An Empirical Study of Tactical Vulnerabilities

Joanna C. S. Santos[a], Katy Tarrit[a], Adriana Sejfia[a], Mehdi Mirakhorli[a], Matthias Galster[b]

[a]*Software Engineering Department, Rochester Institute of Technology. USA.*
[b]*Department of Computer Science and Software Engineering, University of Canterbury. New Zealand.*

Abstract

Architectural security tactics (e.g., authorization, authentication) are used to achieve stakeholders' security requirements. Security tactics allow the system to react, resist, detect and recover from attacks. Flaws in the adoption of these tactics into the system's architecture, an incorrect implementation of security tactics, or deterioration of tactic implementations over time can introduce severe vulnerabilities that are exploitable by attackers. Therefore, in this work, we present the *Common Architectural Weakness Enumeration* (CAWE), a catalog of known weaknesses rooted in the design or implementation of security tactics which can result in tactical vulnerabilities. We categorized all known software weaknesses as tactic-related and non-tactic related. This way, our CAWE catalog enumerates common weaknesses in a security architecture that can lead to tactical vulnerabilities. From our CAWE catalog, we found 223 different types of tactical vulnerabilities. In this work, we also used this catalog to study tactical vulnerabilities in three large-scale open source projects: Chromium, PHP, and Thunderbird. In a detailed analysis, we identified the most occurring vulnerability types on these projects. From this study we observed that (i) Improper Input Validation and Improper Access Control were the most occurring vulnerability types in Chromium, PHP and Thunderbird and (ii) "Validate Inputs" and "Authorize Actors" were the security tactics mostly affected by these tactical vulnerabilities. Moreover, in a qualitative analysis of 632 tactical vulnerabilities and their fixes in these systems, we characterized their root causes and investigated the way the original developers of each system fixed these vulnerabilities. From this qualitative analysis, we found 44 distinct root causes that lead to these tactical vulnerabilities. The results of this study not only show how architectural weaknesses in systems have created severe vulnerabilities, but also provide recommendations driven by empirical data for addressing such security problems.

Keywords: software security architecture, security tactics, tactical vulnerabilities, architectural weaknesses

1. Introduction

Software engineers face an increasing pressure to deliver software applications that are *secure by design* [1], i.e., software applications that are designed from the ground up in a way that prevents or at least minimizes the impacts of vulnerabilities (here, a vulnerability is a possibility of a system being attacked or harmed). To achieve this, software architects work with stakeholders to identify security concerns and adopt appropriate architectural solutions to address them, forming the software's *security architecture* [2, 3]. These architectural solutions are often based on *security tactics* [4], which are reusable design solutions for achieving security quality attributes. Bass et al. [3] provide a comprehensive list of such tactics and classify them into tactics for *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").

Security tactics are the building blocks of a security architecture. A flaw in the *adoption* of these tactics into the architecture of a system, incorrect *implementation* of these tactics in the source code [5], or their *deterioration* during maintenance activities [6] can lead to severe vulnerabilities. In this paper, we define and refer to these vulnerabilities as **tactical vulnerabilities**. An example of a tactical vulnerability is the *Use of Client-Side Authentication*. In this example, the "Authenticate Actors" tactic [3] is adopted at the client side, therefore, the authenticity verification is performed by the code on the client rather than by the code on the server. This will enable attackers to reverse engineer the client code and develop a modified client that omits the authentication check, bypassing the authentication mechanism. While this example shows a weakness that can occur during the software *design* process, in a previous work [7] we found that even when the architecture is appropriately designed to satisfy its quality requirements, developers may *implement* the architectural tactics incorrectly, compromising architectural quality.

As an example, consider that architects mitigate the issue in the "Authenticate Actors" tactic by changing the security architecture to place the authentication check on the server side. Even though the system is now more resilient against attacks, developers still may fail to correctly implement the tactic by relying on cookies to implement the authentication logic. Listing 1 shows such incorrect implementation. In this code snippet, a PHP Web application is storing a value equal to "1" in an HTTP cookie (line 5) whenever a new user successfully authenticates. This cookie is later utilized to check whether the user has already logged in (line 2) and to

Email addresses: jds5109@rit.edu (Joanna C. S. Santos), ktics@rit.edu (Katy Tarrit), axs1461@rit.edu (Adriana Sejfia), mehdi@se.rit.edu (Mehdi Mirakhorli), mgalster@ieee.org (Matthias Galster)

grant access to the system (line 11). In this case, developers of this application assumed the immutability of HTTP cookies when, in reality, attackers can change the "`authenticate`" cookie to "`1`" and send an HTTP request to the application with this modified cookie. This would result in an *authentication-bypass*.

Listing 1 An example of an incorrect implementation of the tactic "Authenticate Actors" in a Web application written in PHP resulting in an authentication-bypass.

```php
1   $auth = $_COOKIES['authenticated'];
2   if (!$auth) {
3       if (authenticate($_POST['username'], $_POST['password'])) {
4           // save the cookie to be sent out in future responses
5           setcookie('authenticated', '1', time()+60*60*2);
6       } else {
7           showLoginScreen(); // request user to login
8           die('\n'); // kill the process
9       }
10  }
11  performPrivilegedAction();
```

Despite the importance of the software architecture in achieving security [8], recent empirical studies of software vulnerabilities have not fully explored the architectural context, including design decisions such as tactics and patterns [9, 10, 11]. They typically focus on studying and understanding security issues related to the management of data structures and variables (e.g., buffer overflow/over-read). Others have developed architecture analysis techniques to correlate design violations with software vulnerabilities [1]. While such studies have investigated software vulnerabilities from structural perspectives, we currently lack an in-depth understanding of the nature and root causes of *tactical vulnerabilities*, which would help teach software developers and architects to avoid and mitigate these problems in their systems.

A recent effort towards shifting the focus from mitigating coding mistakes to finding and promoting the awareness of common weaknesses in a security architecture was made by the *IEEE Center for Secure Design* [12]. This center released a list of the top 10 most common architectural weaknesses. However, only a few examples of such security architecture weaknesses have so far been obtained or published to help architects and developers to learn and avoid such security issues.

Therefore, in this work, we first present the *Common Architectural Weakness Enumeration* (CAWE), a catalog of known weaknesses rooted in the design or implementation of security tactics which can result in tactical vulnerabilities. The CAWE catalog was built from an existing catalog of known types of software vulnerabilities[1]. Since this existing catalog did not distinguish between pure coding issues and weaknesses in security tactics, we categorized all known software weaknesses as tactic-related and non-tactic related. This way, our CAWE catalog enumerates common weaknesses in a security architecture that can lead to tactical vulnerabilities. In this work, we also use this catalog to study tactical vulnerabilities in three large-scale open source projects. The results of this study not only show how tactical weaknesses in systems have created severe vulnerabilities, but also demonstrate the importance of architecture-based approaches to avoid software vulnerabilities, and how the CAWE catalog can facilitate this process.

1.1. Research Questions and Outcomes of this Study

In this paper, we investigate the following research questions.

RQ1: *What types of tactical vulnerabilities exist?*

Our goal in answering this question is to identify weaknesses in a security architecture that are the result of a flawed design and/or implementation of security tactics (i.e. types of tactical vulnerabilities). We found 223 different known types of tactical vulnerabilities, summarized in the CAWE catalog.

RQ2: *Which security tactics are more likely to have associated vulnerabilities?*

For each security tactic, we investigated the *potential* types of vulnerabilities (i.e., weaknesses) that are rooted in their design and/or implementation. This was to verify which tactics are at a higher risk of being improperly adopted. We observed that the "Authorize Actors", "Validate Inputs" and "Encrypt Data" tactics are subject to a higher number of weaknesses if not correctly adopted. Therefore, these security tactics need to be implemented and tested more carefully.

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

We also used the CAWE catalog to conduct an in-depth case study of tactical vulnerabilities across three large-scale open-source systems: Chromium, PHP, and Thunderbird. In this study, we retrieved and reviewed software artifacts of each system, such as their source code, version control data, and their disclosed vulnerabilities in the National Vulnerability Database (NVD)[2]. We also identified security tactics adopted in these systems and traced them to the source code. After analyzing these artifacts, we mapped their vulnerabilities to security tactics to identify "tactical" and "non-tactical" vulnerabilities. This led to answering the following research questions about understanding tactical vulnerabilities in real software systems:

RQ3: *What are the most common tactical vulnerability types in Chromium, PHP, and Thunderbird?*

Using the data we collected, we scrutinized the types of tactical vulnerabilities across the three systems, and we found that "Improper Input Validation" is by far the most common vulnerability type.

RQ4: *What security tactics are most affected by tactical vulnerabilities in Chromium, PHP, and Thunderbird?*

While the answer to **RQ2** indicates the security tactics that are more *likely* to be incorrectly designed/implemented (i.e. that have the highest amount of associated types of tactical vulnerabilities in the CAWE catalog), in this question, we propose to observe *to what extent such trend occurs* in the three case studies. Thus, we studied which security tactics were most affected by tactical vulnerabilities in these projects. We found that "Validate Inputs", "Authorize Actors" and "Limit Exposure" were the security tactics most impacted by vulnerabilities in Chromium, PHP and Thunderbird.

RQ5: *What are the root causes of the most frequently occurring types of tactical vulnerabilities in Chromium, PHP and Thunderbird?*

The tactical vulnerability types found in answering **RQ3** indicate (at a high-level of abstraction) classes of vulnerabilities that affected security tactics. Although these tactical vulnerability types provide *clues* about the nature of the problem, they are not concrete enough for developers and architects to act upon. In this respect, for **RQ5**, we conducted a qualitative analysis of tactical vulnerabilities in the case studies and investigated the underlying *root causes* (i.e., the specific violations of tactics) of the most reoccurring types of tactical vulnerabilities that we found in answering **RQ3**. The goal of this question is to use empirical data to demonstrate the root causes of the tactical issues in the case studies along with their implications and potential fixes. All the findings of this part of this research are grounded in empirical data collected from case studies.

1.2. Originality and Extension

This work extends our previous publications [13, 14] in different ways. In our previous works, we established the CAWE catalog [13], and studied tactical vulnerabilities in Chromium, PHP and Thunderbird to investigate their types, complexity to fix and frequency of occurrence over time [14]. In this work, we extend the previous publications by conducting a detailed qualitative analysis of the tactical vulnerabilities across Chromium, PHP and Thunderbird to identify their *root causes* (Section 5). Furthermore, based on the study of vulnerability fixes by the original developers of these systems, we create actionable recommendations for software architects and developers for mitigating and preventing tactical vulnerabilities. This qualitative study was conducted over a period of 6 months. The results are empirically grounded and are driven by an in-depth and manual analysis of 632 tactical vulnerabilities and their fixes.

Thus, the contributions of this paper are:

- A description of the catalog of common types of tactic-related vulnerabilities (CAWE). The proposed CAWE catalog documents the known type of vulnerabilities for each security tactic;
- An in-depth analysis of the relationship between software vulnerabilities and architectural security tactics. This allows us to understand the architectural context of vulnerabilities instead of solely focusing on coding issues related to the management of data structures and variables (e.g., buffer overflow/overread). Furthermore, it makes it possible to get insights about 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;
- A detailed discussion of the root causes for tactical vulnerabilities. The benefit of fine-grained root causes is twofold (i) it gives insights to developers and architects about how they can identify and mitigate these problems; (ii) it can help during software testing, as the expected behavior and misbehavior are clearly specified.

[2]`https://nvd.nist.gov/`

1.3. *Organization of the Paper*

Section 2 briefly introduces vulnerability-related concepts and terms to ensure that the essence of the paper can be understood by a broader audience, along with related work. Section 3 describes our CAWE catalog in details. Section 4 discusses how we used the CAWE catalog to study tactical vulnerabilities in Chromium, PHP, and Thunderbird. Section 5 presents the qualitative analysis of tactical vulnerability reports in order to identify their root causes. Section 6 elaborates on threats to the validity of this work, and Section 7 concludes this paper.

2. Background and Related Work

This section discusses the fundamental concepts and terminology used in our work. We first discuss software vulnerabilities data and vulnerability databases (Section 2.1) and then we explain security tactics and tactical vulnerabilities in more detail (Section 2.2). Finally, we discuss related work (Section 2.3).

2.1. *Software Vulnerabilities*

Vulnerabilities in a software system are caused by defects that affect its intended security properties, and are typically tracked in vulnerability databases. A well-known example is the **National Vulnerability Database (NVD)** which currently contains over 91,000 vulnerabilities that exist in a variety of software products. Each vulnerability recorded in the NVD is assigned a unique **CVE ID** (Common Vulnerabilities and Exposure Identifier) and contains the details about the security problem. An example of a vulnerability record in the NVD is shown below:

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)
[...]

As this excerpt shows, the NVD provides a short **description** of the problem and **references** for the vulnerability, i.e. a list of links to other Web sites (such as issue tracking systems) that may contain more details about the security issue. It also specifies which **software releases** were affected by the vulnerability (in this case, it was version 5.3.7 of PHP). Some of the CVE instances may also provide a **CWE tag** that indicates the **vulnerability type**. This tag refers to an entry from the **Common Weakness Enumeration (CWE)** dictionary, which enumerates common weaknesses in a software system that may lead to vulnerabilities. The vulnerability type denotes a family of security defects that share one or more aspect in common, such as a similar fault (*root cause*), failure (*consequence*), or fix (*repair*) [15]. Thus, the CWE tag is used by the NVD as a way to classify vulnerabilities. It is important to highlight that a **weakness** (or **vulnerability type**) is a class of problems in a software system that may introduce a security defect, whereas a **vulnerability** is an *instance* of a weakness (an actual occurrence of the weakness).

2.2. *Security Tactics and Tactical Vulnerabilities*

Architectural Security Tactics are means of achieving security properties through a series of inter-related design decisions [16]. They are the building blocks of a security architecture and provide reusable solutions for satisfying security requirements, even when the system is under attack [3]. A comprehensive list of security tactics has been provided by Bass et al. [3] classified into the four categories presented in Table 1.

Table 1: Security tactics and their definitions

Category	Tactic	Description
Resist Attacks	Identify Actors	Identifies the external agents that provide inputs into the systems
	Validate Inputs	Sanitizes, neutralizes and validates any externally provided inputs to minimize malformed data from entering the system and preventing code injection in the input data
	Manage User Sessions	Retains the information or status about each user and his/her access rights for the duration of multiple requests
	Authenticate Actors	Verifies the authenticity of actors (i.e. to check if the actor is indeed who it claims to be).
	Authorize Actors	Enforces that agents have the required permissions before performing certain operations, such as modifying data
	Limit Access	Limits the amount of resources that are accessed by actors, such as memory, network connections, CPU, etc.
	Limit Exposure	Minimizes the attack surface through designing the system with the least needed amount of entry points
	Encrypt Data	Maintains data confidentiality through use of encryption libraries
	Separate Entities	Places processes, resources or data entities in separate boundaries to minimize the impacts attacks
	Change Default Settings	Forces users to configure the system before use by changing the default (and potentially less secure) configuration.
React to Attacks	Revoke Access	In case of attacks, the system denies access to resources to everyone until the malicious behavior ends
	Lock Computer	Lockout mechanism that takes effect in case of multiple failed attempts to access a given resource
	Inform Actors	In case of malicious activities, the users/administrators or other entities that are in charge of the system are notified.
Detect Attacks	Detect Intrusion	Monitors network traffic for detecting abnormal traffic patterns caused by intrusion attempts
	Detect Service Denial	Monitors incoming traffic for detecting Denial Of Services (DoS) attacks.
	Verify Message Integrity	Ensures integrity of data, such as messages, resource files, deployment files, and configuration files
	Detect Message Delay	Detects malicious behavior through observing the time spent on delivering messages. In case messages are taking unexpected times to be received, the system may detect a potential data leakage.
Recover from Attacks	Audit	Logs user activities in order to identify attackers and modifications to the system

While these security tactics provide a well-formed solution to address various security concerns, if they are not designed and implemented carefully, they can result in weaknesses in the security architecture [12]. We can classify these weaknesses into **omission**, **commission** and **realization** weaknesses [13]:

- **Omission weaknesses** are caused by *missing* a security tactic when it is needed to satisfy a security requirement. An example of an omission weakness is to exchange keys[3] *without authentication*. In this example, the software architect missed the need for authenticating entities before performing a key exchange to ensure that the sensitive information is transferred to a trustworthy actor. The lack of the "Authenticate Actors" tactic in this scenario allows attackers to perform man-in-the-middle attacks, which can compromise the system's confidentiality.
- **Commission weaknesses** refer to an incorrect choice of tactics which could result in undesirable consequences. An example of this weakness is to *rely on IP addresses for authentication*, in which there is a list of trusted IP addresses that are used to verify the authenticity of messages. While architects have made a design decision to satisfy the requirement of authentication of entities, the weakness in this design will enable attackers to bypass the authentication by forging a trusted IP address.
- **Realization weaknesses** occur when appropriate security tactics are adopted but are incorrectly implemented. For example, a developer does not invalidate prior existing sessions before creating a new session while implementing the "Manage User Sessions" tactic, resulting in a session fixation vulnerability. This enables an intruder to steal user sessions.

Based on the above classification of weaknesses, we define **tactical vulnerabilities** as: *software vulnerabilities introduced in a system because of design and implementation issues related to architectural tactics. More specifically, these vulnerabilities occur due to (i) a lack of security tactics (omission) in the application's architecture; or (ii) adoption of less suitable security tactics for a given design problem or context (commission); or (iii) an incorrect implementation of security tactic principles which results in an incorrect transition from design to code (realization weakness).*

These weaknesses in a security architecture may lead to vulnerabilities that can be successfully exploited by attackers. In this paper, we refer to these vulnerabilities as **tactical vulnerabilities**, as they are rooted in the design and/or implementation of security tactics.

2.3. Related Work

There are many books and publications towards the identification and categorization of security tactics [3, 18, 19]. In our work and the CAWE catalog however, we focused on documenting how these tactics could be compromised when incorrectly adopted. This helps spreading awareness for security problems rooted in the design/implementation of tactics.

The use of security knowledge bases as a resource to help developers and engineers in their daily activities has been previously discussed in the research community. Security ontologies, which represent knowledge within the security domain, have been created to support some activities, such as requirements engineering and quantitative risk analysis, but they did not introduce architectural concepts [20]. Similar to a security ontology, Wu et al. [21] proposed *semantic templates*, which are a structured description of generic patterns of relationships between software components, faults and security consequences built on top of the CWE list and the vulnerabilities reported in the NVD. However, these templates do not differentiate architectural concerns.

A similar effort towards understanding security problems from an architectural perspective was the *IEEE Center For Secure Design*, which recently released a list of the top 10 design flaws [12], based on experiences in industry, academia, and government. However, to this day the descriptions for each flaw are generic, there are not many details for mitigating these flaws, and they come from experience rather than empirical evidence. Thus, in this work, we extensively collected a list of software weaknesses to identify the ones rooted in a security architecture and investigated their occurrences in existing systems.

Existing research in software architecture for security has mainly proposed techniques for facilitating the design of security architecture [22], the analysis and evaluation of the existing security architecture [23, 24] as well as identifying potential threats/vulnerabilities from the architecture [25, 26, 27] . While these works can aid architects in identifying 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 time.

To help avoid deterioration of security architecture during software maintenance, Taspolatoglu and Heinrich [28] 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 over time and result in vulnerabilities, unlike our work, it did not provide evidence on how frequently such problems occur and how complex they are to fix.

Ryoo et al. [29] evaluated to what 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

[3]These keys are used to encrypt messages exchanged between two entities so that a secure communication can be established over an insecure channel [17].

as "Encrypt Data"). While in our work we also analyzed the usage of security tactics in three software systems, our main goal was to investigate how vulnerabilities are caused by *incorrect adoption* of these tactics in the code.

Feng et al [1] investigated the relationship between design rule violation and vulnerabilities. They observed that source files that contain a higher number of design rule violations are highly correlated to the presence of vulnerabilities, as well as high levels of code churn when fixing such vulnerabilities. However, unlike our work, they investigated the files that contain modularity violations 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 what was the efforts to fix them.

In summary, despite the efforts from the research community 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 of what the common root causes of such incorrect implementations are and the corresponding efforts to fix them.

3. A Catalog of Tactical Vulnerability Types

We created the CAWE catalog through a systematic categorization of the CWE list [4], an existing dictionary of common types of vulnerabilities. The CWE list contains about 1,000 entries, but it does not clearly distinguish tactical vulnerability types (i.e., security issues rooted in the design and/or implementation of security tactics) from purely programming issues (such as buffer overflows or null-pointer dereferences). Thus, we systematically classified the entries in the CWE dictionary into *coding bugs* (i.e., not related to security tactics) and *tactic-related weaknesses*. We also identified how these *tactic-related weaknesses* affect well-known security tactics. As a result, the CAWE catalog is a view of the CWE dictionary, enumerating the subset of weaknesses from the CWE list that corresponds to a weakness in a security architecture.

3.1. Creating the CAWE catalog

To establish the CAWE catalog we performed the following steps:

1. We compiled an extensive list of security tactics published in the literature [3, 30]. For each security tactic, we extracted its *description* and *keywords* that summarize the security tactic. This first step resulted in a list of 18 security tactics (see Table 1).

2. We retrieved all entries from the CWE dictionary. An entry in the CWE dictionary can be of four types: a *View* groups weaknesses from a given perspective (e.g., types of errors); a *Category* aggregates entries based on a common attribute (e.g., shared environment (J2EE, .NET), functional area (authentication, cryptography), relevant resources); a *Weakness* corresponds to an actual type of security problem; a *Compound Element* describes security problems due to the occurrence of other weaknesses in a time sequence. Since *View* and *Category* entries group other weaknesses rather than representing software weaknesses, they are not included in our analysis. This way, out of the 1,004 entries in version 2.9 of the CWE dictionary, we retrieved the subset of 727 entries of type *Weakness* or *Compound Elements*. For each *Weakness* and *Compound Element* type, the CWE dictionary provides information such as a *description*, *mitigation techniques*, *common consequences*, *code examples*, etc [5].

3. We searched the 727 CWE entries for the keywords related to the 18 security tactics identified in the first step. This search resulted in a list of *potential* connections between security tactics and CWE entries.

4. We manually analyzed all provided data for all 727 entries (i.e., their descriptions, mitigation techniques, consequences, attack patterns and time of introduction) to confirm whether these potential connections indeed existed and verified whether there were not any missing connections. During this manual analysis, we decomposed each CWE into three dimensions: its *root cause* (identified based on the entry's description and time of introduction), its *failure* (observed from the entry's enumerated consequences), its *fix* (identified from the described mitigation techniques). As defined in Section 2, the criteria to consider a CWE entry to be a *tactic-related weakness* is that the weakness is either caused by (i) a lack of a design decision (*omission*); or (ii) an incorrect choice of security tactics which results in "bypasses", i.e., an attacker being able to bypass the security mechanism and breach into the system (*commission*) or (iii) an incorrect transition from tactic design to implementation in the code (*realization weakness*). If a CWE entry matched any of these conditions, it was considered to be rooted in the design and/or implementation of a security tactic and classified as a *tactic-related weakness*. We annotated each of these *tactic-related weaknesses* with (i) the security tactic affected by the weakness and (ii) the type of impact (commission, omission or realization weakness).

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

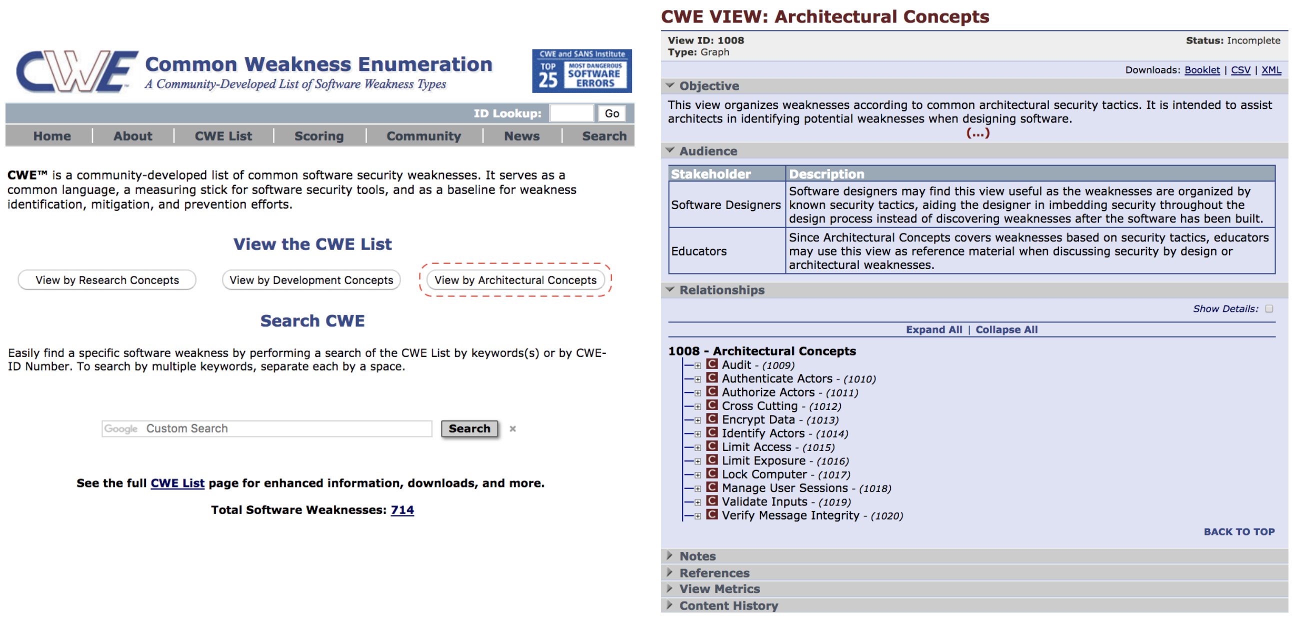
[5]The complete information provided in the CWE dictionary is documented on MITRE's Website: `https://cwe.mitre.org/data/xsd/cwe_schema_v5.4.2.xsd`

To illustrate this systematic process, consider CWE-354 ("Improper Validation of Integrity Check Value"). It contains some of the keywords related to the security tactic "Verify Message Integrity". Thus, after performing the third step, (the keyword-based search), this CWE was considered to be *potentially* related to the "Verify Message Integrity" tactic because it contained keywords related to the tactic. When we subsequently manually inspected this CWE instance, we found that this type of problem is caused by an incorrect verification of the checksums [6] of messages. This leads to the software system to potentially accept corrupted or intentionally modified messages. From this inspection, we considered this CWE entry to be a "realization weakness" affecting the "Verify Message Integrity" tactic because it occurs due to an incorrect implementation of the tactic (as described in the mitigation section, it implies that the system handles a message protocol that supports message integrity verification, but the application failed to correctly implement such mechanism).

Since the keyword-based search may not show *all* the potential connections between CWE instances and tactics, it is important to highlight that we also carefully inspected all entries which were not identified through the keyword-based search. In particular, if a CWE was tagged with "Architecture and Design" as the time of when this weakness is introduced in a system, we inspected if the CWE discussed that the issue occurred because of a lack of a security tactic. For instance, the CWE-306 ("Missing Authentication for Critical Function") is caused by the absence of adopting the "Authenticate Actors" tactic (i.e., an "omission weakness").

To minimize inherent biases in this manual analysis, four individuals worked independently over all these 727 entries to categorize them. Once they had completed their analysis, results were double-checked. For the entries with disagreements (84 in total), they discussed their rationale and reached a consensus of what would be the appropriate classification.

3.2. Overview of the CAWE Catalog


Common Weakness Enumeration
A Community-Developed List of Software Weakness Types
CWE and SANS Institute TOP 25 MOST DANGEROUS SOFTWARE ERRORS
ID Lookup: Go
Home | About | CWE List | Scoring | Community | News | Search

CWE™ is a community-developed list of common software security weaknesses. It serves as a common language, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

View the CWE List

View by Research Concepts
View by Development Concepts
View by Architectural Concepts

Search CWE

Easily find a specific software weakness by performing a search of the CWE List by keywords(s) or by CWE-ID Number. To search by multiple keywords, separate each by a space.

See the full CWE List page for enhanced information, downloads, and more.

Total Software Weaknesses: 714

CWE VIEW: Architectural Concepts

View ID: 1008 **Status:** Incomplete
Type: Graph

Downloads: Booklet | CSV | XML

Objective

This view organizes weaknesses according to common architectural security tactics. It is intended to assist architects in identifying potential weaknesses when designing software.
(...)

Audience

Stakeholder	Description
Software Designers	Software designers may find this view useful as the weaknesses are organized by known security tactics, aiding the designer in imbedding security throughout the design process instead of discovering weaknesses after the software has been built.
Educators	Since Architectural Concepts covers weaknesses based on security tactics, educators may use this view as reference material when discussing security by design or architectural weaknesses.

Relationships

1008 - Architectural Concepts
- Audit - *(1009)*
- Authenticate Actors - *(1010)*
- Authorize Actors - *(1011)*
- Cross Cutting - *(1012)*
- Encrypt Data - *(1013)*
- Identify Actors - *(1014)*
- Limit Access - *(1015)*
- Limit Exposure - *(1016)*
- Lock Computer - *(1017)*
- Manage User Sessions - *(1018)*
- Validate Inputs - *(1019)*
- Verify Message Integrity - *(1020)*

(a) (b)

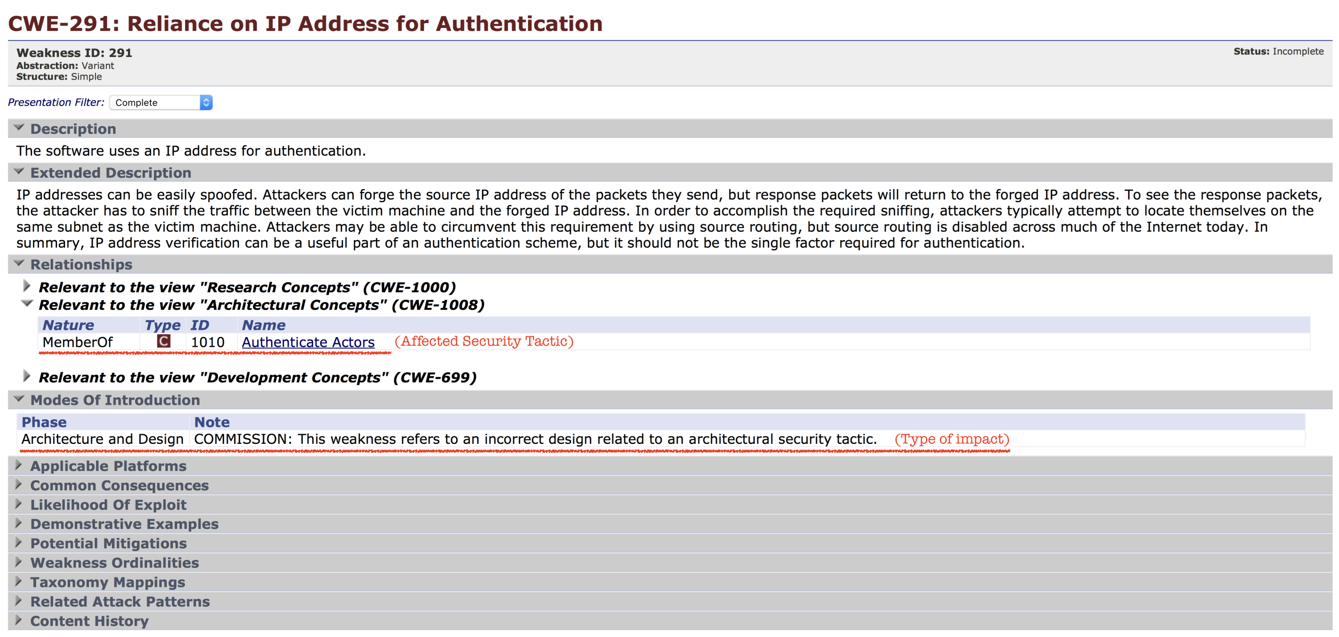
Figure 1: (a) Home page of MITRE's list of software weaknesses (b) The CAWE catalog integrated into MITRE's Website as a View

As shown in Figure 1(a), the CAWE catalog is integrated intro MITRE's list of software weaknesses as a *View*. This view is named as "Architectural Concepts" and was assigned an ID equals to *1008*. The CAWE view is publicly accessible through the following link: `http://cwe.mitre.org/`.When users navigate directly to the CAWE View's URL or click on the "View by Architectural Concepts" button in Figure 1(a), it takes them to the page shown in Figure 1(b). This Web page shows the list of affected security tactics (collapsed). When these tactics are expanded it shows the associated tactical weaknesses.

Currently, our CAWE catalog has 223 tactic-related weaknesses categorized based on 11 security tactics. The CAWE catalog also has a category called "Cross-Cutting", which encompasses weaknesses that can impact multiple security tactics (see category #1012 in Figure 1(b)). An example of a tactic-related weakness is presented in Figure 2. This weakness leads to a bypass of the "Authenticate Actors" tactic caused by leveraging IP addresses to verify the authenticity of actors (a commission weakness).

It is important to highlight that although MITRE's Website had a view that encompasses "mistakes made during the design and/or architecture phase"[7] our definition and purposes for the CAWE view are slightly broader. The goal of the CAWE view is to promote the awareness of mistakes related to the security architecture itself (as an artifact). In other words, weaknesses are

[6]Checksums are extra data that is attached to messages to detect errors and modifications in the message.
[7]"CWE-701: Weaknesses Introduced During Design": http://cwe.mitre.org/data/definitions/701.html



CWE-291: Reliance on IP Address for Authentication

Weakness ID: 291
Abstraction: Variant
Structure: Simple

Status: Incomplete

Presentation Filter: Complete

Description

The software uses an IP address for authentication.

Extended Description

IP addresses can be easily spoofed. Attackers can forge the source IP address of the packets they send, but response packets will return to the forged IP address. To see the response packets, the attacker has to sniff the traffic between the victim machine and the forged IP address. In order to accomplish the required sniffing, attackers typically attempt to locate themselves on the same subnet as the victim machine. Attackers may be able to circumvent this requirement by using source routing, but source routing is disabled across much of the Internet today. In summary, IP address verification can be a useful part of an authentication scheme, but it should not be the single factor required for authentication.

Relationships

Relevant to the view "Research Concepts" (CWE-1000)

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	*Type*	*ID*	*Name*
MemberOf	C	1010	Authenticate Actors (Affected Security Tactic)

Relevant to the view "Development Concepts" (CWE-699)

Modes Of Introduction

Phase	Note
Architecture and Design	COMMISSION: This weakness refers to an incorrect design related to an architectural security tactic. (Type of impact)

Applicable Platforms

Common Consequences

Likelihood Of Exploit

Demonstrative Examples

Potential Mitigations

Weakness Ordinalities

Taxonomy Mappings

Related Attack Patterns

Content History

Figure 2: CWE-291 "Reliance on IP Address for Authentication" with the Added Metadata from our Work (the Impact Type and affected Tactic)

then either omission/commission (that occur during the design process) or realization (that occur during the transition of a correct architecture to source code).

3.3. Using the CAWE catalog to Answer RQ1 and RQ2

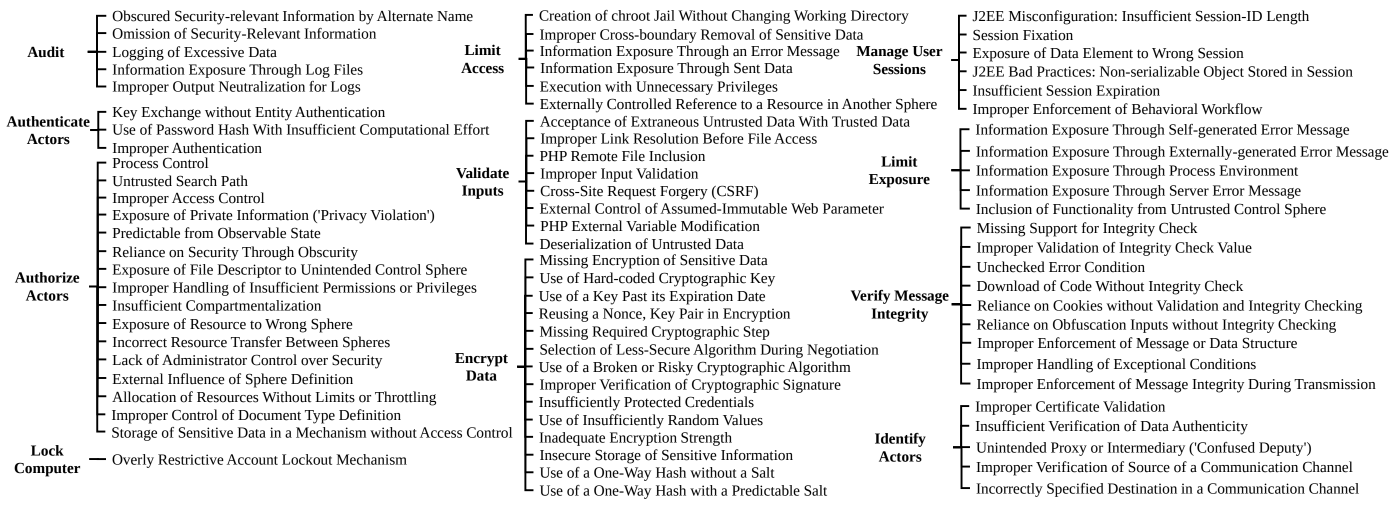

Audit
Obscured Security-relevant Information by Alternate Name
Omission of Security-Relevant Information
Logging of Excessive Data
Information Exposure Through Log Files
Improper Output Neutralization for Logs
Authenticate Actors
Key Exchange without Entity Authentication
Use of Password Hash With Insufficient Computational Effort
Improper Authentication
Authorize Actors
Process Control
Untrusted Search Path
Improper Access Control
Exposure of Private Information ('Privacy Violation')
Predictable from Observable State
Reliance on Security Through Obscurity
Exposure of File Descriptor to Unintended Control Sphere
Improper Handling of Insufficient Permissions or Privileges
Insufficient Compartmentalization
Exposure of Resource to Wrong Sphere
Incorrect Resource Transfer Between Spheres
Lack of Administrator Control over Security
External Influence of Sphere Definition
Allocation of Resources Without Limits or Throttling
Improper Control of Document Type Definition
Storage of Sensitive Data in a Mechanism without Access Control
Lock Computer
Overly Restrictive Account Lockout Mechanism
Limit Access
Creation of chroot Jail Without Changing Working Directory
Improper Cross-boundary Removal of Sensitive Data
Information Exposure Through an Error Message
Information Exposure Through Sent Data
Execution with Unnecessary Privileges
Externally Controlled Reference to a Resource in Another Sphere
Validate Inputs
Acceptance of Extraneous Untrusted Data With Trusted Data
Improper Link Resolution Before File Access
PHP Remote File Inclusion
Improper Input Validation
Cross-Site Request Forgery (CSRF)
External Control of Assumed-Immutable Web Parameter
PHP External Variable Modification
Deserialization of Untrusted Data
Encrypt Data
Missing Encryption of Sensitive Data
Use of Hard-coded Cryptographic Key
Use of a Key Past its Expiration Date
Reusing a Nonce, Key Pair in Encryption
Missing Required Cryptographic Step
Selection of Less-Secure Algorithm During Negotiation
Use of a Broken or Risky Cryptographic Algorithm
Improper Verification of Cryptographic Signature
Insufficiently Protected Credentials
Use of Insufficiently Random Values
Inadequate Encryption Strength
Insecure Storage of Sensitive Information
Use of a One-Way Hash without a Salt
Use of a One-Way Hash with a Predictable Salt
Manage User Sessions
J2EE Misconfiguration: Insufficient Session-ID Length
Session Fixation
Exposure of Data Element to Wrong Session
J2EE Bad Practices: Non-serializable Object Stored in Session
Insufficient Session Expiration
Improper Enforcement of Behavioral Workflow
Limit Exposure
Information Exposure Through Self-generated Error Message
Information Exposure Through Externally-generated Error Message
Information Exposure Through Process Environment
Information Exposure Through Server Error Message
Inclusion of Functionality from Untrusted Control Sphere
Verify Message Integrity
Missing Support for Integrity Check
Improper Validation of Integrity Check Value
Unchecked Error Condition
Download of Code Without Integrity Check
Reliance on Cookies without Validation and Integrity Checking
Reliance on Obfuscation Inputs without Integrity Checking
Improper Enforcement of Message or Data Structure
Improper Handling of Exceptional Conditions
Improper Enforcement of Message Integrity During Transmission
Identify Actors
Improper Certificate Validation
Insufficient Verification of Data Authenticity
Unintended Proxy or Intermediary ('Confused Deputy')
Improper Verification of Source of a Communication Channel
Incorrectly Specified Destination in a Communication Channel

Figure 3: High-Level Overview of the CAWE Catalog [13]

3.3.1. RQ1: Types of Tactical Vulnerabilities

From the CAWE catalog, we observed that among the 727 software weaknesses we inspected from the CWE dictionary, 223 are tactic-related weaknesses, i.e., corresponding to different types of vulnerabilities rooted in the design/implementation of security tactics. Figure 3 presents a high-level hierarchical view of these types of tactical vulnerabilities from the CAWE catalog per tactic. Note that some tactic-related weaknesses are children of other entries, but for simplicity reasons, this figure only shows the higher-level entries.

Key Finding for RQ1:

– There are **223** different types of tactical vulnerabilities.

3.3.2. RQ2: Security Tactics Likely to have Associated Vulnerabilities

To answer this question, we computed the total number of tactical weaknesses associated with each security tactic in the CAWE catalog. This allows us to understand which security tactics are more likely to be incorrectly adopted (since it has more ways to be flawed). Table 2 shows the number of tactical vulnerabilities types relevant to each security tactic along with a breakdown by the impact type (omission, commission and realization weaknesses). This table shows that the "Authorize Actors" tactic, which is used to ensure that only legitimate users can access data and/or resources, is subject to a higher number of known weaknesses if not implemented correctly (38 realization weaknesses). Therefore, it needs to be implemented and tested more carefully. Similarly, tactics "Validate Inputs" and "Encrypt Data" need to be implemented carefully to avoid incorrect assumptions during their design and/or implementation. We also found 9 tactical weaknesses that are cross-cutting, i.e., that affect multiple security tactics.

Table 2: Total Number of Vulnerabilities per Security Tactics

Security Tactic	# CAWEs	Realization	Omission	Commission
Audit	6	3	1	2
Authenticate Actors	29	12	2	15
Authorize Actors	60	38	16	6
Cross Cutting	9	3	3	3
Encrypt Data	38	18	13	7
Identify Actors	12	10	2	0
Limit Access	8	7	0	1
Limit Exposure	6	6	0	0
Lock Computer	1	0	0	1
Manage User Sessions	6	5	0	1
Validate Inputs	39	35	4	0
Verify Message Integrity	10	6	4	0

Key Finding for RQ2:

– Tactics "Authorize Actors", "Validate Inputs" and "Encrypt Data" are at a higher risk of being incorrectly adopted in a software system.

4. Empirical Investigation of Tactical Vulnerabilities in Real Software Systems

After establishing the CAWE catalog, which enumerates common types of tactical vulnerabilities, we investigated the occurrence of these weaknesses in real systems. We conducted an in-depth case study with three cases [31] based on guidelines for industrially-based multiple-case studies [32] (where each of the three systems is one case). The unit of analysis in our study was a *software project*. In each case (Chromium, PHP, and Thunderbird), we investigated **RQ3** and **RQ4** (what are the most common tactical vulnerability types on Chromium, PHP, and Thunderbird, and what security tactics are mostly affected by vulnerabilities in Chromium, PHP, and Thunderbird).

4.1. Case Selection

The criteria we used for selecting cases for our study were that the systems should be (i) widely adopted by a large number of users, (ii) among the top 50 software projects with the highest number of vulnerabilities [33], (iii) implementing a wide range of security tactics, (iv) using an issue tracking system for managing and fixing defects, and (v) from different software domains. Through these criteria, we ensured that the selected projects provided a rich set of artifacts regarding the software development activities conducted (to have access to all necessary data for our study), security tactics used, reported vulnerabilities, and fixes to vulnerabilities. Based on these criteria, we selected **Chromium** [8] (a Web browser), **Mozilla Thunderbird** [9] (an email and news feed client) and **PHP** [10] (the interpreter of the PHP programming language) as case studies. These projects are diverse in size, age, and domain, but similar with respect to their underlying programming language (they were mostly written in C/C++), as shown in Table 3.

4.2. Data Collection and Analysis

We performed the following steps: (i) identification of the security tactics adopted in each project (Section 4.2.1); (ii) retrieval of each project's disclosed vulnerabilities in the NVD (Section 4.2.2); (iii) classification of vulnerabilities as tactical and non-tactical (Section 4.2.3). To help the reader understand our analysis process, we show the collected artifacts and their relationships in Figure 4.

[8]http://www.chromium.org/
[9]http://mozilla.org/thunderbird/
[10]http://php.net/

Table 3: Details about the studied systems (statistics collected as of January 2017).

	Chromium	**PHP**	**Thunderbird**
Size (LOC)	>14 MLOC	>4 MLOC	>1 MLOC
# 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	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	1,380	531	705
Number of users	~1 billion	~244 millions	~9 millions
Rank	4th	23rd	15th

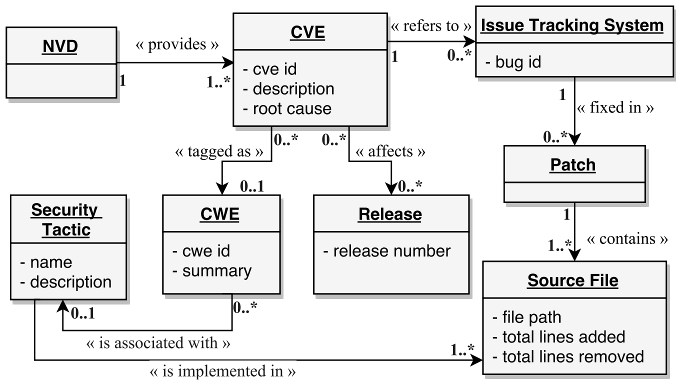


Figure 4: Data Extraction Information Model

4.2.1. *Identifying Security Tactics in each Project*

The first step involved identifying the security tactics used in the three projects. To ensure the accuracy of the identifications, we performed the following complementary activities:

- We reviewed the available literature and technical documentation for each project [34] to look for any references to specific security tactics and manually checked if these tactics occurred in the code.
- We manually browsed through the source files in each project to identify tactic-related files
- We searched tactic-related keywords (e.g. "authenticate") on the source code of the projects.
- We used a previously developed technique that automatically reverse-engineers architectural tactics from source code [35, 30].

The results of these four activities were merged to document the set of tactics used in each project. We then obtained feedback from developers involved in these projects 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 implementation of the security tactics. The list of identified security tactics for each project is shown in Table 4.

Table 4: 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 Setings	Inform Actors	Detect Denial of Service Attack	Detect Intrusion	Verify Message Integrity	Audit
Chromium	*	*	*	*	*	*	*		*	*	*	*	*	*	*
PHP		*	*	*	*	*		*						*	*
Thunderbird	*	*	*	*	*			*			*		*	*	*

4.2.2. Extracting Disclosed Vulnerabilities for each Project

We retrieved all CVEs for the three systems from the NVD. As shown in Figure 4, these CVEs are the starting point to collect the required artifacts. Thus, to ensure the accuracy and completeness of our data, we perform the following steps:

- **Completeness check:** Even though the NVD can provide a variety of information for each vulnerability, not all CVE instances provide the data we need to conduct our study (e.g., patches that were released to fix the vulnerability). Hence, we manually analyzed each collected CVE instance to check whether the corresponding entries in the issue tracking system of the three studied projects were included in the NVD. In case NVD failed to provide this information, we searched the CVE ID in the project issue tracking system 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 obtained a total of 2,386 CVEs spanning across the lifetimes of these projects until January 2016. From these vulnerabilities, 1,252 were related to the Chromium project since 2008, 430 were associated with the PHP project published since 1997, and 704 were in the Thunderbird project, reported since 2002.

- **Removal of invalid CVEs:** While manually inspecting the CVEs in the previous step, we discarded *invalid vulnerabilities*, i.e., those CVE instances which were labeled as deprecated or as a duplicate of another CVE in the NVD, or CVEs that were not related to Chromium, PHP or Thunderbird (including applications written in PHP rather than in PHP itself). Furthermore, we discarded CVEs for which we could not identify a corresponding entry in the issue tracking system or when the issue was declared private in the issue tracking system, i.e., there were internal restrictions that prevented issues from being shown to the general public.

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

4.2.3. Identification of Tactical and Non-Tactical Vulnerabilities

Next, we used a bottom-up approach and a top-down approach to identify tactical vulnerabilities in the three systems.

In the **bottom-up** approach, we manually reviewed all CVE reports of the studied projects to classify them as *tactical* or *non-tactical*. To reduce effects of bias on this classification, we performed a peer evaluation by two developers (one with eight years of experience in software architecture and security and the other one with three years of experience in this field). These subject matter experts inspected all the collected CVEs and provided a *rationale* for how they classified CVE reports. To ensure consistency, each expert was provided with instructions for classifying CVEs, as shown in Table 5. The provided instructions ask these experts to read the CVE reports and its associated artifacts in order to identify where the issue is located and its root causes and provide a rationale and evidence for tactical vulnerabilities. It is important to highlight that Table 5 merely provide examples of low-level and tactical problems, but these examples are not meant to be exhaustive. 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 4.2.1, 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.

In the **top down** approach, we used our CAWE catalog (Section 3) as a gold standard to differentiate tactical and non-tactical vulnerabilities across the three systems. As shown in Figure 4, each CVE may have a CWE tag that can provide clues whether the problem is related to a security tactic or not. Thus, we use these tags to automatically classify CVEs as tactical or non-tactical (i.e., if the vulnerability's CWE tag is in our CAWE catalog, the vulnerability is considered as *tactical*). However, for some vulnerabilities, the NVD did not provide a CWE tag [11]. In this case, 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 (see Figure 4).

Finally, we consolidated the results of the bottom-up and top-down classifications and peer-reviewed the cases for which we observed mismatches between the bottom-up and top-down approach. There was a 93.3% agreement in the classification between bottom-up and top-down for Thunderbird, 90.2% in PHP and 88.3% in Chromium. These 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.

[11] 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.

Table 5: Instructions given to the experts to classify CVEs into tactical and non-tactical

Instructions	
Steps: (i) Read the CVE description, (ii) Check the modified code: comments, changed function/method/class, (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 authentication tactic	
- Improper handling of insufficient privileges in authorization tactic	
- Errors in tactical code and principles of the tactic.	
- CVE violates a design decision made by the developer.	
- Missing the 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*, *Reacting* to 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 impacted tactic?	
Your decision: Tactical (Yes) / Non-tactical (No)	☐ **Yes** ☐ **No**
Describe your rationale and provide evidence:	

4.2.4. *Overview of our Vulnerability Dataset*

Table 6 shows an overview of our vulnerability dataset, indicating the total number of collected vulnerabilities (**# CVEs**), the number of instances that were **discarded** (as explained in Section 4.2.3), the remaining CVEs that we **analyzed** and how many **tactical** and **non-tactical CVEs** we found in each system. From the vulnerabilities we analyzed in our dataset, we observed that 42.5% (403 out of 949 CVEs), 38.7% (63 out of 163 CVEs) and 38.2% (255 out of 668 CVEs) were tactical vulnerabilities in Chromium, PHP, and Thunderbird, respectively. From this dataset, we can observe that while these systems have implemented many security tactics to achieve security by design, a considerable number of reported vulnerabilities in these systems were due to incorrect implementations of these tactics.

Table 6: 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

4.3. *Using the Dataset to Answer our Research Questions*

From this analysis we obtained a dataset which contains, for each vulnerability, its *CVE ID*, its *Description*, the *Affected Releases*, its *type* (i.e., CWE tag), *associated tactic* (for tactical vulnerabilities) and the *Patch* that indicates the *source files* that were changed to fix the vulnerability as well as the total number of lines that were added/removed from these files. We used these collected artifacts as follows to answer RQ3 (Section 4.3.1) and RQ4 (Section 4.3.2).

4.3.1. *RQ3: Most Common Types of Tactical Vulnerabilities in the Case Studies*

To answer this question, we identified the most frequently occurring types of tactical CVEs in each project and their underlying security tactics. Table 7 lists the tactical vulnerability types 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 reoccurring vulnerability type in Thunderbird. Moreover, PHP's and Chromium's second most common vulnerability type was the *Inclusion of Functionality from Untrusted Control Sphere (CWE-829)*, which is about reusing/importing vulnerable third-party functionality.

Key findings for RQ3:

Table 7: Most Common Tactical Vulnerability Types in the Studied Projects

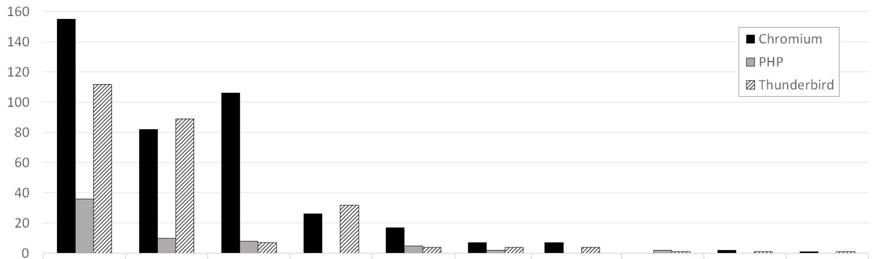
Security Tactic	Vulnerability Type	Chromium	PHP	Thund.	Total
Validate Inputs	CWE-20 Improper Input Validation	131	23	46	200
Limit Exposure	CWE-829 Inclusion of Functionality from Untrusted Control Sphere	106	8	7	121
Authorize Actors	CWE-284 Improper Access Control	35	–	51	86
Validate Inputs	CWE-79 Improper 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	–	1	–	1

– Improper Input Validation (CWE-20) and Improper Access Control (CWE-284) are the most occurring vulnerability types in Chromium, PHP and Thunderbird.

– 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.

4.3.2. *RQ4: Security Tactics Mostly Affected by Vulnerabilities in the Case Studies*

To answer this question, we identified the tactics associated with the CWE tags of the vulnerabilities across the three projects. This way, we computed how many times each security tactic was incorrectly adopted in the three systems. Figure 5 shows the number of CVEs per tactic. Most of the tactical issues in the studied systems are related to a failed mechanism that validates 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 7). 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 7) are common among the three systems.



	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 5: Total number of vulnerabilities (CVEs) per security tactic for each system

Key findings for RQ4:

– "Validate Inputs" and "Authorize Actors" are common tactics affected by tactical vulnerabilities in Chromium, PHP and Thunderbird.

5. Vulnerability Root Cause Analysis for Chromium, Thunderbird and PHP

To answer **RQ5**, we performed a qualitative analysis of the vulnerability reports to identify the *root causes* of vulnerabilities. We focused on the root causes of the top 20 most frequent types of tactical vulnerabilities (see Table 7). In the next subsections, we explain the qualitative data analysis we performed (Section 5.1) and the root causes of each tactical vulnerability type (Section 5.2).

5.1. Data Analysis to Identify Vulnerability Root Causes from Vulnerability Reports

We performed a qualitative analysis [36] of 632 vulnerability reports and their associated artifacts to identify the root causes of the most reoccurring tactical vulnerability types (listed in Table 7). This analysis comprised the following steps:

1. For each vulnerability, we studied the following artifacts: (i) vulnerability report; (ii) each comment in the issue tracking system made by developers and/or the reporter, (iii) the modified source code(s) in the patch released to fix the vulnerability, (iv) the patch's commit message, and (v) design documents [34, 37, 38, 39, 40, 41]. Through analyzing these artifacts, we filled out a template for each vulnerability. The template captured information regarding the *context* in which the vulnerability occurred, a brief description of the *problem*, including an explanation of the root cause and the consequences as well as the *solution* implemented by developers to fix the problem.

2. Two of the authors coded [36] vulnerability reports. During this coding process, they iteratively reviewed the *context* and *problems* of the vulnerabilities as captured in the previous step and annotated each vulnerability with a *code*, which indicates the root cause of the vulnerability. These coders also provided their *rationale* behind the decision to label the vulnerability with a specific code. As they performed the analysis, they either annotated the vulnerability reports (CVEs) with existing codes or created new codes that emerged from the data (if the existing codes were not suitable for the CVE being analyzed). For each created code, the authors also added its meaning into a "*codebook*" [36]. This codebook contained a *summary* of the root cause indicated by the code, associated *consequences* which indicated how the vulnerability affected the security mechanisms of these systems.

3. After coding all the CVEs, the last step was to refine the codebook. The goal of this step was to merge or split codes when needed to ensure the same level of granularity of these codes.

As a result of this rigorous analysis of CVEs, we obtained a "codebook" [36] which contains a list of *codes* per tactical vulnerability type and their corresponding *meaning*.

5.2. RQ5: Root Causes of the Most Common Tactical Vulnerability

Using the data from our qualitative analysis, we elaborate on the specific root causes that lead to tactical vulnerabilities in order to answer RQ5. For each root cause, we provide an *example*, the *impact* of the associated vulnerabilities on the system's security as well as a brief *explanation* of how these vulnerabilities were mitigated. For tactical vulnerabilities classified as an "*omission*" or "*commission*", our root cause analysis indicates which aspects of the associated security tactics were not chosen (omission) or incorrectly designed during the software design process (commission).

It is important to highlight that the majority of tactical vulnerability types are cases of "*realization*" weaknesses (see Table 7). As such, most of our root causes occurred during the implementation/maintenance of these tactics.

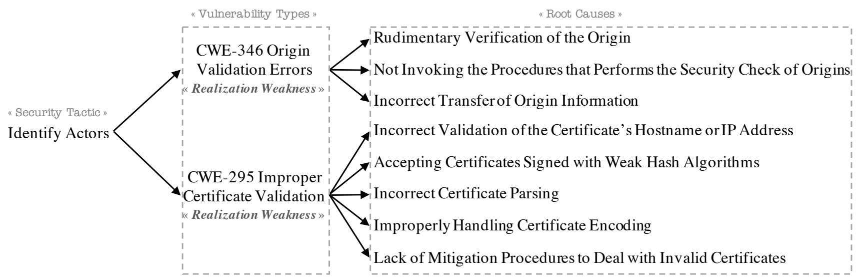

« Security Tactic »
Identify Actors
« Vulnerability Types »
CWE-346 Origin Validation Errors
« Realization Weakness »
CWE-295 Improper Certificate Validation
« Realization Weakness »
« Root Causes »
Rudimentary Verification of the Origin
Not Invoking the Procedures that Performs the Security Check of Origins
Incorrect Transfer of Origin Information
Incorrect Validation of the Certificate's Hostname or IP Address
Accepting Certificates Signed with Weak Hash Algorithms
Incorrect Certificate Parsing
Improperly Handling Certificate Encoding
Lack of Mitigation Procedures to Deal with Invalid Certificates

Figure 6: Root Cause Analysis of Tactical Vulnerabilities related to the "Identify Actors" Tactic

5.2.1. "Identify Actors" Tactic

This tactic was affected by **Origin Validation Errors** (**CWE-346**) and **Improper Certificate Validation** (**CWE-295**) in Chromium and Thunderbird (Figure 6). These tactical vulnerabilities occurred in these projects as follows:

- **CWE-346 Origin Validation Errors:** This tactical vulnerability type refers to classes of problems in which the application fails to correctly verify the validity of the source of data or communication. In Chromium and Thunderbird, this tactical vulnerability type occurred due to problems related with violations of the *Same-Origin Policy* (*SOP*) [42] and the *Content Security Policy* (*CSP*) [43], two complementary security policies commonly applied in Web applications to implement the "Identify Actors" tactic. In both policies, an origin of a Web resource is defined by the *scheme*, *host* and *port* of its URL [42].

On one hand, the *SOP* is used to enforce that documents/scripts loaded from different sources (i.e. *origins*) do not interact with each other. This means that scripts/documents can only access data from another document or script if they are from the same origin. On the other hand, the *CSP* is a complementary security control that allows Web servers to specify a whitelist of origins, which indicates the only sources of resources (e.g. scripts, HTML documents, etc) that should be trusted. This way, resources from an origin that does not match the list of trusted origins in the whitelist should be ignored by the client application. Violations of these two policies occurred due to:

- Rudimentary Verification of the Origin: the application has an ad-hoc implementation of the policy (SOP or CSP) which incorrectly checks the origin of a request.
  Example: According to the CSP specification [43], if the policy's hostname starts with a wildcard (e.g., "*.example.com"), then the system should only match subdomains (e.g., "a.example.com" or "b.example.com") but not the domain (i.e., "example.com). However, in Chromium's CVE-2015-6785 when the *host* part of a content security policy started with a wildcard (e.g., "*.domain.com") the system was mistakenly matching this host to resources originated from "domain.com", violating the expected behavior of the CSP.
  Impact: Although both systems applied the CSP and SOP as a mechanism to identify actors providing input to the system, the implementation of CSP and SOP failed to guarantee the basic premise that the identification of these actors is precise. It results in a bypass of the tactic's protection mechanism, which can be used by attackers to steal data (e.g. authentication tokens) or inject code.
  Recommendations: Developers should strictly follow existing specifications (e.g. [43, 42]) when implementing the CSP/SOP as means of adopting the "Identify Actors" tactic. In particular, having a centralized component that performs such policy enforcements minimizes the risk of inconsistent implementations. In fact, we observed multiple CVEs in which developers discussed deeper refactorings that involved moving the scattered origin checks to a central point to consistently enforce these policies concerning cross-origin requests.
- Not Invoking the Procedures that Perform the Security Check of Origins: the application does not invoke the necessary origin check procedures during a cross-origin request to load, execute or access a resource.
  Examples: In Thunderbird's CVE-2012-4192, the SOP implementation was not identifying the request origin before granting access to the properties of the `location` object, violating the Same-Origin Policy. This CVE was due to a regression issue: developers removed the calls to the functions that perform origin checks while fixing an unrelated defect. As another example, in Chromium's CVE-2015-1236 developers did not understand the expected behavior in a cross-origin request to read off-line audio samples, so they have not invoked the routines that would enforce the Same-Origin Policy in this case. This Chromium vulnerability allowed attackers to read an audio file (or a conversion of that file to an audio buffer) and to send the read data to a remote location.
  Impact: This flawed tactic implementation does not check the identity of the actors performing a request, leading to a policy-bypass. Attackers can therefore compromise the system's confidentiality and integrity (being able to read and/or modify data).
  Recommendations: Developers should call the origin check functions in all the components that handle cross-origin requests.
- Incorrect Transfer of Origin Information: The application does not transmit origin information from one process to the forked process or from one object to another.
  Example: In the case of Web page redirects, Thunderbird's Same-Origin Policy implementation did not expose the final URL to the component performing the origin check (CVE-2008-5507). It allowed remote attackers to bypass the policy using JavaScript to redirect the user to another domain (target of the attack).
  Impact: To apply both the *SOP* and *CSP* correctly when identifying actors, an important assumption is that the information about the origin is always available when the check needs to be performed. Otherwise, it results in a policy bypass, in which unauthorized actors would access the system's resources.
  Recommendations: The origin information needs to be passed (if needed) to child processes and/or objects. This is particularly important in a chain of redirects, in which the origin check should be based on the target URL (final URL) and not the original URL.

- **CWE-295 Improper Certificate Validation:** A common security mechanism across Web systems is to use digital certificates to check the identity of the actors that interact with the system. Each certificate contains multiple fields, such as an *expiration date*, *common name* (CN) and the *certification authority* (CA) that issued the certificate. One crucial aspect of using a certificate is to check whether it is *valid*. However, both Chromium and Thunderbird had flaws in the implementation of their certificate validation which were caused by:

  - Incorrect Validation of the Certificate's Hostname or IP Address: The system's implementation of the certificate validation only checked a portion of the *hostname* or *IP address* of a certificate to verify whether the certificate was issued to the entity performing the request.

Example: In Thunderbird's CVE-2010-3170, a certificate with a *CN* attribute equals to "*.168.3.48" was accepted as a valid certificate when it should have been treated as invalid because IP addresses in the CN attribute should not have wildcards ($*$).
Impact: The hostname/IP address in a certificate corresponds to the identity of the actor requesting for a connection. Implementing an incorrect hostname/IP address matching allows remote attackers to spoof trusted certificates and bypass the security tactic.
Recommendations: The implementation of the certificate validation should strictly follow existing guidelines [44] for matching the *CN* and *subjectAltNames* attributes before accepting the connection associated with the certificate.

- Accepting Certificates Signed with Weak Hash Algorithms: Occurs when the implemented certificate validation accepts certificates that were signed using less secure hashing algorithms.
  Example: Chromium's certificate validation routine accepted SSL connections to a Web site that provided an X.509 certificate signed with either the MD2 or MD4 hashing algorithms, which are not strong enough (CVE-2009-2973).
  Impact: It exposes the application to man-in-the-middle attacks.
  Recommendations: Developers should enforce and test that less secure hash algorithms (i.e., those that are at a higher risk of collision attacks) are not accepted by the certificate validation routine. This way, the application rejects connections from an actor that provides a certificate signed with a less secure algorithm.
- Incorrect Certificate Parsing: Occurs when the implemented certificate validation component incorrectly parses the attribute values within a certificate.
  Example: Thunderbird did not properly handle extra data in a signature that uses an RSA key with exponent 3, which allowed remote attackers to forge signatures for SSL/TLS and email certificates (CVE-2006-5462).
  Impact: An incorrect certificate parsing leads to wrong values in the certificate's attributes, affecting the certification validation routine. It results in crashes or misleading the application to accept connections from actors that provided malformed certificates.
  Recommendations: Each certificate may be provided in different file formats. Therefore, the tactic's implementation should have dedicated parsers implemented according to existing format specifications for each certificate type supported by the application.
- Improperly Handling Certificate Encoding: when certificates are used to identify actors, it is important to correctly recognize the encoding of the certificate, so that the actor information can be properly extracted from the certificate. However, we found instances in which the software's implementation does not correctly handle the certificate encoding.
  Example: In CVE-2014-1559 (Thunderbird), the tactic's implementation assumed that incoming X.509 certificates were encoded using UTF-8 if they were not in ASCII.
  Impact: An implementation that assumes the underlying encoding of certificates without actually checking the encoding can lead to incorrect parsing of the certificate. A malicious actor could leverage this vulnerability to spoof their identity.
  Recommendations: Certificate attributes may be encoded using different character sets (charsets). Thus, the implemented certificate validation routines should never expect certificates to be provided using a specific encoding. Instead, the implemented routine(s) must always infer the actual encoding used from the certificate attributes.
- Lack of Mitigating Procedures to Deal with Invalid Certificates: the tactic's implementation correctly parses and validates certificates, but it fails to properly handle invalid certificates.
  Example: In CVE-2014-7948 (Chromium), the certificate validation implementation did not correctly handle the error scenario (i.e., when the actor provides an invalid certificate). It resulted in Chromium caching resources from Websites with invalid certificates.
  Impact: It exposes the application to successful man-in-the-middle attacks.
  Recommendations: To avoid this problem, the certificate validation routine can throw an exception in case of invalid certificates. This exception is later captured in the code and prevents the attacker to bypass the security tactic.

5.2.2. "Authenticate Actors" Tactic

Chromium and Thunderbird suffered from **Improper Authentication** (**CWE-287**) issues, affecting their "Authenticate Actors" tactic.

- **CWE-287 Improper Authentication:** Systems interact with a multitude of actors during their operations. To ensure the security of a system, a commonly implemented mechanism is properly *authenticating* all actors interacting with a system. This is done to ensure that the system and other users know if an actor is whom they claim to be. These types of tactical vulnerabilities were caused by the following problems:
  - Incorrect Information About Entity Requesting Credentials in HTTP Authentication: The application does not display enough information about the entity requesting the credentials in an HTTP authentication.
    Example: When implementing HTTP Basic Authentication, Chromium displayed to the user the message provided by

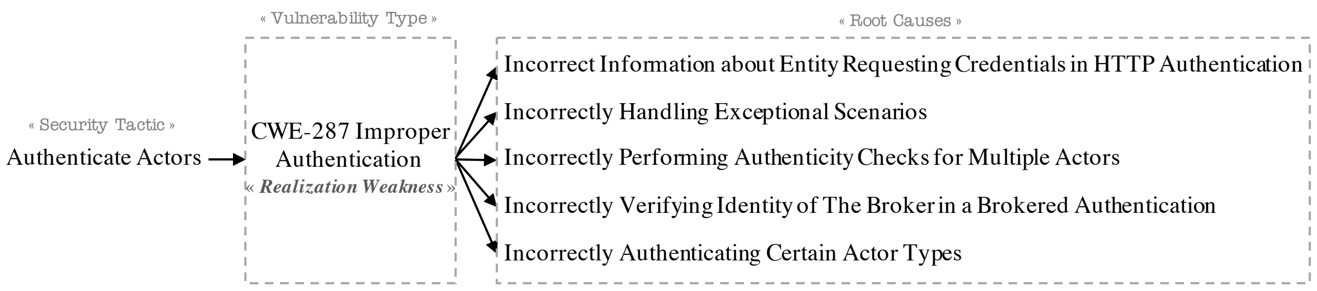

« Vulnerability Type »
« Root Causes »
« Security Tactic »
Authenticate Actors
CWE-287 Improper Authentication
« Realization Weakness »
Incorrect Information about Entity Requesting Credentials in HTTP Authentication
Incorrectly Handling Exceptional Scenarios
Incorrectly Performing Authenticity Checks for Multiple Actors
Incorrectly Verifying Identity of The Broker in a Brokered Authentication
Incorrectly Authenticating Certain Actor Types

Figure 7: Root Cause Analysis of Tactical Vulnerabilities related to Authenticate Actors Tactic

the server in the "*WWW-Authenticate*" HTTP header. The problem is that this message may be ambiguously written to lead the user to believe that the server is trustworthy (e.g. "*The site "www.trusted-website.com" is requesting your e-mail password for security purposes*").
Impact: One important aspect of the Authenticate Actor tactic's implementation is that the system makes users aware of which entity they are providing their credentials to. When users are unaware of which entity is requesting their credentials, it allows user-assisted attacks. Users may be tricked into trusting a fake entity with their credentials.
Recommendations: Developers should force the application's UI implementation to display the *entire* entity's origin (domain and scheme) along with the entity's provided message in an unambiguous fashion. As discussed by developers of the case studies, an approach to help solve ambiguity is to show the server's origin and provided message separately and with distinct labels to each of them.

– Incorrectly Handling Exceptional Scenarios: A general authentication implementation workflow is: (1) system requests actor's credentials; (2) the actor provides its credentials; (3) system checks whether credentials are valid. However, the actor might also cancel the authentication request during any of these steps. We found instances of vulnerabilities both in Thunderbird and Chromium in which the cancel request was not properly processed by the tactic implementation.
Example: When a user canceled the sign-in request to synchronize data, Chromium would still start the synchronization (CVE-2013-6643).
Impact: An attacker can exploit this flawed tactic implementation to bypass the authentication mechanism and steal data (e.g. passwords) without the victim's awareness.
Recommendations: Developers should implement an error handling mechanism that captures such exceptional scenarios involving failures or a cancel request.

– Incorrectly Performing Authenticity Checks For Multiple Actors: Occurs when the tactic's implementation concurrently receives multiple requests from different actors, but it checks only the authenticity for one of the actors in the request.
Example: In Thunderbird's CVE-2008-5022, an attacker could bypass the authenticity check through registering multiple listeners to the same event.
Impact: Although the "Authenticate Actors" tactic has been adopted into the system, its incorrect implementation results in an authentication bypass.
Recommendations: Developers should ensure that all requests are queued and the authenticity check is performed for each of these requests.

– Incorrectly Verifying the Identity of the Broker in a Brokered Authentication: In a brokered authentication [45], there is an authentication broker in charge of assigning security tokens to actors. This problem occurs when the brokered authentication implementation incorrectly verifies whether the token obtained by the actor was issued by a trustworthy authentication broker.
Example: The OAuth protocol allows redirecting to a Website after a successful authentication. In CVE-2013-6634, Chromium used the wrong URL (in a chain of redirects) when checking the identity of the broker that issued the token in the authentication. It allowed attackers to hijack user sessions.
Impact: It results in a bypass of the tactic.
Recommendations: Mitigating such a problem requires that the tactic's implementation verifies that tokens are signed by the issuing authentication broker. This implementation needs to take into account redirect scenarios in which the correct URL is the last one in a chain of redirects.

– Incorrectly Authenticating Certain Actor Types: Typically, a system has multiple types of actors interacting with it, such as end users (i.e. humans), machines, plug-ins, etc. We observed vulnerabilities in which the tactic's implementation did not authenticate a subset of these actors.
Example: In CVE-2013-0910, Chromium allowed plug-ins (an external actor) to be executed without checking their trustworthiness.

Impact: These actors would be granted access to the system, and be able to access unauthorized data.
Recommendations: While some actors are obvious (such as users) others might be more subtle and implicit (such as plug-ins or extensions). The fix requires ensuring that all actors (users or external programs) that interact with the software are identified, and authenticated.

5.2.3. "Limit Access" Tactic

This tactic is concerned with limiting access to resources such as memory, files, and network connections. Both PHP and Chromium had weaknesses in their "Limit Access" tactic related to **External control of File or Path** (**CWE-73**) and **Execution with Unnecessary Privileges** (**CWE-250**).

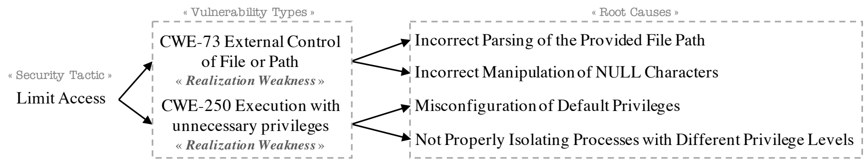

« Security Tactic »
Limit Access
« Vulnerability Types »
CWE-73 External Control of File or Path
« Realization Weakness »
CWE-250 Execution with unnecessary privileges
« Realization Weakness »
« Root Causes »
Incorrect Parsing of the Provided File Path
Incorrect Manipulation of NULL Characters
Misconfiguration of Default Privileges
Not Properly Isolating Processes with Different Privilege Levels

Figure 8: Root Cause Analysis of Tactical Vulnerabilities related to Limit Access Tactic

- **CWE-73 External Control of File or Path:** Both PHP and Chromium handle requests in which a path to a file resource is provided in order to access the file or perform a file-related operation (e.g., create a compressed archive of a directory). These requests are intended to be contained in a "safe area", meaning that files/directories outside this area should not be accessed. However, there were vulnerabilities in these systems that were resulting in an escape of this safe area due to the following problems:

  – Incorrect Parsing of the Provided File Path: The tactic's implementation incorrectly handled file paths that contained "." or ".." characters or that were *symbolic links*.
  Example: Chromium leveraged a user-provided filepath to open/create a database. By design, the callee is allowed to access any file inside a dedicated database directory (isolated). However, in Chromium's CVE-2014-1715, the implementation of this design decision did not check that the filepath was not a *symbolic link*, resulting in an attacker accessing files from the user.
  Impact: These "*dot-dot*" ("." or "..") characters and symbolic links can be used by attackers to access resources outside the safe area, thereby successfully bypassing the "Limit Access" tactic.
  Recommendations: To correctly enforce a safe area while implementing the "Limit Access" tactic, developers need to ensure that any externally provided path does not mistakenly escape this safe area. This involves checking for the presence of "*dot-dot*" sequences on the filepath as well as verifying whether the filepath points to an actual file and not a *symbolic link*.

  – Incorrect Manipulation of NULL Characters: The tactic implementation incorrectly handles a provided path that contains NULL-related characters (e.g. "\x00" or "%00" or "\0") while implementing the "Limit Access" tactic.
  Example: PHP (in CVE-2015-4025) truncated a provided filepath that contained a "\0" character.
  Impact: It allows attackers to bypass the tactic and access restricted files/directories.
  Recommendations: This problem is prominent in programming languages that require NULL characters as a way to terminate strings. From our observations, the problem can be mitigated by leveraging existing frameworks that handle invalid characters in a file path while implementing the tactic.

- **CWE-250 Execution with Unnecessary Privileges:** Chromium has a multi-process architecture, meaning that different components run in different processes. These processes communicate with each other through an Inter-Process Communication layer (IPC). Each of these processes may also have different privilege levels, based on their capabilities. Similarly, the PHP interpreter executes PHP scripts with different privilege levels. We found cases in which processes were executed with more privileges than intended, caused by the following:

  – Misconfiguration of Default Privileges: Occurred when the system's privileges default configuration is too loose, providing unnecessary privileges to processes.
  Example: The default permissions configuration of the PHP's process manager allowed any user to run arbitrary code with the same permission level of the process manager (CVE-2014-0185).
  Impact: An attacker can leverage this vulnerability to perform over-privileged operations.
  Recommendations: The tactic's implementation should protect the system by default following the *least privilege* secure design principle [46]. When implementing software that will execute in a shared environment, an alternative is to

specify the default permissions to only allow read/write access to the file owners and other users in the group (e.g. 660 permission in Unix-based systems).

– Not Properly Isolating Processes with Different Privilege Levels: Processes in a sandboxed environment must only interact with resources and/or processes from within the sandbox. Thus, they should not be communicating with higher-level processes and/or processes with different privilege levels. However, we found cases in which the system failed to deny the communication between processes with different privilege levels.
Example: In CVE-2012-2846, Chromium allowed sandboxed processes to use the Unix *ptrace* command to manipulate Chrome's UI process in order to execute arbitrary code.
Impact: This improper process isolation implementation results in a lower privileged process leveraging a process outside the safe area to perform an operation at a higher privilege level.
Recommendations: We observed two alternatives to mitigate this problem: (1) starting the sandboxed process at low-integrity level (for performing required initialization tasks), then dropping these privileges to the minimum after the initialization is complete; (2) adding a policy to the sandbox engine that the sandboxed process cannot invoke system calls that are used to manipulate other processes (such as `ptrace`).

5.2.4. "Authorize Actors" Tactic

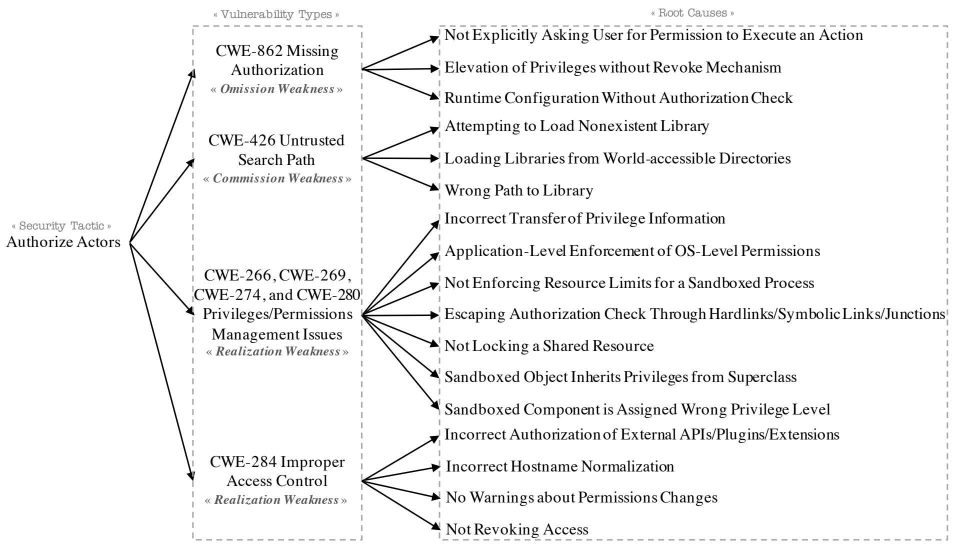

« Security Tactic »
Authorize Actors
« Vulnerability Types »
CWE-862 Missing Authorization
« Omission Weakness »
CWE-426 Untrusted Search Path
« Commission Weakness »
CWE-266, CWE-269, CWE-274, and CWE-280 Privileges/Permissions Management Issues
« Realization Weakness »
CWE-284 Improper Access Control
« Realization Weakness »
« Root Causes »
Not Explicitly Asking User for Permission to Execute an Action
Elevation of Privileges without Revoke Mechanism
Runtime Configuration Without Authorization Check
Attempting to Load Nonexistent Library
Loading Libraries from World-accessible Directories
Wrong Path to Library
Incorrect Transfer of Privilege Information
Application-Level Enforcement of OS-Level Permissions
Not Enforcing Resource Limits for a Sandboxed Process
Escaping Authorization Check Through Hardlinks/Symbolic Links/Junctions
Not Locking a Shared Resource
Sandboxed Object Inherits Privileges from Superclass
Sandboxed Component is Assigned Wrong Privilege Level
Incorrect Authorization of External APIs/Plugins/Extensions
Incorrect Hostname Normalization
No Warnings about Permissions Changes
Not Revoking Access

Figure 9: Root Cause Analysis of Tactical Vulnerabilities related to Authorize Actors Tactic

The "Authorize Actors" tactic had instances of tactical vulnerability types **Improper Access Control** (**CWE-284**), **Privilege/Permission Management Issues** (**CWE-266**, **CWE-269**, **CWE-274**, and **CWE-280**), **Untrusted Search Path** (**CWE-426**) and **Missing Authorization** (**CWE-862**).

- **CWE-862 Missing Authorization**: This tactical vulnerability type is a consequence of not adopting the "Authorize Actors" tactic such that the system performs authorization checks before an operation takes place. We found instances of this weakness in the three case studies caused by the following:

  – Not Explicitly Asking the User for Permission to Execute an Action: The system's design does not adopt an authorization mechanism that explicitly asks the user if the system is allowed to perform a certain task or grant access to a certain resource.
  Example: Chromium executed JRE applets without explicitly asking permissions from the user (CVE-2011-3898).
  Impact: It allows attackers to perform malicious activities without user awareness.
  Recommendations: The mitigation involves identifying the actions that require user mediation, in which case the system needs to adopt the *Authorize Actors* tactic in order to request user consent.

  – Elevation of Privileges Without Revoke Mechanism: This occurs when the system is designed to load extensions/plug-ins that are allowed to perform privileged actions without any configuration that could restrict or drop privileges.
  Example: PHP allowed the *libxslt* extension to create and write to files. There was no configuration to allow end-users to revoke this privilege (CVE-2012-0057).
  Impact: The system remains unprotected from data tampering.

Recommendations: The mitigation procedure consists in adopting the "Authorize Actors" tactic such that it has configuration parameters that enable/disable specific types of operations (reading files, accessing networks, creating directories, etc).

  - Runtime Configuration Without Authorization Check: The system allows the change of security-sensitive settings at runtime without any authorization check.
    Example: A vulnerability in PHP (CVE-2007-5900) occurred because attackers were able to overwrite protected configurations using the function "ini_set()" from the PHP language.
    Impact: Attackers can leverage this vulnerability to overwrite configuration parameters.
    Recommendations: It requires defining the subset of security properties that are read-only at runtime. This way, the application adopts the "Authorize Actors" tactic to check whether the entry is allowed to be modified at runtime before modifying its value upon request.

- **CWE-426 Untrusted Search Path**: Both Chromium and Thunderbird load system libraries at runtime to perform certain tasks. The main threat related to loading libraries is that they cross trust boundaries (i.e. they are not under the direct control of the system), so an attacker could leverage this to inject a malicious copy of the desired library. In both systems, we found vulnerabilities that allowed attackers to execute an arbitrary library caused by:

  - Attempt to Load an Inexistent Library: The system's design does not take into account different operating system versions in which the system may run. This way, the application attempts to dynamically load a library that does not exist on the underlying operating system.
    Example: There was a vulnerability in Thunderbird caused by attempting to load the "dwmapi.dll" library on all Windows versions. However, this DLL is only available on versions after Windows XP, which means that intruders could place a malicious "dwmapi.dll" in the working directory of a machine with a Windows XP and have their malicious code successfully executed.
    Impact: This allows attackers to create a malicious library placed in the expected location, resulting in the system executing this fake library code.
    Recommendations: To mitigate the problem, during the system's design, create a list of libraries per operating system version. In this case, the application first verifies whether such library would exist in the underlying OS even before attempting to load it to the memory.
  - Loading Libraries from World-Accessible Directories: The application attempts to search for the desired library dynamically, but the devised search algorithm includes unsafe directories (i.e., directories which are world-readable such as the current working directory).
    Example: Thunderbird attempted to load the "wsock32.dll" through using the dynamic search algorithm provided by the Windows API (CVE-2012-1943). This Windows API attempts to find the library from many locations, including the directory from which the application was loaded and the working directory of the parent process, which are potentially unsafe (as they are not read/write protected).
    Impact: Since these directories are not protected by default against modifications, an attacker could place a malicious library in one of these unsafe locations and execute it.
    Recommendations: Fixing the problem can be performed in two ways. The first approach is to design a library-loading mechanism that uses absolute file paths to access the library. Another approach is to design the system to load libraries only from the system directories, which are protected by default against public reading and writing.
  - Wrong Path to Library: The system is designed to hardcode paths to a library, which can lead to the execution of the wrong library if the hardcode path is incorrect.
    Example: During an install on Windows machines, Thunderbird would execute the code from an executable "program.exe" located at "C:\" instead of the executable placed in its installation directory.
    Impact: It allows local attackers to execute arbitrary code through a Trojan horse executable file placed in the system's root directory.
    Recommendations: It requires designing the system such that the hardcoded paths are according to the underlying operating system and version.

- **CWE-266, CWE-269, CWE-274, and CWE-280 Privileges/Permissions Management Issues:**

  - Incorrect Transfer of Privilege Information: To perform authorization, a basic premise is that the permissions and privileges are available at all times when the authorization check is to be performed. This means that the permissions and privileges information needs to be propagated (if needed) to child process/objects/etc before the authorization takes place.
    Example: In Chromium, users are allowed to select certain Websites that are authorized to load plug-ins into the browser. In CVE-2010-2108 the list of blocked Websites is not transmitted to the component that performs the authorization check,

resulting in untrusted websites to execute arbitrary plug-ins without user consent.
Impact: It results in a bypass of the "Authorize Actors" tactic.
Recommendations: The tactic's implementation should transmit any privilege information to the component that performs the authorization check.

– Application-Level Enforcement of OS-level Permissions: Files within a computer typically have a list of permissions that indicate the subjects allowed to access them. This type of enforcement is performed by the underlying operating system. However, we found vulnerabilities in PHP which were caused by attempting to perform this OS-level permissions enforcement at the application level.
Example: PHP had the `safe_mode` configuration parameter in prior version 5.4.0 to enforce access control to files and directories on the Web server running the PHP interpreter. The goal of this parameter is to avoid scripts from different applications running on the same server accessing files/directories from each other. However, attempting to enforce resources permissions at the application level is inappropriate. Moreover, such enforcement mechanisms needed to be implemented throughout the modules of PHP that performed any file-related operations. However, there were several cases that developers did not 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 be exposed to security breaches.
Impact: Attackers can bypass this application-level enforcement.
Recommendations: In shared execution environments, applications should not attempt to protect their files from access. Instead, they should configure the access control lists of the underlying operating system.

– Not Enforcing Resource Limits for a Sandboxed Process: The implementation of the sandboxing mechanism that enforces that processes run isolated from each other, incorrectly enforces a threshold that dictates the maximum amount of resources that a process can use (such as memory).
Example: In CVE-2015-3335, Chrome's Native Client implementation did not enforce limits for data usage, allowing "row-hammer" attacks.
Impact: It affects the system's performance. Such excessive resources usage can enable an attacker to implement sandbox escaping attacks through memory manipulation or to attempt to access data files which are not necessary for the performance of the sandboxed process.
Recommendations: Developers should ensure that the tactic's implementation enforces that sandboxed processes have a threshold value that limits access to system resources, ranging from logical resources (such as user data) to hardware ones (e.g. CPU).

– Escaping Authorization Check Through Hardlinks/Symbolic Links/Junctions: This problem occurs when the implemented sandboxing mechanism follows links that go outside the safe area, bypassing the protection mechanism.
Example: In CVE-2013-1672, Thunderbird's update service does not take into account the existence of junctions, which allow a local attacker to trigger the execution of a malicious executable during an automatic update.
Impact: Such a mistake in the implementation of the "Authorize Actors" tactic enables attackers to bypass the sandboxing solution.
Recommendations: The fix involves not following the links provided inside in a sandbox that are pointing to locations outside the defined safe area.

– Not Locking a Shared Resource: Developers do not lock read/write access to a sensitive file while using it.
Example: Thunderbird did not lock write access to an archive file, allowing local attackers to perform trojan attacks.
Impact: Attackers could leverage race conditions to modify the file and get the process to use that corrupted file, rather than the original file.
Recommendations: Fixing the problem involves (i) locking the shared resource; (ii) checking its integrity/trustworthiness (verify whether it has not been modified) and then using it (releasing the lock after the task is completed).

– Sandboxed Object Inherits Privileges from Superclass: This occurs when developers create an object which is meant to run in a sandboxed area. However, this object's class inherits methods from a superclass which is not sandboxed, meaning that there are some methods that run without privileges (bypassing the sandbox protection area).
Example: Thunderbird allowed attackers to create objects outside the sandbox and then leverage calls to the `valueOf()` method to escape the sandbox (CVE-2006-2787).
Impact: It leads to privilege escalation and remote code execution.
Recommendations: To prevent this problem, the implementation of the "Authorize Actors" tactic needs to check that the pointer of the object being manipulated ("this") is within the right privilege level.

– Sandboxed Component is Assigned Wrong Privilege Level: Occurs when the tactic's implementation allows a lower privileged component to be granted more permissions than intended by the design.
Example: In CVE-2010-4041, Chromium executed worker processes outside the sandboxed environment.
Impact: A sandboxed component should have a defined level of privilege. Different components in a sandbox may have

different privilege levels, according to the tasks they perform. This problem occurs when developers fail to check the context and functionality of the component and therefore, grant incorrect privilege levels. Such an error in the implementation of the tactic can result in *Cross-Site Request Forgery* (CSRF), an attack that forces the user to execute unwanted actions on a web application in which they're currently authenticated.
Recommendations: The mitigation involves passing the code to the sandbox environment before execution.

- **CWE-284 Improper Access Control:**
  - Incorrect Authorization of External APIs/Plug-ins/Extensions: the application has a flawed authorization implementation for external programs (i.e., APIs, plug-ins, extensions, and libraries).
    Example: In CVE-2013-1717, Thunderbird did not perform an authorization check before granting access to local files to Java applets (a plug-in).
    Impact: Unauthorized APIs/Plug-ins/Extensions gain access to the data or control of other extensions/plug-ins. It can also tamper with the internal integrity of the system.
    Recommendations: The fix involve adding an intermediary protection layer between the application core's functionalities and plug-ins/external libraries. This intermediary layer is in charge of performing authorization checks.
  - No Warnings About Permissions Changes: When the system allows extensions or plug-ins to change their permissions at runtime, the implementation of the "Authorize Actors" tactic does not warn the end user.
    Example: Chromium did not display a warning indicating that a malicious plug-in has access to the camera (CVE-2015-3334).
    Impact: Attackers can use plug-ins or extensions to collect users' data without their awareness (e.g. camera or microphone).
    Recommendations: Any permission elevation requested by plug-ins/extensions have to be mediated by the user through confirmation dialogs.
  - Incorrect Hostname Normalization: The system's tactic implementation leverages the hostname to check whether an actor is allowed to perform certain tasks, but it incorrectly normalizes the hostname during the authorization check.
    Example: An extra dot (".") at the end of the hostname in the Chromium project has misled the authorization mechanism, leading to a bypass vulnerability (CVE-2015-1269).
    Impact: A mistake in the implementation of the hostname identification and matching can result in a bypass of the authorization tactic.
    Recommendations: Mitigation procedures include normalization of hostnames and rejecting hostnames with invalid characters.
  - Not Revoking Access: The system's tactic implementation does not revoke access to the resource when it is not being used anymore.
    Example: Chromium allowed remote attackers to obtain video camera data through a session that remains active even though the user had navigated away from the webpage (CVE-2014-1586).
    Impact: It corresponds to a violation of the least privilege design principle [46]. It allows attackers to steal sensitive data without user awareness.
    Recommendations: Drop access to privileges as soon as the resource is not being used anymore.

5.2.5. "Validate Inputs" Tactic

The following results were obtained for the Validate Inputs tactic:

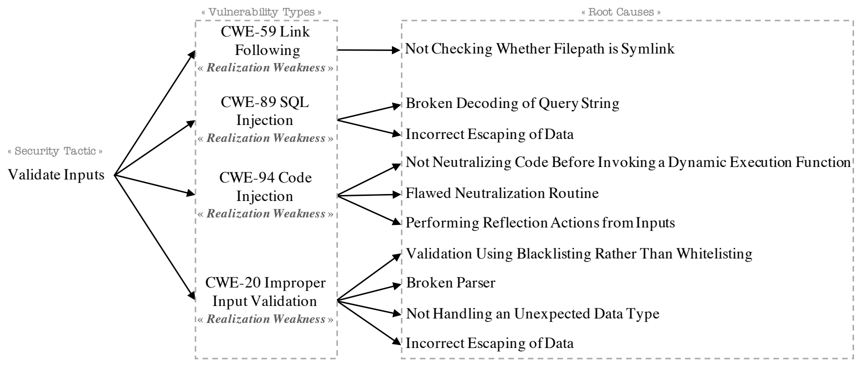

« Security Tactic »
Validate Inputs
« Vulnerability Types »
CWE-59 Link Following « Realization Weakness »
CWE-89 SQL Injection « Realization Weakness »
CWE-94 Code Injection « Realization Weakness »
CWE-20 Improper Input Validation « Realization Weakness »
« Root Causes »
Not Checking Whether Filepath is Symlink
Broken Decoding of Query String
Incorrect Escaping of Data
Not Neutralizing Code Before Invoking a Dynamic Execution Function
Flawed Neutralization Routine
Performing Reflection Actions from Inputs
Validation Using Blacklisting Rather Than Whitelisting
Broken Parser
Not Handling an Unexpected Data Type
Incorrect Escaping of Data

Figure 10: Root Cause Analysis of Tactical Vulnerabilities related to Validate Input Tactic

- **CWE-59 Link Following:**
  - Not Checking Whether Filepath is Symbolic Link: This occurs when the tactic's implementation receives a file path as input but it does not check whether the symlink resolves to an unprotected file.
    Example: PHP's configuration script uses a predictable filename in `/tmp/` to store temporary installation files. A local attacker could replace that file by a symlink and be able to overwrite/delete user files (CVE-2014-3981).
    Impact: It allows local attackers to read/overwrite/delete user files.
    Recommendations: The fix involves verifying the type of the file before performing any file operations. In the case of symlinks, the implementation then checks what is the file/directory that the link resolves.

- **CWE-89 SQL Injection:**
  - Incorrect Escaping of Data: Special characters (quotes, backslashes, etc) are not escaped or removed in a SQL query enabling an attacker to implement a SQL injection attack.
    Example: PHP used to not escape characters of externally provided SQL query string as input to the function *mysqli_fetch_assoc* (CVE-2010-4700).
    Impact: It tampers with the integrity of data stored in relational databases.
    Recommendations: Fixing involving escaping some characters that are part of the SQL syntax, such as back/forward slashes, quotes (double and single), percentages (%) etc.

- **CWE-94 Code Injection:** This is a tactical vulnerability type that occurs when malicious code segments are created based on external inputs. In this case, attackers can provide inputs in the form of code syntax, thereby injecting malicious behavior to the software. Without implementing checks on the type of input, attackers can inject their malicious codes into the application's runtime behavior to collect data or disrupt the application. In our analysis we have found that some of the underlying causes of this type of vulnerability are the following:
  - Not Neutralizing Code Before Invoking a Dynamic Execution Function: Occurs when the application's input validation implementation execute code provided as a string input without neutralizing any code injected in the input.
    Example: Thunderbird's built-in XML Binding Language (XBL) allowed intruders to execute arbitrary code due to incorrect input validation in the following XBL binding methods: *valueOf.call* and *valueOf.apply* (CVE-2006-1733).
    Impact: It allows attackers to execute arbitrary code.
    Recommendations: In interpreted languages such as Java, we commonly have a function/method that can execute code provided as a String input (e.g., `eval()` in JavaScript). In this case, to ensure that only safe commands are passed to the function, the software needs to parse the provided input in order to detect and remove unsafe commands from the input, before passing it to the execution engine.
  - Flawed Neutralization Routine: Occurs when developers implemented a function/method to neutralize unsafe externally provided commands, but the routine does not correctly cover all the possible types of unsafe commands.
    Example: Thunderbird's CVE-2012-3980 allowed attackers to inject arbitrary JavaScript code with higher privileges through forwarding this code to an *eval* operation. The eval operation did not neutralize the unsafe commands provided by the attacker.
    Impact: The attackers can provide unsafe commands as input and cause corrupted memories and other issues.
    Recommendations: The observed examples required adding verifications about the context of the call (i.e., from a lower / higher privileged actor) and performing the neutralization accordingly.
  - Performing Reflection Actions from Inputs: Some interpreted languages like Java support reflection. In this case, similar to "*Not Neutralizing Code Before Invoking a Dynamic Execution Function*", developers were not implementing a command neutralization routine before performing reflection operations based on user-provided inputs.
    Example: In Firefox's CVE-2006-1735, the JavaScript engine allowed attackers to retrieve a constructor from XBL compilation scope through leveraging a reflection call.
    Impact: It can be used to execute arbitrary code, tamper with the application expected behavior or elevate privileges.
    Recommendations: The mitigation procedures for this problem are (i) implement code neutralization procedures for the user provided data before adopting it in a reflection context; or (ii) hide reflection calls from external actors.

- **CWE-20 Improper Input Validation:** Different segments of the software expects user input, which needs to be validated against different requirements related to its type, size, boundary values, etc. If these requirements are not satisfied and the system receives unintended input, an altered control flow, arbitrary code execution or control of a resource can occur. According to our analysis, this type of tactical vulnerability occurred due to the following causes:
  - Validation Using Blacklisting rather than Whitelisting: Occurs when the system uses blacklists rather than a whitelist-based approach for input validation.

Example: Chrome's CVE-2009-3931 used a blacklist of files to block the download of certain dangerous file extensions, but the blacklist was incomplete: it did not cover potentially dangerous extensions such as "*(1) .mht and (2) .mhtml files, which are automatically executed by Internet Explorer 6; (3) .svg files, which are automatically executed by Safari; (4) .xml files; (5) .htt files; (6) .xsl files; (7) .xslt files; and (8) image files that are forbidden by the victim's site policy*"[12].
Impact: Implementing the "Validate Input" tactic based on blacklists is prone to implementation mistakes: the validation mechanism may not cover all possible malicious input types. It allows attackers to craft special inputs that are not covered by the blacklist.
Recommendations: In our data, we observed that the problem can be fixed through (i) adding the missing malicious data into the blacklist or converting the validation routine to a whitelist-based approach. Blacklists enumerate the prohibited input types in the code, whereas whitelists enumerate the accepted input types in the code. Generally, using whitelists is the safest approach. Blacklists are prone to mistakes, as the likelihood of it containing all the potential ways the input can go wrong is low. This, in turn, increases the chances that attackers can figure out a way in which the input does not violate the blacklist, but is still harmful to the software.

– Not Handling an Unexpected Data Type: This problem occurs when systems do not handle unexpected input data types properly. It can also be referred to as a "Type Confusion".
Example: PHP's input validation implementation assumed that a provided input was of an array type without actually checking this assumption (CVE-2015-4148).
Impact: This rudimentary implementation of the tactic results in crashes.
Recommendations: Usually, input received from the user needs to be of a certain type. Developers should implement checks to ensure that the input's data type is the correct one. Moreover, they need to also develop routines that handle situations where unexpected data types are provided as input. The system needs to be able to recognize that the incorrect input type has been provided and proceed with the rest of the functionalities in the aforementioned scenario.

– Broken Parser: The system requires a data structure provided as input and needs to parse it. However, the accuracy and the level of inclusiveness of the parsing method used may be faulty and fail its initial purpose. If this parsing method fails to parse the structure in such a way that it can extract its values, a broken parser problem occurs.
Example: In CVE-2014-7899, Chromium did not correctly parse a URL starting with "blob: " followed by a URL and a long username. It allowed attackers to spoof the URL bar.
Impact: This rudimentary implementation of the tactic results in an application crash or usage of wrong values, affecting the system's logical behavior.
Recommendations: When implementing the input validation tactic, developers should check the data structure received as input against a data schema.

– Incorrect Escaping of Data: The system does not correctly neutralize "control" characters in an input.
Example: Chromium incorrectly escaped input, allowing that the content in an `href` attribute to be rendered as regular HTML entities. It allowed attackers to steal data or CSRF tokens (CVE-2015-6790).
Impact: The rudimentary implementation of a validation input tactic may result in arbitrary code execution and memory corruption
Recommendations: Fixing this tactic implementation requires two things (i) identifying the underlying context in which the data will be used and (ii) adopting escaping procedures according to this context. For instance, in HTML rendering context, a user input should be escaped to HTML entities (e.g. "<html>" is escaped as "&lt;html&gt;") before rendering it to a Web page.

6. Threats to Validity

This section discusses validity threats based on the validation scheme presented by Runeson and Hoest [31] (*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 4). To identify the types of vulnerabilities, we leveraged 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. Consequently, we may have missed vulnerabilities that were not tracked by the NVD. Also, we had to discard vulnerabilities because we could not find the corresponding entry in the issue tracking system or the issue was still private at 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 in our study is related to the manual analysis of CVE instances in order to observe the nature of security design

[12]https://nvd.nist.gov/vuln/detail/CVE-2009-3931

issues and to identify tactical and non-tactical vulnerabilities. To mitigate this threat, we performed both top-down and bottom-up classification of the vulnerabilities (see Section 4). 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. We consider that the peer evaluation minimized the impacts of biases and mistakes by the manual inspection of CVEs.

External validity evaluates the generalizability of our findings. There are two threats in this respect:

- We analyzed the historical vulnerability reports from three systems (PHP, Chromium, and Thunderbird), which are Internet applications and mostly implemented in C/C++. Here, we do not aim for statistical generalization, but analytical generalization: we carefully selected the three systems from different software domains and with a high number of reported vulnerabilities. Therefore, we expect the systems to be representative of a typical large-scale software engineering environment. Also, when discussing our results, we highlighted which findings are specific to a system and which findings apply to all systems.
- We identified the root causes of vulnerabilities based on a subset of types of vulnerabilities from the CWE catalog (Section 4.2.3). 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.

7. Conclusions and Future Work

This paper presented the concept of CAWE (Common Architectural Weakness Enumeration), a catalog of common types of architectural weaknesses. This catalog constitutes an effort towards stimulating critical reflections about security-related issues in developers to avoid fundamental design problems at both architectural design and implementation time. Furthermore, the catalog helps researchers to develop novel techniques to identify and mitigate such flaws. Currently, the catalog enumerates 223 architectural weaknesses, documenting how these weaknesses may affect security tactics. As future work, we plan to evaluate our catalog with security experts, in order to expand it.

Furthermore, 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. We discovered that the improper implementation of the "Authorize Actors", "Validate Inputs" and "Encrypt Data" security tactics may cause the highest number of potential problems. In the three systems, the tactics most impacted by vulnerabilities are "Validate Inputs", "Authorize Actors", and "Limit Exposure". Further, our analysis suggests that "Improper Input Validation" is the most common type of vulnerability across all three systems.

Lastly, looking more in-depth the most common types of tactical vulnerabilities, we analyzed and categorized their root causes. This helps architects and researches aware of the most common mistakes they can make that can introduce vulnerabilities in a system.

Acknowledgments

This work was partially funded by the US National Science Foundation under grant numbers CNS-1823246, CNS-1629810, CNS-1816845 and IIP-0968959 under funding from the S2ERC I/UCRC program and US Department of Homeland Security.

References

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

[2] M. Galster, M. Mirakhorli, J. Cleland-Huang, J. E. Burge, X. Franch, R. Roshandel, P. Avgeriou, Views on software engineering from the twin peaks of requirements and architecture, SIGSOFT Softw. Eng. Notes 38 (5) (2013) 40–42. `doi:10.1145/2507288.2507323`.
URL `http://doi.acm.org/10.1145/2507288.2507323`

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

[4] E. B. Fernandez, H. Astudillo, G. Pedraza-García, Revisiting architectural tactics for security, in: 9th European Conference on Software Architecture (ECSA), 2015, pp. 55–69.

[5] M. Mirakhorli, Preserving the quality of architectural decisions in source code, Ph.D. thesis, PhD Dissertation, DePaul University Library (2014).

[6] C. Izurieta, J. M. Bieman, How software designs decay: A pilot study of pattern evolution, in: ESEM, 2007, pp. 449–451.

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

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

[9] M. Hafiz, M. Fang, Game of detections: how are security vulnerabilities discovered in the wild?, Empirical Software Engineering (2015) 1–40.

[10] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, 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, IEEE, 2013, pp. 65–74.

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

[12] IEEE Center for Secure Design, Avoiding the top 10 software security design flaws, `http://cybersecurity.ieee.org/center-for-secure-design/`, (Accessed on 10/06/2016) (2015).

[13] J. C. S. Santos, K. Tarrit, M. Mirakhorli, A catalog of security architecture weaknesses, in: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), 2017, pp. 220–223. `doi:10.1109/ICSAW.2017.25`.

[14] J. C. S. Santos, A. Peruma, M. Mirakhorli, M. Galster, J. V. Vidal, A. Sejfia, Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird, in: 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 69–78. `doi:10.1109/ICSA.2017.39`.

[15] M. Monperrus, A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 234–242.

[16] F. Bachmann, L. Bass, M. Klein, Deriving architectural tactics: A step toward methodical architectural design, Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST (2003).

[17] R. C. Merkle, Secure communications over insecure channels, Communications of the ACM 21 (4) (1978) 294–299.

[18] J. Ryoo, P. Laplante, R. Kazman, Revising a security tactics hierarchy through decomposition, reclassification, and derivation, in: Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on, IEEE, 2012, pp. 85–91.

[19] C. Preschern, Catalog of security tactics linked to common criteria requirements, in: Proceedings of the 19th Conference on Pattern Languages of Programs, The Hillside Group, 2012, p. 7.

[20] C. Blanco, J. Lasheras, R. Valencia-García, E. Fernández-Medina, A. Toval, M. Piattini, A systematic review and comparison of security ontologies, in: 3rd International Conference on Availability, Reliability and Security (ARES), 2008, IEEE, 2008, pp. 813–820.

[21] Y. Wu, R. A. Gandhi, H. 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, ACM, 2010, pp. 22–28.

[22] T. Heyman, R. Scandariato, 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, IEEE, 2012, pp. 41–50.

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

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

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

[26] B. J. Berger, K. Sohr, R. Koschke, Extracting and analyzing the implemented security architecture of business applications, in: 17th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2013, pp. 285–294. `doi:10.1109/CSMR.2013.37`.

[27] S. Al-Azzani, 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), IEEE, 2012, pp. 51–60.

[28] E. Taspolatoglu, R. Heinrich, Context-based architectural security analysis, in: 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), 2016, pp. 281–282.

[29] J. Ryoo, B. Malone, P. A. Laplante, P. Anand, The use of security tactics in open source software projects, IEEE Transactions on Reliability 65 (3) (2016) 1195–1204. doi:10.1109/TR.2015.2500367.

[30] J. C.-H. Mehdi Mirakhorli, Detecting, tracing, and monitoring architectural tactics in code, IEEE Trans. Software Eng.doi:doi:10.1109/TSE.2015.2479217.
URL http://doi.ieeecomputersociety.org/10.1109/TSE.2015.2479217

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

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

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

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

[35] M. Mirakhorli, A. Fakhry, A. Grechko, M. Wieloch, 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.

[36] M. B. Miles, A. M. Huberman, J. Saldana, Qualitative data analysis, Sage, 2013.

[37] C. Reis, A. Barth, C. Pizano, Browser security: lessons from google chrome, Communications of the ACM 52 (8) (2009) 45–49.

[38] N. Carlini, A. P. Felt, D. Wagner, An evaluation of the google chrome extension security architecture., in: USENIX Security Symposium, 2012, pp. 97–111.

[39] The PHP Group, Php: Security - manual, http://php.net/manual/en/security.php, (Accessed on 02/01/2018) (2018).

[40] Mozilla, Mail client architecture overview, https://developer.mozilla.org/en-US/docs/Mozilla/Thunderbird/Mail_client_architecture_overview, (Accessed on 02/01/2018) (2013).

[41] Mozilla, Network security services, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS, (Accessed on 02/01/2018) (2018).

[42] A. Barth, Rfc 6454 - the web origin concept.

[43] S. Stamm, B. Sterne, G. Markham, Reining in the web with content security policy, in: Proceedings of the 19th International Conference on World Wide Web, WWW '10, ACM, New York, NY, USA, 2010, pp. 921–930. doi:10.1145/1772690.1772784.
URL http://doi.acm.org.ezproxy.rit.edu/10.1145/1772690.1772784

[44] E. Rescorla, HTTP over TLS, https://www.rfc-editor.org/rfc/rfc2818.txt, (Accessed on 02/01/2018) (2000).

[45] J. Howard, D. Schiappa, K. Ahmed, K. Young, Authentication broker service, uS Patent 7,607,008 (Oct. 20 2009).
URL https://www.google.com/patents/US7607008

[46] J. H. Saltzer, M. D. Schroeder, The protection of information in computer systems, Proceedings of the IEEE 63 (9) (1975) 1278–1308.