for our study (e.g. the patch that was released to fix the vulnerability). Therefore, we conducted a manual analysis of the CVE entries to verify whether the corresponding entries in the issue tracking systems of the three projects were provided. In case the NVD did not provide a link to the corresponding entry in the issue tracking system, we searched over the project's bug repository using the CVE ID to verify that each CVE was indeed acknowledged by the developers, fixed, and that the fix was released. This manual analysis was conducted by three researchers over a time span of a year. As a result, we collected a total of 2,386 CVEs spanning across the lifetime of these projects. From these vulnerabilities, 1,252 were related to the Chromium project (since 2008 when the project started), 430 were associated with the PHP project published since 1997, while 704 were in the Thunderbird project reported since 2002.

Table III
INSTRUCTIONS GIVEN TO THE EXPERTS TO CLASSIFY CVES

Instructions	
Steps: (i) Read the CVE description, (ii) Check the modified code: comments, changed function/method/class, etc (iii) Read the bug tracking discussion (iv) Read the commit message.	
Examples of low level issues:	
- Solely coding mistake	
- An integer overflow / underflow	
- Use of a pointer after free	
- Incorrect calculations of buffer sizes	
Examples of tactical issues:	
- Missing critical step in the authentication tactic	
- Improper handling of insufficient permissions or privileges in authorization tactic	
- Errors in tactical code and principles of the tactic	
- CVE violates a design decision made by the developer	
- Missing encryption of sensitive data	
Answer Sheet	
Is the error very low level?	☐Yes ☐ No
Is the source code changed implementing any security mechanisms for *Resisting* , *Detecting* or *Recovering* from a potential attack?	☐Yes ☐ No
Is CVE in a tactical file? (Yes: Investigate)	☐Yes ☐ No
Is CVE impacting the tactic?	☐Yes ☐ No
What is the name of the impacted tactic?	
Your decision: Tactical (Yes) / Non-tactical (No)	**☐Yes ☐ No**
Describe your rationale and provide evidence:	

Eliminating invalid CVEs: During this process we discarded *invalid vulnerabilities*, i.e., vulnerabilities that NVD tags as deprecated or duplicated CVEs, or vulnerabilities external to the studied cases (e.g. vulnerabilities in applications written in PHP rather than in PHP itself). Furthermore, we discarded CVEs for which we could not identify their corresponding entry in the issue tracking system or when the issue was still marked as private,i.e., there were restrictions that prevented issues being shown to the general public.

Tracing CVEs to patches: For each CVE that we collected the corresponding defect entry in the project's *Issue Tracking System*, we obtained the *patch* that was released to fix the vulnerability as well as the *source files* that were modified to fix the vulnerability.

E. Bottom-up Approach to Identify Tactical Vulnerabilities

This approach relied on manually reviewing all CVE reports of the studied projects to classify CVE's as tactical or non-tactical. To reduce the bias of a manual classification, we performed a peer evaluation by two developers (one with eight years of experience in software architecture and security and the other with three years of experience in this field). These subject matter experts manually analyzed all the CVEs for each studied project and provided a *rationale* for their decisions. To ensure consistency, each expert was provided with instructions for classifying CVEs (see Table III).

Both subject matter experts also conducted detailed code reviews to classify the CVEs. We provided the tactical files (i.e., source files that implement tactics) in these projects and a matrix indicating the overlap of CVEs and tactical files. As described in Section III-C, we reverse-engineered security tactics in the source code.

Once each subject matter expert had finished their classification, disagreements were discussed (based on each person's rationale) and resolved. Finally, the classifications of both experts were consolidated into one dataset that contains for each vulnerability the *CVE ID*, the *Description*, the *Affected Releases*, its *Root Cause* (i.e., CWE), *associated tactic* (for tactical vulnerabilities) and the *Report Date*.

F. Top-down Approach to Identify Tactical Vulnerabilities

This approach relied on the CWE catalog (see Section II) to distinguish tactical and non-tactical vulnerabilities. This CWE catalog defines known software weaknesses that can result in software vulnerabilities and tags every type of vulnerability (CWE) with information about when the vulnerability is introduced (*requirements*, *architecture*, *implementation* etc.). This provides clues if a CWE is due to an architectural or coding issue. Also, since descriptions of CWEs are generic and not specific to a project, they typically include the name of security mechanisms (e.g. tactic/patterns) involved.

Thus, in this top-down approach we performed two steps:

- **Step 1:** We labeled CWEs as tactical or non-tactical. This resulted in a **gold standard** for tactical vulnerability types. This required the following steps:
  - First, we obtained a list of all software vulnerability types (CWE instances) from the CWE catalog.
  - Then, we searched existing security tactics [5], [15] and extracted information about the context in which they can be applied, the solution proposed by the tactic, related tactics, and the keywords that summarizes a tactic.
  - Based on this knowledge, we verified which CWEs are caused by an incorrect implementation of a security tactic or when a tactic is missing. This verification was done through searching the tactic's name or keywords related to the tactic in the CWE name and description. The result of this search is a list of *potential* connections between security tactics and generic software vulnerability types (CWEs). We then manually inspect each CWE, to confirm whether these connections indeed existed. At the end of this step we **created** our **gold standard**, a catalog of tactical and non-tactical CWEs.
- **Step 2:** We **used** the **gold standard to classify CVEs** (see Figure 1 for connection between CVEs, CWEs and tactics):
  - Since vulnerabilities (CVEs) reported by NVD typically contain a *CWE Tag* describing the type of software vulnerability, we could use these tags to automatically classify CVEs as tactical or non-tactical.
  - In some cases the CVEs did not have a CWE tag specified [8]. Thus, we have used the links between *Security Tactics*, *Source Files* and *CVEs* and reviewed the content of these artifacts to tag the CVE with the most appropriate entry in our gold standard.

[8]There were 182 CVEs in Chromium, 160 in PHP and 187 in Thunderbird without CWE tags, which corresponds to 14.5%, 37.2% and 26.6% of their CVEs, respectively.

The steps described above have been vetted in a peer-evaluation process. In this evaluation, four subject matter experts worked independently analyzing each entry in the CWE catalog to identify the ones that are tactic-related. Once these four individuals finished their work, the results were double-checked and disagreements were resolved. Following this procedure, we have identified 223 common types of tactical vulnerabilities among the 727 software weaknesses from the CWE catalog. Table IV presents how many vulnerability types (CWE instances) from the CWE catalog were mapped to each security tactic.

Finally, we compared and merged the results of the bottom-up and top-down approaches. Both approaches had a significant agreement on the results. We observed 93.3% agreement in Thunderbird, 90.2% in PHP and 88.3% in Chromium. The disagreements occurred mainly because the CWE tag provided to CVEs in the NVD does not have a consistent meaning: it may indicate the specific root cause of the vulnerability (e.g "CWE-798 Use of Hard-code Credentials") or describe the consequence of a vulnerability (e.g, "CWE-200 Information Leak / Disclosure"), or it is at a higher-level of abstraction (e.g., "CWE-17 Code" which describes vulnerabilities introduced during coding), thereby it introduces mistakes in the second step of this top-down approach. In a group review session we resolved the disagreements and decided which CVEs were tactical or non-tactical CVEs.

Table IV
TOTAL NUMBER OF CWES PER SECURITY TACTIC

Security Tactic	# CWEs
Audit	6
Authenticate Actors	29
Authorize Actors	60
Cross Cutting	9
Encrypt Data	38
Identify Actor	1
Identify Actors	11
Limit Access	7
Limit Exposure	6
Lock Computer	1
Manage User Sessions	6
Validate Inputs	39
Verify Message Integrity	10

G. Data Analysis

RQ1 (most common tactical vulnerabilities): To answer this question, we identified the most frequently occurring types of tactical CVEs in each project and underlying security tactics.

RQ2 (tactical vulnerabilities over time): This question is answered by observing the trend of these vulnerabilities over the time and over releases.

RQ3 (complexity to fix vulnerabilities): We estimate the complexity to fix a given vulnerability by analyzing the total number of files and lines of code changed (modified, added, or removed) to fix the vulnerability (code churn). This was done by using the link to the issue tracking systems in each CVE and by identifying the patches that fixed the vulnerability.

IV. Results

In this section, we first present an overview of the security vulnerabilities identified in the three cases. Then, we present the answers to our research questions.

Table V
Overview of the Vulnerability Dataset

Project	#CVEs	#Discarded	#Analyzed	#Tactical	#Non-Tactical
Chromium	1252	303	949	403	546
PHP	430	267	163	63	100
Thunderbird	704	36	668	255	413

A. Overview

Table V shows the total number of vulnerabilities collected (column *#CVEs*), the number of discarded and analyzed vulnerabilities, and the number of tactical and non-tactical vulnerabilities for each project. From 1,252 vulnerabilities in Chromium, 430 in PHP, and 704 in Thunderbird, we discarded 303, 267 and 36 vulnerabilities, respectively, as explained in Section III-E. From the vulnerabilities that we analyzed, 42.5% of vulnerabilities in Chromium were tactical (403 CVEs), making Chromium the project with the highest percentage of tactical vulnerabilities (compared to Thunderbird and PHP). The percentage of tactical vulnerabilities was 38.7% for PHP (43 CVEs) and 38.2% for Thunderbird (255 CVEs).

Key findings:

- While Chromium, PHP and Thunderbird have adopted a wide range of architectural tactics to secure the systems by design, a remarkable number of vulnerabilities discovered in these systems are due to incorrect implementations of these tactics.

B. RQ1: Common Tactical Vulnerabilities

Table VI lists the the root causes (i.e., vulnerability types) of tactical vulnerabilities in each of the three studied systems, the related architecture tactics, as well as the total number of CVEs caused by the given vulnerability type. The first result of note is that *Improper Input Validation (CWE-20)* was the most common vulnerability type in both PHP and Chromium, while *Improper Access Control (CWE-284)* was the most re-occurring vulnerability type in Thunderbird. Moreover, PHP's and Chromium's second most common root cause was the *Inclusion of Functionality from Untrusted Control Sphere (CWE-829)*, which is about reusing/importing vulnerable third-party functionality. Chromium reuses code from the WebKit project, which is a rendering web browser engine that currently has 230 vulnerabilities disclosed in the NVD. In PHP, from the 8 vulnerabilities caused by the inclusion of vulnerable libraries, 7 were due to reusing code from the FileCommand project[9] while the remaining vulnerability was due to the use code from the Libmbfl 1.1.0 project [10]. Vulnerability type CWE-829 also appears in the Thunderbird (at the 10th position).

[9]https://github.com/file/file

[10]https://github.com/moriyoshi/libmbfl

Key findings:

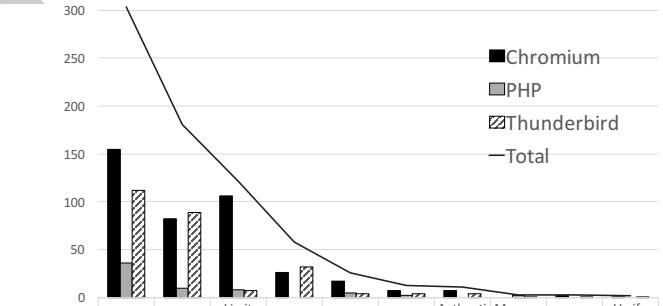
- Improper Input Validation (CWE-20) and Improper Access Control (CWE-284) are the most occurring root causes for security vulnerabilities in Chromium, PHP and Thunderbird.
- Vulnerabilities in the three studied systems are mostly related to tactics "Validate Inputs" and "Authorize Actors" for resisting attacks.

In Thunderbird these vulnerabilities occurred due to importing libraries related to 2D graphics (Skia[11]), media-handling (GStreamer[12]), font processing (Libgraphite[13]), and computer graphics (Angle[14]).

Figure 2 shows the number of CVEs per tactic. Most of the tactical issues in the studied cases are related to failing mechanism that validate inputs consistently and correctly, i.e., the tactic "Validate Inputs" (CWE-20, CWE-59, CWE-74, CWE-77, CWE-79, CWE-89, and CWE-94 in Table VI). Failing to validate user inputs can lead to a variety of consequences, such as crashes (denial of service) and leakage of sensitive information. We also observe that vulnerabilities related to the tactic "Authorize Actors" (CWE-266, CWE-269, CWE-274, CWE-284, CWE-280, CWE-426, and CWE-862 in Table VI) are common among the three systems.

Key findings:

- Security of studied projects was compromised by reusing or importing vulnerable versions of third-party libraries. In the case of Chromium such vulnerabilities occurred 106 times, while in Thunderbird and PHP, 7 and 8 times, respectively.



	Validate Inputs	Authorize Actors	Limit Exposure	Identify Actors	Limit Access	Encrypt Data	Authenticate Actors	Manage User Sessions	Audit	Verify Message Integrity
Chromium	155	82	106	26	17	7	7	0	2	1
PHP	36	10	8	0	5	2	0	2	0	0
Thunderbird	112	89	7	32	4	4	4	1	1	1
Total	303	181	121	58	26	13	11	3	3	2

Figure 2. Number of CVEs per tactic for each system

C. RQ2: Tactical Vulnerabilities over Time and Releases

This question investigates how the number of security vulnerabilities related to security tactics evolves over time,

[11]https://skia.org/

[12]https://gstreamer.freedesktop.org/

[13]http://scripts.sil.org/cms/scripts/page.php?site_id=projects&item_id=graphite_home

[14]https://github.com/google/angle

Table VI
MOST COMMON VULNERABILITY TYPES (ROOT CAUSES) IN THE STUDIED CASES

Tactic	Vulnerability Type	Chromium	PHP	Thunderbird	Total
Validate Inputs	CWE-20 Improper Input Validation	131	23	46	200
Limit Exposure	CWE-829 Inclusion of Functionality from Untrusted Control Sphere	106	8	7	121
Authorize Actors	CWE-284 Improper Access Control	35	–	51	86
Validate Inputs	CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	12	1	31	44
Identify Actors	CWE-346 Origin Validation Error	21	–	17	38
Validate Inputs	CWE-94 Improper Control of Generation of Code ('Code Injection')	5	1	30	36
Authorize Actors	CWE-274 Improper Handling of Insufficient Privileges	19	–	–	19
Identify Actors	CWE-295 Improper Certificate Validation	5	–	11	16
Authorize Actors	CWE-269 Improper Privilege Management	3	–	8	11
Authenticate Actors	CWE-287 Improper Authentication	7	–	3	10
Authorize Actors	CWE-426 Untrusted Search Path	2	–	8	10
Authorize Actors	CWE-280 Improper Handling of Insufficient Permissions or Privileges	2	6	–	8
Authorize Actors	CWE-266 Incorrect Privilege Assignment	1	–	7	8
Limit Access	CWE-73 External Control of File Name or Path	3	4	–	7
Limit Access	CWE-250 Execution with Unnecessary Privileges	4	1	–	5
Authorize Actors	CWE-862 Missing Authorization	2	2	1	5
Validate Inputs	CWE-59 Improper Link Resolution Before File Access ('Link Following')	–	2	1	3
Validate Inputs	CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')	–	2	–	2
Validate Inputs	CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	–	2	–	2
Validate Inputs	CWE-74 Improp. Neutraliz. of Spec. Elements in Output Used by a Downstream Component ('Injection')	–	1	–	1

Table VII
CORRELATION COEFFICIENTS: TACTICAL & NON-TACTICAL CVES

System	Year	Release
Chromium	0.767	0.836
PHP	0.583	0.915
Thunderbird	0.952	0.997

i.e., if the number tactic-related vulnerabilities increases, decreases, or remains constant. Figure 3 shows the number of vulnerabilities (tactical and non-tactical) over the years they were reported and Figure 4 shows these vulnerabilities over the project's releases.

Figure 3 shows that since 2012 vulnerabilities decreased over the time for Chromium and Thunderbird. This trend is confirmed when observing how vulnerabilities spread over releases (Figure 4). The most recent releases had fewer vulnerabilities than older ones. This indicates that even though over time new features are added to both systems, more vulnerabilities are removed than introduced by changing the code/adding features. The results for PHP, however, follow an opposite pattern; we observe a slight increase over the years (Figure 3). Also, when looking at vulnerabilities across releases, there is no clear trend that the number of vulnerabilities decreases with new releases.

We calculated the correlation coefficient between tactical and non-tactical vulnerabilities for the three systems to compare the patterns of evolution (see Table VII). Since the data were not normally distributed we used Spearman's rho correlation test. We found strong correlations between the evolution of tactical and non-tactical vulnerabilities over the years and releases for each system ($p = 0$ in all cases).

Key findings:

- Tactical and non-tactical vulnerabilities have a similar distribution over time and releases, even though the absolute numbers of tactical and non-tactical vulnerabilities differ.

Table VIII
AVERAGE CODE CHURN WHEN FIXING VULNERABILITIES

Project	Tactical vulnerability	Non-Tactical vulnerability
Thunderbird	113.18 (*min* = 1; *max* = 515)	92.91 (*min* = 1; *max* = 603)
Chromium	66.48 (*min* = 0; *max* = 320)	51.83 (*min* = 0; *max* = 213)
PHP	28.22 (*min* = 1; *max* = 144)	42.36 (*min* = 2; *max* = 175)

Table IX
NUMBER OF AFFECTED FILES WHEN FIXING VULNERABILITIES

Project	Tactical vulnerability	Non-Tactical vulnerability
Thunderbird	4.04 (*min* = 1; *max* = 14)	4.23 (*min* = 1; *max* = 21)
Chromium	4.72 (*min* = 1; *max* = 17)	4.09 (*min* = 1; *max* = 12)
PHP	2.12 (*min* = 1; *max* = 6)	2.24 (*min* = 1; *max* = 4)

D. RQ3: Complexity to Fix Vulnerabilities

We estimated the complexity to fix a vulnerability in terms of code churn and the total number of files affected by the fix.

1) Code Churn: In Table VIII we show the average number of lines changed when fixing a vulnerability. For Thunderbird and Chromium, average code churn appears to be higher for tactical vulnerabilities. On the other hand, for PHP, non-tactical vulnerabilities result in a higher code churn. We used a Mann-Whitney test (as the data did not have a normal distribution) to compare the code churn caused by fixing tactical and non-tactical CVEs. In the case of Chromium and PHP there was no statistically significant differences in the code churn caused by fixing tactical and non-tactical CVEs, while for Thunderbird the results were statistically significant ($p = 0.043$).

2) Affected Files: Table IX shows the average number of affected files for each system. Thunderbird and PHP show on average fewer affected files for fixing tactical vulnerabilities, while in Chromium, fixing tactical vulnerabilities affects more files. A Mann-Whitney test to compare the number of affected files when fixing tactical and non-tactical vulnerabilities showed no statistically significant difference for any of the three systems.

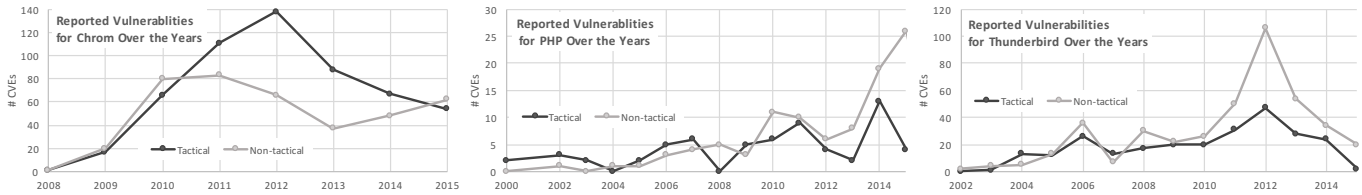


Figure 3. Occurrence of vulnerabilities (tactical and non-tactical) over the time for the case studies

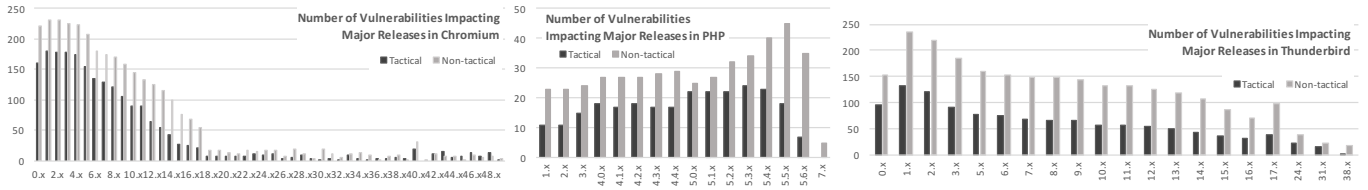


Figure 4. Vulnerabilities over the releases of the systems

Key findings:

- When fixing tactical vulnerabilities, there is no statistically higher or lower code churn compared to fixing non-tactical vulnerabilities.
- When fixing tactical vulnerabilities, the number of affected files is not statistically significantly higher or lower compared to fixing non-tactical vulnerabilities.

V. Discussion

A. Vulnerability Types and Tactics

We highlighted commonalities and differences between studied systems in Section IV. In this section, we discuss why these vulnerabilities (and related tactics) appear.

- **Some security tactics are more fragile and harder to implement:** From the results we noted that a large number of tactical vulnerabilities *in all three projects* were related to the lack of the "Validate Inputs" tactic when it would have been needed, or the incorrect implementation of this tactic. Input validation problems are more pervasive and not limited to a specific domain. Software will always be provided with inputs, which may be valid or malformed (intentionally by attackers or unconsciously by misuses from legitimate users). Vulnerabilities related to input validation issues can happen in any software system. They result in attacks related to *SQL injection*, *Command Injection*, *data manipulation*, and other similar issues such as *Cross-Site Scripting*.
- **Some tactical vulnerabilities are more prevalent in a domain:** Chromium has significantly more vulnerabilities related to the tactic "Limit Exposure" than PHP and Thunderbird. Partially this is because of the domain of Chromium as a web browser. "Limit Exposure" is about reducing the number of entry points of a system where an attacker can try to enter data to or extract data from a software process. This is particularly relevant for a web browser that has built-in processes for interacting with add-ins, add-ons (often from third parties), client-side scripts, and other web applications. Similarly, frequent vulnerabilities in Thunderbird have been impacted by its domain as a mail client.

B. Implications for Practitioners

Generalizability and applicability of findings: As argued by Wieringa, describing the context of the studied cases as in Table I allows us to "generalize" our findings by analogy (i.e., our findings may apply to projects and systems that are similar to the cases of this study) [26]. We also discuss generalizability in Section VI when we acknowledge validity threats.

Recommendations for architecture practice: Based on the findings from the case study, we follow an inductive approach [26] and infer several recommendations:

- **Track external components**: "Reuse" versus "development from scratch" is a common design decision to reduce cost and time to market. However, as shown in our findings for RQ1, security can be compromised by reusing or importing vulnerable versions of third-party libraries. Therefore, if a project reuses components (code, libraries, etc.) either entirely or partially from an external source, it is important to keep track of (i) which components are being used, (ii) which parts of the software these components interact with, (iii) if these components have a known vulnerability, and (iv) if critical data, resources, or process of the system are exposed to attackers by integration of the third-party component.
- **Trace architectural decisions to code:** As we have shown in Section IV-A, nearly half of vulnerabilities in Chromium were due to tactical issues. This means that while it is important to design and evaluate a robust security architecture, it is even more important to ensure that the security architecture is implemented correctly and there is no violation or drift from the principles of an intended architecture [14]. Tracing the tactics to source code [19], monitoring evolution of architectural decisions in the source code [15], [18] and writing test cases to ensure their correctness can help prevent tactical vulnerabilities.
- **Follow tactic-centric approach to security:** By knowing the type of tactical vulnerability that can impact the

security architecture of a software, we can plan targeted testing and assurance strategies to mitigate the security risks of the system.

- **Educate developers in architecture principles:** A recent survey of over 1,400 developers [3], indicated that 80 percent of developers missed questions about tactics and mechanisms for protecting sensitive data and 74 percent missed questions on identifying root cause of software vulnerabilities. The findings for our RQ1 supports the results of this survey that while appropriate design decisions were made by architecture (e.g. "Authorize Actors") the developers failed to correctly implement many of these design decisions, resulting severe vulnerabilities. Therefore, it is important to train the developers about the security architecture principles, tactics and vulnerabilities associated with them.

C. Implications for Software Architecture Research

Our study has several implications for software architecture research. These implications are mainly pointers towards future work. For example, while we observed a common trend of when tactical and non-tactical vulnerabilities are reported (see RQ2), our study does not offer an explanation of *why* this common trend exists. This could be due to many factors, e.g., size and age of systems, developer communities, contributors, etc. Also, given that Chromium implements the most tactics (compared to PHP and Thunderbird) but also has the highest number of tactical vulnerabilities, gives the impression that implementing tactics introduces vulnerabilities. However, our study does not explain the reasons for these vulnerabilities. Therefore, additional explanatory studies are required in the future. More specifically, future work has two major thrusts: (i) We need to conduct further empirical studies to determine the severity of tactical vulnerabilities and the cost associated to fixing them, (ii) We need to perform longitudinal empirical studies of tactical vulnerabilities to understand how such security problems develop over time, what factors contribute to tactical vulnerabilities and how we can detect/prevent them.

VI. Threats to Validity

This section discusses validity threats based on a validation scheme presented by Runeson and Hoest [20] (*construct*, *internal* and *external* validity).

Construct validity is about how accurately the applied operational measures truly represent the concepts that researchers are trying to study. In our study, these included the measures used to identify tactical and non-tactical vulnerabilities, see (Section III). To perform this identification, we leveraged on the vulnerabilities tracked by the NVD along with data from bug and issue tracking systems of Chromium, PHP, and Thunderbird. Therefore, our analysis relies on the accuracy of the data reported in these systems and we may have missed vulnerabilities that were not tracked by the NVD or we had to discard because we could not find the corresponding entry in the issue tracking system or the issue was still private by the time of our study.

Internal validity reflects the extent to which a study minimizes systematic error or bias so that a causal conclusion can be drawn. The primary threat is related to the manual analysis of CVE instances in order to observe the nature of security design issues and to identify tactical and non-tactical vulnerabilities. To mitigate this threat, we performed top-down and bottom-up classification of vulnerabilities (see Section III). Moreover, we conducted a peer review process, in which two individuals analyzed vulnerabilities and shared their rationale with each other to resolve disagreements. Parts of this peer review also included practitioners. Hence, we consider that such peer evaluation minimized the impacts of biases and mistakes by the manual inspection of CVEs.

External validity evaluates the generalizability of our findings and how they can be generalized to other systems (see Section V-B). There are two threats in this respect. Firstly, we analyzed the historical vulnerability reports from three systems (PHP, Chromium, and Thunderbird), which are mostly implemented in C/C++. We do not aim for statistical generalization, but analytical generalization: we carefully selected the three cases from different software domains and with a higher number of reported vulnerabilities. Therefore, we expect the systems to be representative of a typical software engineering environment. Also, when discussing our results, we highlighted which findings are specific to a system and which findings apply to all systems. Secondly, we identified the root causes based on a subset of types of vulnerabilities from the CWE catalog (Section III-F). Thus, we acknowledge that it may not be complete, i.e., that it does not include all possible ways that developers can implement tactics incorrectly. However, this subset comes from a community-established list of possible types of security issues that have been observed and documented in the real world.

VII. Related Work

Existing research in software architecture for security has mainly proposed techniques for facilitating the design of the security architecture [13], the analysis and evaluation of the existing security architecture [12], [21] as well as identifying potential threats/vulnerabilities from the architecture [2], [6], [27] . While these works can aid architects to identify existing threats and to appropriately adopt security patterns/tactics into a system, such activities may not be enough to avoid vulnerabilities, as the implementation of design decisions may be incorrect or erode over the time.

To help avoid deterioration of the security architecture during software maintenance, Taspolatoglu and Heinrich [23] described an approach that extended architecture description languages to formally document security requirements. While this work recognized that the implementation of security decisions may erode overtime and result in vulnerabilities, unlike our work, it did not provide evidence on how frequently such problems occur and how complex it is to fix them.

Ryoo et al. [22] evaluated to which extent security tactics are being used in open-source systems and whether there are discrepancies between the original design and the actual

implementation. Their findings suggested that developers are not strictly implementing the original design envisioned by architects and that only a subset of tactics are being implemented in systems (such as "Encrypt Data"). While in our work we also analyzed the usage of security tactics in three case studies, our main goal was to investigate how vulnerabilities are caused by incorrect adoption of these tactics in the code.

Feng et al [9] investigated the relationship between design rule violation and vulnerabilities. They observed that the files that contain a higher number of design rule violations are highly correlated to vulnerabilities as well as high levels of code churn when fixing such security bugs. However, unlike our work, they investigated the files that contain modularity violation against vulnerabilities, whereas we traced the vulnerabilities rooted in an improper implementation of security tactics and inspected what their root causes were, how they occurred over time, and efforts to fix them.

Despite the research community efforts to facilitate the design decisions for developing more secure software and to study vulnerabilities from an architectural perspective, there is a gap for an in-depth study that addresses the problem of investigating how security tactics are being incorrectly implemented in the code. Furthermore, to the best of our knowledge, there is no previous work that provides evidence on what the common root causes of such incorrect implementations are and the corresponding efforts to fix them.

VIII. Conclusion

This paper has presented a first-of-its-kind empirical study towards understanding software vulnerabilities related to security tactics. We identified tactical and non-tactical vulnerabilities in three software systems. While most vulnerabilities are non-tactical, on all three systems more than 30% were tactical. Furthermore, while the number of tactical and non-tactical vulnerabilities differ, the rate at which they are reported over time and across releases develops similarly. Finally, when fixing vulnerabilities, code churn and number of affected files are not significantly higher for Chromium and PHP, but for Thunderbird, code churn is slightly higher while the number of affected files is slightly lower for tactical vulnerabilities.

Acknowledgments

This work was partially funded by the US National Science Foundation under grant numbers CCF-1543176, CNS-1629810 and IIP-0968959 under funding from the S2ERC I/UCRC program and US Department of Homeland Security.

References

[1] Top 50 products having highest number of cve security vulnerabilities. https://www.cvedetails.com/top-50-products.php.

[2] S. Al-Azzani and R. Bahsoon. Secarch: Architecture-level evaluation and testing for security. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, pages 51–60. IEEE, 2012.

[3] I. Aspect Security. 2014 state of developer application security knowledge report. https://www.aspectsecurity.com/the-2014-state-of-developer-application-security-knowledge-report-landing-page.

[4] A. Barth, C. Jackson, C. Reis, T. Team, et al. The security architecture of the chromium browser, 2008.

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

[6] B. J. Berger, K. Sohr, and R. Koschke. Extracting and analyzing the implemented security architecture of business applications. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 285–294. IEEE, March 2013.

[7] A. Bosu and J. C. Carver. Peer code review to prevent security vulnerabilities: An empirical evaluation. In *7th International Conference on Software Security and Reliability Companion*, pages 229–230. IEEE, 2013.

[8] H. Cervantes, R. Kazman, J. Ryoo, D. Choi, and D. Jang. Architectural approaches to security: Four case studies. *IEEE Computer*, 49:60–67, 2016.

[9] Q. Feng, R. Kazman, Y. Cai, R. Mo, and L. Xiao. Towards an architecture-centric approach to security analysis. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 221–230, 2016.

[10] E. B. Fernandez, H. Astudillo, and G. Pedraza-García. Revisiting architectural tactics for security. In *9th European Conference on Software Architecture (ECSA)*, pages 55–69, 2015.

[11] M. Hafiz and M. Fang. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering*, pages 1–40, 2015.

[12] S. T. Halkidis, N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Architectural risk analysis of software systems based on security patterns. *IEEE Transactions on Dependable and Secure Computing*, 5(3):129–142, 2008.

[13] T. Heyman, R. Scandariato, and W. Joosen. Reusable formal models for secure software architectures. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 41–50. IEEE, 2012.

[14] C. Izurieta and J. M. Bieman. How software designs decay: A pilot study of pattern evolution. In *ESEM*, pages 449–451, 2007.

[15] J. C.-H. Mehdi Mirakhorli. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Trans. Software Eng.*, 2015.

[16] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[17] M. Mirakhorli and J. Cleland-Huang. Modifications, tweaks, and bug fixes in architectural tactics. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 377–380. IEEE Press, 2015.

[18] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, and J. Cleland-Huang. Archie: A tool for detecting, monitoring, and preserving architecturally significant code. In *CM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.

[19] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic centric approach for automating traceability of quality concerns. In *International Conference on Software Engineering, ICSE (1)*, 2012.

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

[21] J. Ryoo, R. Kazman, and P. Anand. Architectural analysis for security. *IEEE Security & Privacy*, (6):52–59, 2015.

[22] J. Ryoo, B. Malone, P. A. Laplante, and P. Anand. The use of security tactics in open source software projects. *IEEE Transactions on Reliability*, 65(3):1195–1204, Sept 2016.

[23] E. Taspolatoglu and R. Heinrich. Context-based architectural security analysis. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 281–282, 2016.

[24] J. Verner, J. Sampson, V. Tosic, N. A. A. Bakar, and B. Kitchenham. Guidelines for industrially-based multiple case studies in software engineering. In *Third IEEE International Conference on Research Challenges in Information Science*, pages 313–324, 2009.

[25] C. Visaggio. Session management vulnerabilities in today's web. *IEEE Security & Privacy*, (5):48–56, 2010.

[26] R. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

[27] E. Yuan and S. Malek. Mining software component interactions to detect security threats at the architectural level. In *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*, pages 211–220. IEEE, 2016.