5-2016

Toward Establishing a Catalog of Security Architecture Weaknesses

Joanna Cecilia da Silva Santos
jds5109@rit.edu

Toward Establishing a Catalog of Security Architecture Weaknesses

by

Joanna Cecilia da Silva Santos

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Software Engineering

Supervised by

Dr. Mehdi Mirakhorli

Department of Software Engineering

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 2016

The thesis "Toward Establishing a Catalog of Security Architecture Weaknesses" by Joanna Cecilia da Silva Santos has been examined and approved by the following Examination Committee:

Dr. Mehdi Mirakhorli
Assistant Professor
Thesis Committee Chair

Dr. J. Scott Hawker
Associate Professor

Dr. Stephanie Ludi
Professor

Dedication

To Clarice (mother), Messias (father) and Lucas (brother) for their love and assistance.

To my friends, who make my days happier and more enjoyable.

To professor Anselmo, whose classes triggered my passion for STEM fields.

To professor Admilson, whose lessons and advice positively contributed in my academic life and encouraged me to pursue an academic career.

Acknowledgments

I am extremely thankful to Dr. Mehdi Mirakhorli for the opportunity, guidance, patience and support while pursuing my Master's degree. I am also thankful to Dr. Hawker for his valuable comments on my thesis work. I also would like to thank Jairo Vidal for his review over the CAWE catalog. I am grateful to Waleed Zogaan, Raghuram Gopalakrishnan, Nuthan Munaiah and Harold Valdivia-Garcia for their relevant feedback over my thesis presentation. I am also grateful to CAPES-Brazil for sponsoring my graduate studies.

Abstract

Toward Establishing a Catalog of Security Architecture Weaknesses

Joanna Cecilia da Silva Santos

Supervising Professor: Dr. Mehdi Mirakhorli

The architecture design of a software system plays a crucial role in addressing security requirements early in the development lifecycle through forming design solutions that prevent or mitigate attacks in a system. Consequently, flaws in the software architecture can impact various security concerns in the system, thereby introducing severe breaches that could be exploited by attackers. In this context, this thesis presents the new concept of Common Architectural Weakness Enumeration (CAWE), a catalog that identifies and categorizes common types of vulnerabilities rooted in the software architecture design and provides mitigation techniques to address each of them. Through this catalog, we aim to promote the awareness of architectural flaws and stimulate security design thinking to developers, architects and software engineers. This work also investigates the reported vulnerabilities from four real and complex software systems to verify the existence and implications of architecture weaknesses. From this investigation, we noted that a variety of breaches are indeed rooted in the software design (at least 35% in the investigated systems), providing evidence that architectural weaknesses frequently occurs in complex systems, resulting in medium to high severe vulnerabilities. Therefore, a catalog of such type of weaknesses can be useful for adopting proactive approaches to avoid design vulnerabilities.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software security, which is concerned about engineering a software that remains working under malicious attacks, is a relatively new field [27] and has been pointed as an afterthought: firstly, the software is released, then the security problems that are found during its usage are fixed. This fact can be observed when reading the release notes of a software product, which usually indicate some patches to fix vulnerabilities. The problem with this reactive approach is that there could be potential consequences with the exploitation of the discovered breaches (such as brand reputation damage and money losses [36]).

The architecture of a software exhibits the set of design decisions made by the architect to satisfy quality requirements, such as availability, performance, security and so forth. Thus, software architecture design is the first and the fundamental step to address security concerns early in the software development lifecycle. To satisfy a security concern, an architect must consider alternate design solutions, evaluate their trade-offs, identify the risks and select the best option [4]. These design decisions are often based on well-known security patterns [3, 4, 20, 21], which provide reusable solutions for enforcing the required authentication, authorization, confidentiality, data integrity, privacy, accountability, availability, safety and non-repudiation requirements, even when the system is under attack.

Previous estimations [27] indicate that roughly 50% of security problems are the result of software design flaws, such as miss-understanding architecturally significant requirements, poor architectural implementation, violation of design principles in the source code and degradation of the security architecture. Since the software security architecture is one of the main factors to achieve security goals within a software, flaws in the architecture can have a greater impact on the security aspect of a software than purely coding bugs [1]. *Design flaws* in the architecture of a software system mean that successful attacks could lead to enormous consequences.

Design flaws (or only “flaws”) are different from *Bugs*, as the latter are more code-level (such as buffer overflows caused by miscalculations) while the former are at a higher-level of abstraction, require a deeper analysis on the software’s design and are much more subtle than bugs [1]. Although a software system will always have bugs, recent studies show that the security of many software applications is breached due to flaws in the architecture [1, 32].

Architectural flaws are results of inappropriate design choices in early stages of software development, incorrect implementation of security patterns, or degradation of security architecture over time [29, 40]. An example of an architectural flaw is the “Use of Client-Side Authentication” [10], in which a client/server product performs authentication within the client code, but not in the server code. This design decision allows the authentication feature to be bypassed via a modified client that omits the authentication check. It creates a flaw in the security architecture that can be successfully exploited by an intruder with reverse-engineering skills. Another instance of an architectural flaw is the “Cleartext Transmission of Sensitive Information”, in which a sensitive information (*e.g.*credit card number) is exchanged without encryption [15]. In this flaw, an attacker could steal this information by capturing the messages being transmitted in the network.

Even though there are many techniques and practices that help to develop a secure software system (such as threat modeling [39], penetration testing [2], static and dynamic code analysis [6, 35], *etc.*), there have not been many previous works in the literature that approach security from an architectural perspective. A recent effort is the IEEE Center for Secure Design launched by the IEEE Computer Society. However, as of today, there are not many examples of design flaws obtained or published yet that can help architects and developers to learn and avoid such flaws.

1.1 Objectives

As discussed in the introduction, there is still a need for tackling this problem in terms of design flaws [32]. Hence, the primary goal of this work is to approach security according to an architectural point of view through the new concept of *Common Architectural Weakness Enumeration* (CAWE): a catalog of architectural weaknesses (i.e. security issues rooted in the software security architecture). The overall idea is to provide an artifact to help developers and architects in understanding the consequences of having such breaches in their software and identify possible ways to avoid them to occur. This catalog was built on top of a previous library of Common Software Weaknesses Enumeration (CWE) [1], which is a community-developed enumeration of common types of vulnerabilities (weaknesses) maintained by the MITRE Corporation. This library, however, does not categorize weaknesses based on their architectural impacts and does not clearly distinguish architecture-related issues from purely programming issues.

The second objective is to report the results of an analysis of the vulnerabilities from four large complex systems (namely Linux Kernel, Chrome, Thunderbird, and PHP) and to demonstrate instances of architectural weaknesses within these systems. Besides that, this work discusses the practical aspects of CAWE catalog, indicating how it can be used in activities within the software development lifecycle and how it can be integrated into the existing MITRE's library.

1.2 Research Questions

This work addresses the following research questions:

- *RQ1: What security weaknesses are rooted in the software architecture?*

  The answer to this question proposes to identify, among the documented types of security weaknesses from the MITRE's library, which ones can happen due to a

[1] `http://cwe.mitre.org/`

design flaw or an architectural drift.

- *RQ2: What security patterns are more likely to have associated vulnerabilities?*

  By knowing the answer of R1, we aim to classify each security weakness per security pattern. This way, we can spot the security patterns that are more likely to introduce vulnerabilities when improperly applied and in which ways these vulnerabilities may occur.

- *RQ3: What are the most common design issues in real software systems?*

  In this research question, we aim to verify the proportion of vulnerabilities in existing systems are rooted in their architecture, identifying what are the most common architectural flaws in software systems.

1.3 Thesis Organization

This work is organized as follows: Chapter 2 explores the concepts of security patterns and architectural flaws. Chapter 3 presents the CAWE catalog, its structure, and creation process. Chapter 4 presents the answers to all the three research questions and provides some real examples of architectural weaknesses. Chapter 5 discusses ways to use the catalog within an IT organization and to integrate the CAWE catalog to the MITRE's library. Chapter 6 presents the related works. Chapter 7 explains the threats to the validity of this work. The conclusion and future work are presented in the last Chapter.

Chapter 2

Designing for Security

Security principles need to be implemented from the ground up to ensure an application is secure. During requirements analysis, malicious practices are taken for granted, and requirements engineers identify all the use cases which are interests of an attacker. During architecture design, architects carefully analyze these requirements and adopts appropriate security patterns to resist, detect and/or recover from attacks [34].

2.1 Security Patterns

Security patterns are the building blocks of a security architecture. A combination of patterns are required to be implemented to deliver a secure system in which the legitimate users are properly authenticated; access controls are applied to enforce that only authorized users can access their designated functionality; user activities are audited, so recovering from a malicious activity is feasible; information integrity is preserved and similar security characteristics are addressed.

Several researchers and practitioners have worked on collecting and organizing security patterns [34]. For instance, Hafiz *et al.*. [20] discussed an organization of a subset of patterns according to different classification schemes (*e.g.* based on the application context and the STRIDE [22] models) and later showed a pattern language to indicate the relationship among those patterns [21]. Likewise, Kienzle *et al.* [25] presented a repository of security patterns. Besides research papers, many textbooks described in details security patterns either specific to a software technology (*e.g.* J2EE) [38] or to general use, regardless of software domain and underlying technical choices [34].

While these security patterns provide a well-formed solution to address various security concerns, if these patterns are not adopted and implemented carefully, they can result in severe breaches in a security architecture. In the subsequent section, we discuss several ways that a security architecture can be flawed or degraded.

2.2 Flaws in a Security Architecture

The IEEE Center for Secure Design has been focusing on identifying the most relevant design flaws based on experiences in the industry, academia, and government. Currently, it identified top ten design flaws that are caused either by a weak design (i.e. an incomplete architecture whose security mechanisms can be bypassed by attackers) or due to lack of design decisions (*e.g.* not having an authorization enforcement in the system) [1]. Even if the architecture fulfills all security requirements appropriately, previous studies have shown that the architecture can erode as the software evolves or may be wrongly implemented in the code [24, 29, 40]. Consequently, based on these observations, we can classify architectural flaws to Omission, Commission and Realization Flaws:

- **Omission Flaws** are caused by decisions that were never made (e.g. ignoring a security requirement or potential threats). A common omission design flaw is to store a password in a file without encryption [8]. In this flaw, the architect overlooks the need of protecting sensitive data from unauthorized users and assumes that attackers would never have access to the file, thereby considering that the password stored in plaintext would not correspond to a compromise of the system. However, this lack of encryption can open the system to attacks, because anyone, who has granted read access to the file, will be able to read all the stored passwords. Similarly, the flaw of having cleartext transmission of sensitive information [15] also can result in a steal of sensitive information by attackers that can capture the data being exchanged in the communication channel.

- **Commission Flaws** refer to the design decisions which were made and could lead to undesirable consequences. An example of such type of flaw is “Client side authentication” [10], mentioned earlier in the introduction. While architects have made a design decision, the flaw in this design will enable attackers to bypass the authentication through implementing a modified client that does not have the authentication check. Another example of such flaw is “Using a Weak Cryptography for Passwords” to achieve better performance while maintaining data confidentiality. In this flaw, the passwords are stored with an obfuscation mechanism that is computationally less complex but easier for attackers to guess [13]. Consequently, such improper design choice makes it possible that attackers to recover the passwords via an exhaustive search.

- **Realization Flaws** are the design decisions that are correct (i.e. satisfies the software’s security requirements), but its implementation suffers from a coding mistake. For example, a developer fails to perform authenticity check in the critical parts of the system [14]. In another instance, the developers incorrectly sanitize special elements in user-provided inputs, which can lead to many consequences, such as crashes (denial of service) or bypass of protection mechanisms [12].

In the Chapter that follows, we present the CAWE catalog which documents such types of flaws in a systematic way.

Chapter 3

Catalog of Architectural Weaknesses

The MITRE Corporation, with the support of the National Cyber Security Division at US Department of Homeland Security (DHS), maintains a collection of common software weaknesses (http://cwe.mitre.org/). This collection contains over 1,000 software weaknesses, but these vulnerabilities are not categorized based on their architectural impacts and implications. Thus, we classified those weaknesses architecture-related vulnerabilities. As a result, we have developed the *Common Architectural Weakness Enumeration (CAWE)*, a catalog that enumerates common architectural flaws in a software system that can lead to a security vulnerability.

The next sections of this Chapter detail the systematic process followed to create the catalog and how the data within is structured.

3.1 Creating the Catalog

As briefly mentioned before, the CAWE catalog was built on top of the MITRE's compilation of software weaknesses (CWE collection). Figure 3.1 shows a structural overview of the CWE collection, in which we only showed some of the elements for the sake simplicity [1]. As noted in this figure, each entry in the CWE collection can be of four types: *View* (for grouping weaknesses in a given perspective), *Category* (used to categorize weaknesses based on a common attribute), *Weakness* (an actual security issue) and *Compound Element*

[1]The complete structure of the CWE collection can be found at MITRE's Website: https://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd

(a security issue due to the occurrence of other weaknesses in a time sequence). For each *Weakness* and *Compound Elements* types, the collection provides information about the issue, such as its description, mitigation techniques, code examples, and so forth.

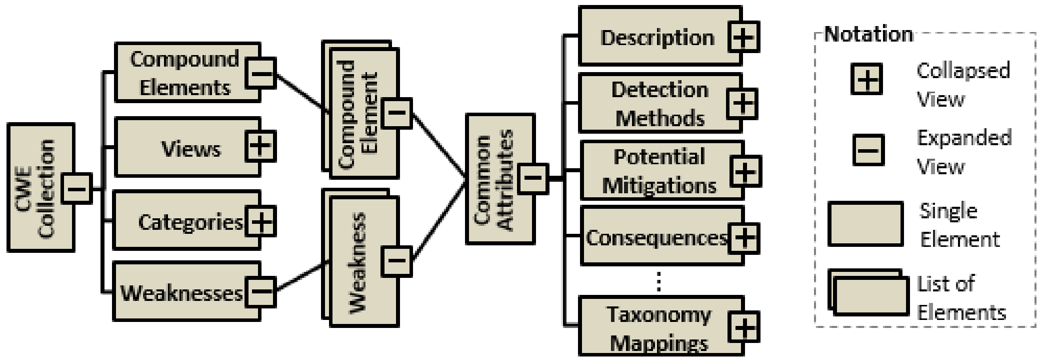


Figure 3.1: Partial Structure of MITRE‘s CWE Collection with its Upper Elements.

Since the CWE collection is available as an XML (eXtensible Markup Language) document in the MITRE’s Web site [2], we retrieved a list of all entries that are of type *Weakness* or *Compound Elements*. The other element types (*Categories* and *Views*) were not included as they serve more as a grouping of weaknesses rather than providing details about a specific type of vulnerability. Then, we analyzed each collected entry to classify whether it is rooted in the security architecture. Once we considered that an entry has an architectural implication, we further investigated it to verify (i) which security pattern(s) can be related to the weakness and (ii) and how the related pattern(s) may be impacted.

To increase the accuracy of this mapping, a second graduate student in Software Engineering, who was familiar with security patterns and software architecture design, peer reviewed the catalog. Thus, this individual inspected the entries from the MITRE’s collection and performed the same steps described before (classification of the entries into architectural/non-architectural and identification of the related security patterns), sharing the rationale behind each mapping. After this peer review process, we established the *CAWE catalog*, which provides a collection of architectural weaknesses.

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

3.2 Catalog Structure

To accommodate all the gathered information and to provide a rich knowledge base about architectural weaknesses, we structured the CAWE catalog as shown in Figure 3.2. In this figure, we note that the catalog organizes the design flaws based on the security patterns that are impacted by them. This way, the CAWE catalog encompasses a list of `Impacted Pattern` elements, which are used to provide the details about each security pattern (its context, problem, solution, and so forth) and the relationships between the pattern and one or more design flaws.

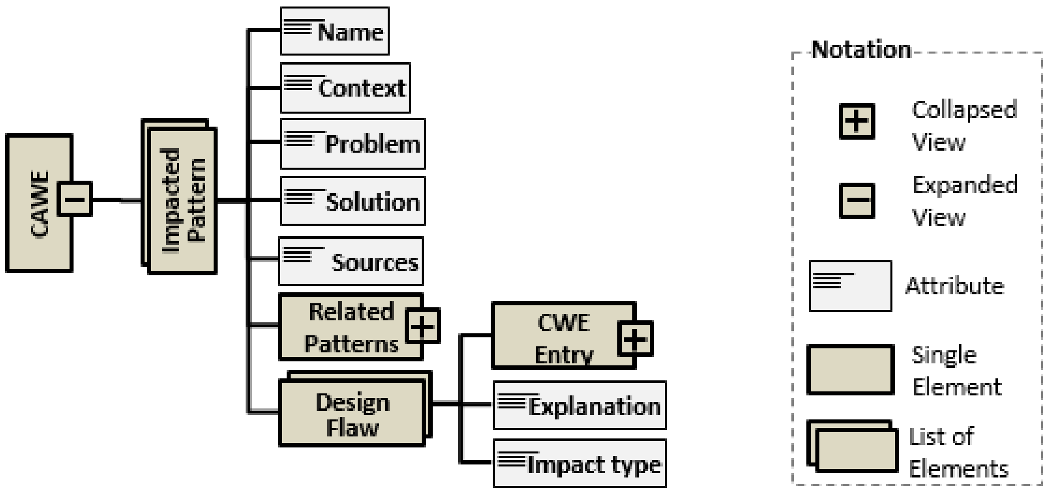


Figure 3.2: Structure of the CAWE catalog .

As discussed in Chapter 2,a missing design choice, an incorrect implementation of an architectural choice or an improper design decision can cause flaws in the software. Therefore, each `Design Flaw` element contains an `impact type` attribute, which indicates whether the flaw is an *omission*, *commission* or a *realization* flaw, and an `explanation` attribute, that describes how the associated security pattern is impacted. Every `Design Flaw` element also points to one entry from the CWE collection. Such link is used to provide detailed information about the architectural weakness.

3.3 Catalog Overview

The CAWE catalog is publicly available online at `http://design.se.rit.edu/cawe`. It currently contains 384 flaws that were categorized based on their impacts over 39 security patterns. Table 3.1 shows an entry from the CAWE catalog, which presents a weakness due to an incorrect implementation (*Realization Flaw*) of the *Secure Session Management* pattern [34] (some text in this Table was hidden for the sake of clarity). From this table, we observe that each CAWE instance refers to an entry from the MITRE's library of software weaknesses. This reference contains the detailed information about the design flaw, such as a textual description, source code examples, mitigation techniques and detection methods of the weakness. Moreover, a CWE entry can indicate how this flaw can be exploited by an intruder through pointing to external entries from the Common Attack Pattern Enumeration and Classification (CAPEC) [11], which is a dictionary of known attack patterns.

Through this catalog architects and developers can learn to avoid common architectural issues in the software. Since the catalog is organized around security patterns, architects and developers can easily identify potential flaws related to a particular security pattern. For instance, a common design decision is to use encryption algorithms from libraries to store/exchange data [32]. However, developers and architects may overlook the properties of these encryption algorithms, making incorrect assumptions about their usage. Since understanding the algorithm is crucial to properly secure data, our catalog enumerates a set weaknesses that can guide them to obtain such knowledge. Examples of such weaknesses are: "CAWE-328 Reversible One-Way Hash" and "CAWE-780 Use of RSA Algorithm without Optimal Asymmetric Encryption Padding".

Table 3.1: An Example of an Architectural Weakness from the Catalog

Impacted Pattern	**Name**	Secure Session Management
	Context	(...)
	Problem	(...)
	Solution	(...)
	Related Patterns	(...)
Design Flaw	**Impact Type**	Realization Flaw
	Explanation	This incorrect implementation of the session management can lead to information disclosure. (...)

	CWE Entry	Title: Exposure of Data Element to Wrong Session
		Description: The product does not sufficiently enforce boundaries between the states of different sessions, causing data to be provided to, or used by, the wrong session. (...)
		Demonstrative Example: The following Servlet stores the value of a request parameter in a member field and then later echoes the parameter value to the response output stream. While this code will work perfectly in a single-user environment, if two users access the Servlet at approximately the same time, it is possible for the two request handler threads to interleave in the following way: Thread 1: assign “Dick” to name Thread 2: assign “Jane” to name Thread 1: print “Jane, thanks for visiting!” Thread 2: print “Jane, thanks for visiting!”. Thereby, showing the first user the second user’s name.
		`public class GuestBook extends HttpServlet {`
		`  String name;`
		`  protected void doPost(HttpServletRequest req,`
		`                        HttpServletResponse res) {`
		`    name = req.getParameter("name");`
		`    ...`
		`    out.println(name + ", thanks for visiting!");`
		`  }`
		`}`
		Potential Mitigations:
		● *Architecture and Design Phase:* Protect the application’s sessions from information leakage. Make sure that a session’s data is not used or visible by other sessions.
		● *Testing Phase:* Use a static analysis tool to scan the code for information leakage vulnerabilities (e.g. Singleton Member Field).
		Attack Patterns:
		● CAPEC-59 Session Credential Falsification through Prediction
		● CAPEC-60 Reusing Session IDs (aka Session Replay)
		(...)

Chapter 4

Results and Discussion

4.1 RQ1: Architectural Weaknesses

RQ1: What security weaknesses are rooted in the software architecture?

Once we consolidated the CAWE catalog, we observed that, among the 727 entries of type Weaknesses or Compound Elements from the CWE library, there were 384 of them with architectural implications. A full list of these security architectural weaknesses is provided in our Web site (`http://design.se.rit.edu/cawe`).

4.2 RQ2: Most Impacted Security Patterns

RQ2: What security patterns are more likely to have associated vulnerabilities?

Table 4.1 reports the total number of design weaknesses identified for each security pattern, based on the CAWE catalog. In this table, we observe that, in the case of “Chroot Jail” pattern, we found 70 common flaws that can impact the robustness of this security pattern. For most patterns, this number is smaller, except for “Intercepting Validator” which has 166 flaws related to its implementation. The reason for such a large number is that there are a large number of known security weaknesses which can result in breaches in the system due to issues in the input data [32].

Table 4.1: Total Number of Identified Design Flaws for Each Security Pattern.

Pattern Name	#	Pattern Name	#	Pattern Name	#
Administrator Objects	25	Exception Shielding	31	Policy Delegate	7
Assertion Builder	6	Firewall	1	Policy Enforcement Point	34
Audit Trails	8	Hidden Metadata	3	Protected System	16
Authenticator	41	HMAC	12	Protection Reverse Proxy	2
Authorization	54	Information Obscurity	5	Role based access control	18
Batched Routing	3	Intercepting Web Agent	39	Secure Base Action	3
Brokered Authentication	29	Intercepting Validator	166	Secure Communication	15
Chroot Jail	70	Message Inspector	13	Secure Message Router	5
Control Process Creator	12	Message Interceptor Gateway	16	Security Provider	36
Credential	17	Minefield	2	Security Proxy	3
Demilitarized Zone	1	Morphed Representation	7	Session Management	8
Encrypted Storage	33	Obfuscated Transfer Object	19	Single Sign On	5
Error Detection and Correction	6	Password Synchronizer	20	Subject Descriptor	7

4.3 RQ3: The Most Common Design Issues

RQ3: What are the most common design issues in real software systems?

To respond to this third research question, we investigated the reported vulnerabilities from four real and complex systems. The software systems chosen as case studies were: the **Linux Kernel** (contains the core functions of the Linux Operating System), the **Google Chrome** (a Web browser), the **Mozilla Thunderbird** (an application for managing email and news feeds) and **PHP** (the interpreter of the PHP language). These four systems were selected because they are among the open source projects with a higher number of known vulnerabilities and from distinct domains, which can enrich this discussion of architectural flaws.

4.3.1 Vulnerabilities Dataset

To conduct such analysis, two types of information are required: the security breaches in each case study and their respective root causes. To do so, we queried the National

Vulnerability Database (NVD) [1] to obtain all the reported vulnerabilities for each case study. As a response, the NVD provides the vulnerabilities of a project as a list of CVE (Common Vulnerabilities and Exposures) instances [7]. In short, these CVE instances identify the severity of the vulnerability, the date it was reported, a description and links to external resources with more details about the breach (such as URLs to threads discussion about the issue, bug tracking systems, etc.). Each CVE instance sometimes also indicate its root cause through a *tag* that points to a documented software weakness from MITRE's collection (CWE entry). Thus, in the presence of this link, we verify whether the problem is an *architectural flaw* or a *purely coding bug* through comparing the referred CWE instance against our CAWE catalog. For the CVEs without any further information about its causes (i.e. without a *tag*), we manually inspected them to obtain such information.

4.3.2 Results

Table 4.2 shows the size of these projects (in terms of the number of files in the latest version) and the amount of vulnerabilities collected for each project. This table also presents the total number of CVEs that have an explicit CWE tag and the amount of CVEs that we manually tagged. It is important to highlight that there were vulnerabilities that were deprecated (i.e. invalid) or external to the architecture of the case studies (such as vulnerabilities in applications that execute in a Linux environment but not in the Linux Kernel itself). Therefore, we discarded these vulnerabilities from our dataset, so Table 4.2 reports only the instances that are under the scope of our analysis.

Table 4.2: Overview of the Vulnerability Dataset

Project	# Files	#CVEs	#With Tags	#No Tags	#Arch. CVEs	#Non-arch. CVEs
Chrome	46,544	1251	1067	184	441	810
Thunderbird	19617	704	517	187	310	394
PHP	2028	425	267	158	214	211
Linux Kernel	39787	1342	918	424	540	802

[1]https://cve.mitre.org

In this table, we observe that approximately 50% of the reported vulnerabilities for PHP had an impact on its security architecture, being the project with the highest percentage of architectural issues. In the other systems, this percentage of architectural vulnerabilities was 35% for Chrome, 44% for Thunderbird and 40% for Linux Kernel.

Table 4.3 shows the most common security issues in those cases studies as well as the severity level of these problems (Low, Medium, High or Critical), as provided by the NVD based on the Common Vulnerability Scoring System (CVSS) version 2 [28]. From this table, we note that most of the architectural issues in those case studies is a problem in failing to implement the Intercepting Validator pattern properly (CAWE-20, CAWE-79, CAWE-94, CAWE-134, CAWE-138 and CAWE-158). Failing to validate consistently the user-provided data can lead to a variety of consequences, such as crashes (denial of service) and leakage of sensitive information. We also observe that issues related to enforcing access control (CAWE-200, CAWE-274, CAWE-280, CAWE-284, and CAWE-782) are also common among those systems. These weaknesses are mostly caused by an incorrect implementation of patterns related to permission management such as “Authorization”, “Role-based access control” and “Policy Enforcement Point”. Lastly, we observe that the architectural flaws expose these systems to consequences that are at least at a “Medium” severity level.

One of the reasons that resulted in such high number of input validation issues is due to the fact that some architectural flaws can only occur in certain contexts. For example, the Client Side Authentication Flaw only occurs in software with the client-server architecture. However, regardless of the software domain, a software will be provided which may be valid or malformed (intentionally by attackers or unconsciously by misuses from legitimate users) thereby, input validation issues can happen in any software system.

The next section discusses examples from these case studies of architectural weaknesses.

Table 4.3: The Most Common Architectural Weaknesses in the Case Studies.

	Top 5 Architectural Weaknesses	#CVEs	Severity
Chrome	CAWE-20 Improper Input Validation	170	High
	CAWE-284 Improper Access Control	51	Medium
	CAWE-200 Information Exposure	36	Medium
	CAWE-274 Improper Handling of Insufficient Privileges	34	Medium
	CAWE-79 Improper Neutralization of Input During Web Page Generation	26	Medium
PHP	CAWE-20 Improper Input Validation	79	Medium
	CAWE-280 Improper Handling of Insufficient Permissions or Privileges	36	Medium
	CAWE-158 Improper Neutralization of Null Byte or NUL Character	13	Medium
	CAWE-134 Use of Externally-Controlled Format String	10	High
	CAWE-138 Improper Neutralization of Special Elements	7	High
Thunderbird	CAWE-20 Improper Input Validation	71	Medium
	CAWE-284 Improper Access Control	54	Medium
	CAWE-94 Improper Control of Generation of Code ('Code Injection')	34	High
	CAWE-79 Improper Neutralization of Input During Web Page Generation	34	Medium
	CAWE-200 Information Exposure	25	Medium
Linux	CAWE-20 Improper Input Validation	230	Medium
	CAWE-274 Improper Handling of Insufficient Privileges	40	Medium
	CAWE-284 Improper Access Control	38	Medium
	CAWE-391 Unchecked Error Condition	9	Medium
	CAWE-782 Exposed IOCTL with Insufficient Access Control	6	Medium

4.4 Examples of Architectural Weaknesses

The Secure Session Management pattern is concerned about managing sessions, which are a set of activities performed over a limited period by a specified user. The primary goal of this pattern is to keep track of who is using the system at a given time through managing a session object that contains all relevant data associated with the user and the session [34]. In this pattern, every user is assigned an exclusive identifier (*Session ID*), which is utilized for both identifying users and retrieving the user-related data. Since session IDs are a sensitive information, this pattern may be affected by two main types of attacks: session hijacking (an attacker impersonate a legitimate user through stealing or predicting a valid session ID) and session fixation (an attacker has a valid session ID and forces the victim to use this ID).

The session hijacking can be facilitated by the architectural flaw of not securing the storage of session identifiers. Such flaw can be observed in the "session" module of the PHP language:

CVE ID: CVE-2002-0121
Description: PHP 4.0 through 4.1.1 stores session IDs in temporary files whose name contains the session ID, which allows local users to hijack web connections.

Main Impacted File: ext/session/mod_file.cc

```
46.  #define FILE_PREFIX "sess_"
(...)
108. strcat(buf, FILE_PREFIX);
109. strcat(buf, key);
```

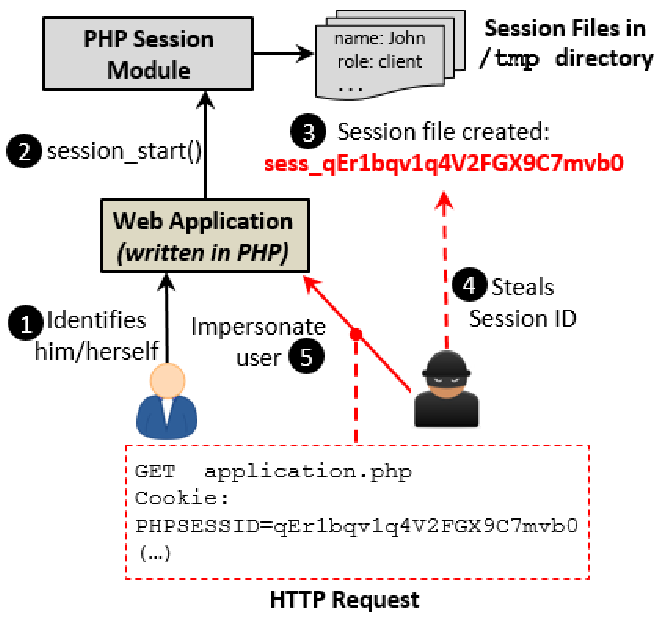
Related CAWEs:

CAWE-311: Missing Encryption of Sensitive Data

CAWE-538: File and Directory Information Exposure

Per this description we note that PHP was designed to store each session data in files in a temporary directory without using a security mechanism for storing these session files (such as encryption). When closely inspecting the source code of PHP in version 4.0, we observe that the `mod_file.cc` names every session file as "sess_xyz" (where "xyz" is the session ID), as shown in the code snippet presented above (where `buf` is a variable later used when creating the session files).

Figure 4.1 shows a scenario in which the flaw could be exploited. First, a legitimate user successfully identifies him/herself to the application. This causes the Web application, written in PHP, to start a session for the user through invoking the `session_start()` from the PHP's `session` module. Then, the `session` module in the PHP assigns a session ID for the user, and it creates a new file named as "sess_qEr1bqv1q4V2FGX9C7mvb0" to store the data about the user's session. At this point, the security of the application is compromised when an attacker observes the session file name and realizes that the user's session ID is equals to "qEr1bqv1q4V2FGX9C7mvb0". Subsequently, the attacker can impersonate the user through sending a cookie (`PHPSESSIONID`) in a HTTP request with this stolen Session ID. The Web application, after calling functions from the PHP's `session`, verifies that the session ID provided matches with the user's data so, the application considers that a legitimate user is making the request.



```
GET  application.php
Cookie:
PHPSESSID=qEr1bqv1q4V2FGX9C7mvb0
(...)
```

Figure 4.1: Session Hijacking in PHP.

From this scenario, we note that such architectural weakness can lead to many consequences. First, if the user has an administrative role in the application, the attacker will be able to perform all the administrative tasks. Second, the attacker may be able to read the contents of the session file, thereby accessing the data about the user, which may be sensitive. It is important to highlight that such flaw affects not only the Secure Session Management but also other security patterns (e.g. Authentication and Authorization) which use the Secure Session Management for performing authentication and access control of users.

The excerpt below shows an example from PHP of an architectural weakness that facilitates the session fixation:

CVE ID: CVE-2011-4718
Description: Session fixation vulnerability in the Sessions subsystem in PHP before 5.5.2 allows remote attackers to hijack web sessions by specifying a session ID.

Main Impacted File: ext/session/mod_files.c

```
69. static int ps_files_valid_key(const char *key)
70. {
(...)
76.   for (p = key; (c = *p); p++) {
77.     /* valid characters are a..z,A..Z,0..9 */
78.     if (!((c >= 'a' && c <= 'z')
79.           || (c >= 'A' && c <= 'Z')
80.           || (c >= '0' && c <= '9')
81.           || c == ','
82.           || c == '-')) {
83.             ret = 0;
84.             break;
85.     }
86.   }
87.
88.   len = p - key;
89.
90.   /* Somewhat arbitrary length limit here, but should be way more than
91.      anyone needs and avoids file-level warnings later on if we exceed  MAX_PATH */
92.   if (len == 0 || len > 128) {
93.     ret = 0;
94.   }
95.
96.    return ret;
97. }
(...)
146. static void ps_files_open(ps_files *data, const char *key TSRMLS_DC)
147. {
(...)
158.   if (!ps_files_valid_key(key)) {
159.     php_error_docref(NULL TSRMLS_CC, E_WARNING, "The session id is too long or
contains illegal characters, valid characters are a-z, A-Z, 0-9 and '-,'");
160.     PS(invalid_session_id) = 1;
161.     return;
162.   }
(...)
201 }
```

Related CAWEs:

CAWE-384 Session Fixation

When verifying the session implementation in the source code of PHP version 5, we note that there is an incorrect implementation (i.e. a realization flaw) in the PHP's session module that accepts uninitialized session IDs before using it for authentication/authorization purposes. In fact, in the line 158 shown above, the function `ps_files_valid_key()` does not correctly validate the session ID. This function only checks whether it contains a valid charset and has a correct length but does not verify whether the ID exists associated to the client performing the HTTP request.

Figure 4.2 shows how this architectural vulnerability is exploited. The attack starts with the attacker establishing a valid session ID (steps 1 to 4). Next, the attacker induces the user to authenticate him/herself in the system using the attacker's session ID (steps 5 and 6).

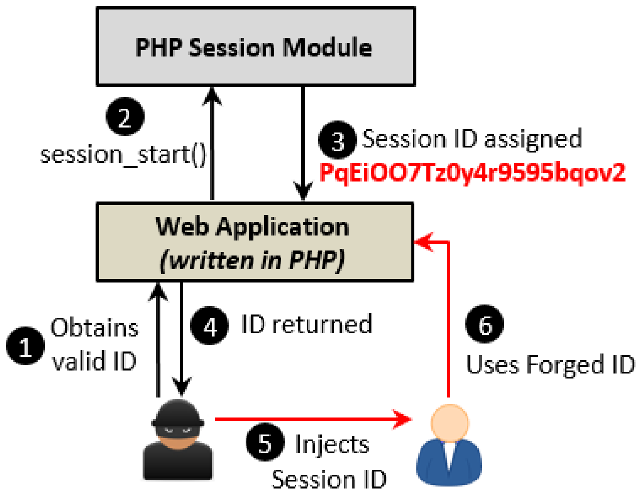


Figure 4.2: Session Fixation Exploitation in PHP.

A common architectural flaw across all case studies is an improper implementation of the validation mechanism for data before its usage in the software. A real example of this design flaw can be observed in Thunderbird:

CVE ID: CVE-2009-2408
Description: Mozilla Network Security Services (NSS) before 3.12.3, Firefox before 3.0.13, Thunderbird before 2.0.0.23, and SeaMonkey before 1.1.18 do not properly handle a '
0' character in a domain name in the subject's Common Name (CN) field of an X.509 certificate, which allows man-in-the-middle attackers to spoof arbitrary SSL servers via a crafted certificate issued by a legitimate Certification Authority. NOTE: this was originally reported for Firefox before 3.5.

Main Impacted File: mozilla/security/nss/lib/certdb/certdb.c

```
1390. cert_VerifySubjectAltName(const CERTCertificate *cert, const char *hn)
1391. {
(...)
1446.   int cnLen = current->name.other.len;
1447.   if (cnLen + 1 > cnBufLen) {
1448.     cnBufLen = cnLen + 1;
1449.     cn = (char *)PORT_ArenaAlloc(arena, cnBufLen);
(...)
1518. }
```

Related CAWEs:

CAWE-20: Improper Input Validation

In this flaw, the validation mechanism fails to validate a field of an X.509 certificate that contains the NULL terminator ('\0' character). As shown in the code snippet, at line 1446 the length of the CN field is calculated without a proper escaping the string variable. Given that the '\0' is used to indicate the end of a string, the `cnLen` variable will have a smaller value than the actual length of the CN field. This improper validation may allow attackers to get a malicious certificate signed through placing the NULL terminator in the certificate's CN field. This way, users would be tricked to consider that the forged certificate is valid then accepting it as trusted when it is not.

One example of a commission flaw was also observed in the PHP project. Initially, the PHP was designed to have a configuration parameter (`safe_mode`) for enforcing access control of files and directories on the Web server. The security requirement behind this parameter was to ensure that applications written in PHP and running on the same Web server would not be able to access files from each other inadvertently [19]. Thus, once the `safe_mode` was enabled, whenever a PHP script tried to access (read/write) a file in the server, the PHP would: (i) check whether the owner of the script's source code file is the same of the file being requested; (ii) then, grant access to those scripts that satisfies this condition. However, such design is inappropriate to the context of PHP [19]. In this case a better design would be that the enforcement was done in the executing environment (i.e. the Operating System).

Moreover, such enforcement mechanism needed to be implemented throughout the modules of PHP that performed any file-related operations. However, in some cases the developers did not implement the check whether the `safe_mode` was enabled and, then invoking the function that does the access control verification, thereby bypassing the designed access control mechanism. Hence, applications that were relying on this `safe_mode` mechanism would have a breach. Therefore, PHP was later redesigned (version 5.4.0+) to remove this `safe_mode` configuration, outsourcing this access control at the Web server and Operating System level.

Lastly, Chrome and Thunderbird had some vulnerabilities that were not only related to the security architecture but also touched upon usability concerns. These types of vulnerabilities were caused by an incorrect implementation/design of a security concern which introduced a breach that intruders can exploit through “user-assisted” attacks [2]. For example, the CVE-2005-2602 reports a problem in the input validation of a long URI in Thunderbird resulting in a blank address bar, which facilitates phishing attacks.

[2]“User-assisted” attacks refer to exploitations that only succeed when the user perform an unusual specific interaction.

Chapter 5

Application of the CAWE Catalog

Since the CAWE catalog provides detailed information about architectural weaknesses, it can be used to guide architects and developers make appropriate design and implementation decisions to preserve security concerns throughout the software development lifecycle. For example, code reviews, which is a common practice applied in many IT organizations [18], are usually performed through a meeting focused on finding bugs through technical discussions and analysis of the source code and other related artifacts (such as a portion of the requirements document, the architecture, etc). Therefore, the reviewers, who are responsible for inspecting the code, could use the CAWE catalog to check common security issues that happen in the software domain under their review.

Past experiences in industry lead to the creation of security-driven software development processes, which emphasizes security concerns early in the software development lifecycle, such as CLASP (Comprehensive, Lightweight Application Security Process) [30] and Microsoft's SDL (Security Development Lifecycle) [23]. A common aspect of these processes is the recommendation of providing proper training of the employees to develop a common background in software security [41]. With this respect, our catalog could be used to aid such training and promote the awareness of the potential architectural issues that their systems may be exposed.

Moreover, those security-driven processes include two activities for modeling potential threats in the software: *threat modeling* [39] and *design of misuse cases* [26]. Since these two activities are usually done through brainstorming sessions, the CAWE could be used in those sessions for obtaining insights. In fact, existing in experiences in the industry report the usage of threat libraries, built from a small subset of weaknesses from the MITRE's catalog, for aiding this threat modeling process [16].

In addition, architectural risk analysis, which is a systematic approach for evaluating design decisions against quality requirements, could also benefit from our catalog. For architectural risk analysis to be effective, the evaluators need to have a previous knowledge of architectural flaws based on the software context (such as its requirements, architectural patterns applied, etc.). Consequently, the CAWE catalog could be used as guidance when performing such assessment. This guidance is twofold: on one hand, the enumerated flaws of omission could be used as a roadmap for evaluators to identify missing relevant architectural decisions; on the other hand, the categorized flaws of commission and realization allows to spot issues in the current architecture of the system, being the first step to assess its impacts and risks.

5.1 Integration Into MITRE/DHS Collection of CWEs

To support the reuse of our findings and release the list of Common Architecture Weaknesses, we can integrate the CAWE catalog to the existing collection of software weaknesses (CWE) collected by the US Department of Homeland Security along with The MITRE Corporation. This integration will be available online in MITRE's collection.

The CWE library supports introducing new concepts without changes in its structure. This is done through the `Taxonomy Mapping` element (presented in Figure 3.1), which allows that every *Weaknesses* or *Compound Elements* to reference nodes in an external taxonomy that have any sort of conceptual relation. Thus, we can use the `Taxonomy Mapping` element to establish a connection between security patterns and software design flaws, thereby augmenting the CWE collection with software architecture concepts.

For example, the CWE entry "Improper Validation of Integrity Check Value" [9] is related to the "Error Detection and Correction" pattern in the CAWE catalog. Hence, we can connect this CWE entry to this pattern by using a new *Taxonomy_Mapping* node with the values presented in Figure 5.1. From this example we observe that the *Mapped_Node_Name* attribute is used to indicate the impacted pattern, the *Mapped_Node_ID* refers to the identifier of a CAWE entry in the catalog and the *Mapping_Fit* (which is hidden to simplify the image) summarizes the impact of the weakness over the pattern. In this example, the value of *Mapping_Fit* explains that not applying the “Error Detection and Correction” when transmitting data over an unreliable communication channel allows the software to use corrupted data inadvertently.

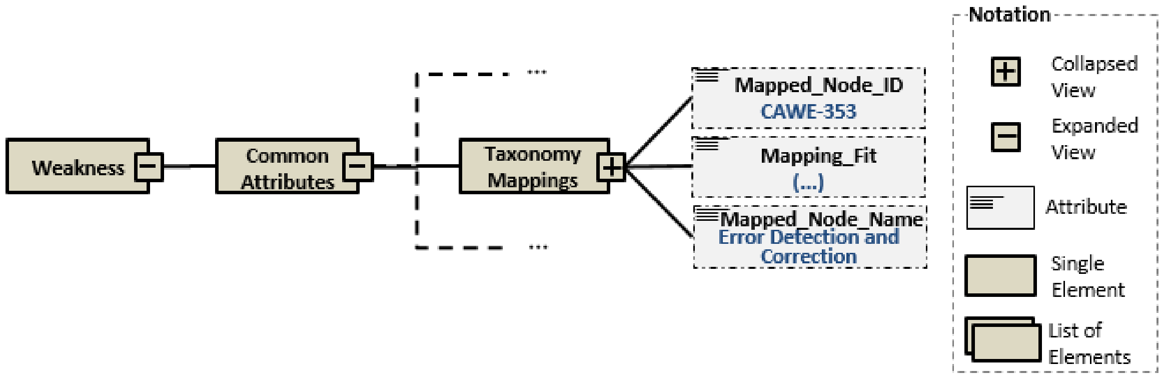


Figure 5.1: Taxonomy Mapping to Help Identify Design Flaws Associated with Security Patterns

Chapter 6

Related Work

Despite the research community efforts to create techniques and tools for developing more secure software, there is a gap for procedures that address the security problem using an architectural point of view [32]. Currently, there are many research and books focused on the identification, categorization, and detailing of security patterns [20, 21, 25, 34, 38]. In this work, however, we take one step further towards observing how these patterns could be compromised when incorrectly implemented or inappropriately designed, thereby filling the existing gap of overcoming security challenges from avoiding architectural flaws.

The usage of security knowledge bases to help developers and engineers in their daily activities have been discussed in the research community. In this matter, security ontologies, which represents knowledge within the security domain, have been created to support some activities (e.g. requirements engineering [37] and quantitative risk analysis [17]). These ontologies, however, does not introduce architectural concepts on it [5].

In this context, similarly to a security ontology, Wu *et al.* [42] proposed the use of *semantic templates* to keep track of the key details related to vulnerabilities. These templates are a structured description of generic patterns of relationship between software components, faults and security consequences built on top of the CWE collection and the CVE dictionary. However, that work does not have a full complete list of templates for all the CWE entries.

In addition to knowledge bases, some works are focused on methodological architecture-related activities for engineering secure software. For example, Pedraza-Garcia *et al.* [31] described activities, tools and notations to specify the security requirements of a software and guide the architect to select the architectural tactics that would better address the requirements. The primary goal of that work was to minimize the impacts of design decisions being informally done and dependent on the architect's experience to select architectural choices. Another example is described by Ryoo *et al.* [33], which proposes a vulnerability-oriented architectural analysis approach that reuses the knowledge from known vulnerabilities as a checklist when evaluating and designing an architecture. Unlike our work, these efforts are focused on helping to engineer a secure software specifically in architectural analysis activities, whereas the scope of our catalog is broader to other software development activities.

In spite of these efforts of creating and sharing security-related knowledge and systematic approaches, to the best of our knowledge, there is no previous work that provided a catalog of architectural weaknesses.

Chapter 7

Threats to Validity

The threats to validity can be classified as *construct*, *internal* and *external* validity. Below we explain the threats that may have impacted this work and the ways these threats were mitigated.

External validity evaluates the generalizability of the approach and the extent to which the results of a study can be generalized to other systems. A potential threat in this category is that we analyzed our catalog against historical vulnerability reports from Linux Kernel, PHP, Chrome and Thunderbird. However, we carefully selected these case studies from different software domains. Therefore, we expect it to be representative of a typical software engineering environment, which suggests that it could generalize to a broader set of systems. Another threat to the external validity is that we focused our discussion on 39 security patterns. We tried to mitigate this threat through selecting security patterns that satisfy different security concerns (such as privacy, authentication, non-repudiation, and so on). Consequently, we believe that the results would not be overly different from other security patterns.

Construct validity evaluates the degree to which the claims were correctly measured. With this respect, one threat was the manual analysis of CVE instances in order to observe how often and the nature of security design issues in real software systems. To mitigate this threat we selected case studies with a higher number of reported vulnerabilities in order to minimize the potential impacts of an incorrect analysis in our datasets. Furthermore, only a small portion of the reported vulnerabilities does not have a CWE tag, so we only a smaller subset of vulnerabilities needed to be manually inspected to define a CWE tag.

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 construction of the CAWE catalog. To mitigate potential biases and incorrect classification of weaknesses, a peer review process was conducted when establishing the CAWE collection. In this review, a second graduate student also analyzed the entries from the MITRE's collection, mapped the appropriate ones to security patterns and shared the rationale of such mapping, as previously explained in Chapter 3. Hence, we consider that such peer evaluation minimized the impacts of biases and mistakes by the manual creation of the CAWE catalog.

Chapter 8

Conclusion and Future Work

This thesis presented the new concept of CAWE (Common Architectural Weakness Enumeration), to stimulate security design thinking into developers daily coding activities through a catalog that documents potential weaknesses related to security patterns. This work aimed to fill in the gap of architectural knowledge of architecture-related security issues.

In addition to creating the CAWE collection, this work reported the analysis of four large case studies and observed that at least 35% of the investigated vulnerabilities are rooted in the architecture of those case studies. Besides that, improper validation of inputs and access control were the two most design flaws observed in real systems.

8.1 Future Work

The current state of the art of vulnerability detection tools is still focused on finding low-level issues (bugs). This way, this catalog could be used as a knowledge base for automated techniques to detect architectural-level vulnerabilities from source code. One potential way to do this is through, firstly, detecting security patterns, then using the CAWE catalog to reason about potential associated weaknesses. This is one research direction of this work.

Moreover, in the future, we could evaluate our catalog with security experts, looking forward to expand the catalog. Also, we can conduct experiments to verify how effective is the catalog in aiding developers/designers to perform the activities explained in Chapter 5 (*e.g.*code inspections).

Bibliography

[1] Ivan Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, and Jacob West. Avoiding the top 10 software security design flaws. `https://www.computer.org/cms/CYBSI/docs/Top-10-Flaws.pdf`, 2014. (Accessed on 05/08/2016).

[2] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security & Privacy*, (1):84–87, 2005.

[3] Felix Bachmann, Len Bass, and Mark Klein. *Deriving Architectural Tactics: A Step Toward Methodical Architectural Design*. Technical Report, Software Engineering Institute, 2003.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (3rd Edition)*. Addison-Wesley Professional, 3 edition, October 2012.

[5] Carlos Blanco, Joaquin Lasheras, Rafael Valencia-García, Eduardo Fernández-Medina, Ambrosio Toval, and Mario Piattini. A systematic review and comparison of security ontologies. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 813–820. Ieee, 2008.

[6] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, (6):76–79, 2004.

[7] The MITRE Corporation. Cve - common vulnerabilities and exposures (cve). `http://cve.mitre.org/`. (Accessed on 04/29/2016).

[8] The MITRE Corporation. Cwe-256: Plaintext storage of a password. `http://cwe.mitre.org/data/definitions/256.html`. (Accessed on 04/29/2016).

[9] The MITRE Corporation. Cwe-354: Improper validation of integrity check value (2.9). `https://cwe.mitre.org/data/definitions/354.html`. (Accessed on 05/09/2016).

[10] The MITRE Corporation. Cwe-603: Use of client-side authentication (2.9). `https://cwe.mitre.org/data/definitions/603.html`. (Accessed on 05/09/2016).

[11] The MITRE Corporation. Capec - common attack pattern enumeration and classification. `http://capec.mitre.org/`, 2015. (Accessed on 05/09/2016).

[12] The MITRE Corporation. Cwe-159: Failure to sanitize special element, 2015. (Accessed on 04/29/2016).

[13] The MITRE Corporation. Cwe-261: Weak cryptography for passwords. `http://cwe.mitre.org/data/definitions/261.html`, 2015. (Accessed on 04/29/2016).

[14] The MITRE Corporation. Cwe-306: Missing authentication for critical function. `http://cwe.mitre.org/data/definitions/306.html`, 2015. (Accessed on 04/29/2016).

[15] The MITRE Corporation. Cwe-319: Cleartext transmission of sensitive information. `https://cwe.mitre.org/data/definitions/319.html`, 2015. (Accessed on 05/09/2016).

[16] Danny Dhillon. Developer-driven threat modeling: Lessons learned in the trenches. *IEEE Security & Privacy*, (4):41–47, 2011.

[17] Andreas Ekelhart, Stefan Fenz, Markus Klemen, and Edgar Weippl. Security ontologies: Improving quantitative risk analysis. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 156a–156a. IEEE, 2007.

[18] Michael E. Fagan. *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*, chapter Design and Code Inspections to Reduce Errors in Program Development, pages 301–334. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[19] The PHP Group. Php: Safe mode - manual. `http://php.net/manual/en/features.safe-mode.php`. (Accessed on 04/27/2016).

[20] Munawar Hafiz, Paul Adamczyk, and Ralph E. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, 2007.

[21] Munawar Hafiz, Paul Adamczyk, and Ralph E. Johnson. Growing a pattern language (for security). In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 139–158, New York, NY, USA, 2012. ACM.

[22] Shawn Hernan, Scott Lambert, Tomasz Ostwald, and Adam Shostack. Threat modeling-uncover security design flaws using the stride approach. *MSDN Magazine-Louisville*, pages 68–75, 2006.

[23] Michael Howard and Steve Lipner. *The security development lifecycle*. O'Reilly Media, Incorporated, 2009.

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

[25] Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-hewitt. Security patterns repository, version 1.0, 2006.

[26] John McDermott and Chris Fox. Using abuse case models for security requirements analysis. In *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*, pages 55–64. IEEE, 1999.

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

[28] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security Privacy*, 4(6):85–89, Nov 2006.

[29] Mehdi Mirakhorli, Yonghee Shin, Jane Cleland-Huang, and Murat Cinar. A tactic centric approach for automating traceability of quality concerns. In *International Conference on Software Engineering, ICSE (1)*, 2012.

[30] OWASP. Owasp clasp project. `https://www.owasp.org/index.php/Category:OWASP_CLASP_Project`, 2016. (Accessed on 05/08/2016).

[31] Gilberto Pedraza-Garcia, Hernan Astudillo, and Dario Correal. A methodological approach to apply security tactics in software architecture design. In *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*, pages 1–8. IEEE, 2014.

[32] S. Rehman and K. Mustafa. Research on software design level security vulnerabilities. *SIGSOFT Softw. Eng. Notes*, 34(6):1–5, December 2009.

[33] Jungwoo Ryoo, Rick Kazman, and Priya Anand. Architectural analysis for security. *IEEE Security & Privacy*, (6):52–59, 2015.

[34] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[35] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[36] IBM Security. Ibm statistics on data breach epidemic. `http://www-935.ibm.com/services/us/en/it-services/security-services/data-breach`, 2014. (Accessed on 05/06/2015).

[37] Amina Souag, Camille Salinesi, Isabelle Wattiau, and Haris Mouratidis. Using security and domain ontologies for security requirements analysis. In *Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual*, pages 101–107. IEEE, 2013.

[38] Chritopher Steel and Ramesh Nagappan. *Core Security Patterns: Best Practices and Strategies for J2EE", Web Services, and Identity Management*. Pearson Education India, 2006.

[39] Frank Swiderski and Window Snyder. *Threat modeling*. Microsoft Press, 2004.

[40] Jilles van Gurp, Sjaak Brinkkemper, and Jan Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *J. Softw. Maint. Evol.*, 17:277–306, July 2005.

[41] Bart De Win, Riccardo Scandariato, Koen Buyens, Johan Grgoire, and Wouter Joosen. On the secure software development process: Clasp, sdl and touchpoints compared. *Information and Software Technology*, 51(7):1152 – 1171, 2009. Special Section: Software Engineering for Secure Systems.

[42] Yan Wu, Robin A Gandhi, and Harvey Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, pages 22–28. ACM, 2010.