6-2021

Understanding and Identifying Vulnerabilities Related to Architectural Security Tactics

Joanna Cecilia da Silva Santos
jds5109@rit.edu

Understanding and Identifying Vulnerabilities Related to Architectural Security Tactics

by

Joanna Cecilia da Silva Santos

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
June, 2021

Understanding and Identifying Vulnerabilities Related to Architectural Security Tactics

by

Joanna Cecilia da Silva Santos

Committee Approval:
We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Mehdi Mirakhorli Dissertation Advisor	Date
Dr. Andrew Meneely Dissertation Committee Member	Date
Dr. Sumita Mishra Dissertation Committee Member	Date
Dr. Hossein Hojjat Dissertation Committee Member	Date
Dr. Dan Phillips Dissertation Defense Chairperson	Date

Certified by:

Dr. Pengcheng Shi Ph.D. Program Director, Computing and Information Sciences	Date

Understanding and Identifying Vulnerabilities Related to Architectural Security Tactics

by

Joanna Cecilia da Silva Santos

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences
Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

To engineer secure software systems, software architects elicit the system's security requirements to adopt suitable architectural solutions. They often make use of architectural security tactics when designing the system's security architecture. Security tactics are reusable solutions to detect, resist, recover from, and react to attacks. Since security tactics are the building blocks of a security architecture, flaws in the adoption of these tactics, their incorrect implementation, or their deterioration during software maintenance activities can lead to vulnerabilities, which we refer to as "tactical vulnerabilities". Although security tactics and their correct adoption/implementation are crucial elements to achieve security, prior works have not investigated the architectural context of vulnerabilities. Therefore, this dissertation presents a research work whose major goals are: (i) to identify common types of tactical vulnerabilities, (ii) to investigate tactical vulnerabilities through in-depth empirical studies, and (iii) to develop a technique that detects tactical vulnerabilities caused by object deserialization. First, we introduce the Common Architectural Weakness Enumeration (CAWE), which is a catalog that enumerates 223 tactical vulnerability types. Second, we use this catalog to conduct an empirical study using vulnerability reports from large-scale open-source systems. Among our findings, we observe that "Improper Input Validation" was the most reoccurring vulnerability type. This tactical vulnerability type is caused by not properly implementing the "Validate Inputs" tactic. Although prior research focused

on devising automated (or semi-automated) techniques for detecting multiple instances of improper input validation (*e.g.*, SQL Injection and Cross-Site Scripting) one of them got neglected, which is the untrusted deserialization of objects. Unlike other input validation problems, object deserialization vulnerabilities exhibit a set of characteristics that are hard to handle for effective vulnerability detection. We currently lack a robust approach that can detect untrusted deserialization problems. Hence, this dissertation introduces DODO (untrusteD Object Deserialization detectOr), a novel program analysis technique to detect deserialization vulnerabilities. DODO encompasses a sound static analysis of the program to extract potentially vulnerable paths, an exploit generation engine, and a dynamic analysis engine to verify the existence of untrusted object deserialization. Our experiments showed that DODO can successfully infer possible vulnerabilities that could arise at runtime during object deserialization.

Acknowledgments

I am thankful to my advisor, Dr. Mehdi Mirakhorli, for the opportunity in pursuing the Ph.D., for his patience, endless support, and willingness to teach me crucial skills for my career. Dr. Mirakhorli countless times helped me to navigate through the rough paths along the way and I could not be more thankful for his supervision.

I also would like to thank the committee members of this Ph.D. for their insightful comments and suggestions: Dr. Sumita Mishra, Dr. Hossein Hojjat, and Dr. Andrew Meneely. Moreover, I am thankful to my collaborators, Dr. Katy Tarrit, Dr. Matthias Galster, Dr. Meiyappan Nagappan, as well as my lab mates and colleagues at RIT.

Last but not least, I am thankful for my family, relatives, and friends that helped me go through this journey (too many to list each of their names). Particularly, my deepest gratitude to my mother, Maria Clarice (*in memoriam*), for nurturing me, encouraging me in continuing my studies, taking care of me, being an example of a resilient person among many other things (that would take me pages to describe). She, certainly, played a crucial role for me to beat the odds and challenges caused by poverty in order to successfully pursue graduate education. I would have not made this far in life without her. She laid out the foundations for me to be the human I am today.

I am merely a "da Silva Santos" with modest beginnings. As such, I only made this far because of the vast support, encouragement, and guidance of many people in my academic journey. To all of them, including you, the reader, **a heartfelt "Thank You"**.

"If I have seen further than others, it is by standing upon the shoulders of giants."

Isaac Newton.

To the memory of my mother, Maria Clarice da Silva *(1953–2017), who established a foundation upon which I stand.*

To the memory of my aunties, Maria Clara da Silva *(1945–2016) and* Izabel Cristina Santos da Silva *(1956–2021), whose lives were an example to me.*

To the memory of loved ones lost to COVID-19 and its complications.

"What is grief, if not love persevering?"

Quote from Episode 8 of WandaVision TV show.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software vulnerabilities are caused by *defects* that makes the software system susceptible to being exploited by intruders [16]. These issues expose the system to attacks that can result in crashes (denial of service), confidentiality violation, and integrity compromises. To engineer a secure software, it is important to follow a *proactive* approach, in which security concerns are addressed early in the software development's lifecycle. The downside of fixing vulnerabilities after they are discovered during the software's usage (*reactive* approach) is that these breaches can be exploited by attackers, culminating in a compromise of the system's security. Hence, there will be potential costs and consequences associated with this exploitation, such as loss of revenue and brand reputation damage [144].

To engineer security software systems, software architects elicit the system's security requirements for adopting suitable architectural solutions. These architectural mechanisms that address security goals form the system's *security architecture* [27, 62, 126]. Software architects often make use of *architectural security tactics* when devising the system's security architecture [27]. Architectural security tactics (henceforth "security tactics") are reusable solutions to *detect*, *resist*, *react to*, and *recover from* attacks [27].

Since security tactics are the building blocks of a security architecture, flaws in the *adoption* of these architectural tactics [16], their incorrect *imple-*

mentation [98,123,125,126] or their *deterioration* during software maintenance activities [77] can lead to vulnerabilities. We refer to these security issues as ***tactical vulnerabilities***.

An example of tactical vulnerability is the use of client-side authentication. In this flaw, the "Authenticate Actors" tactic [27] is adopted at the client-side code instead of the server-side (*i.e.*, the authenticity check is performed at the client-side). As a consequence, an attacker could bypass the authentication feature by reverse engineering the client code and then removing the authenticity check. While this tactical vulnerability is caused by an improper *adoption* of the "Authenticate Actors" tactic, previous research observed that developers may implement security design decisions incorrectly [99, 123].

Although the software architecture is a crucial element onto achieving security [35], prior vulnerability studies have not investigated the architectural context of vulnerabilities, including design decisions such as security tactics and patterns [31, 66, 97]. They typically focus on vulnerabilities related to the management of data structures and variables (*e.g.*, buffer overflow/overread) [66]. Others have developed architecture analysis techniques to correlate design violations with software vulnerabilities [55]. 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.

To fill in this knowledge gap, one of the goals of this PhD research is to better understand how and why tactical vulnerabilities occur in real software systems. We accomplished this goal by conducting several empirical studies [123, 124, 126]. Our studies showed that *Improper Input Validation* was the most reoccurring tactical vulnerability type among the different types introduced by flawed security tactics implementations. This class of tactical vulnerability is caused by not properly implementing the "Validate Inputs" tactic [16]. As a result, inputs provided into the system interfere with the control/data flow of a program leading to injection of malicious commands, denial of service, and data leakage/corruption [32, 124].

Given the pervasive nature of input validation problems, several works have focused on developing automated (or semi-automated) techniques for detecting

these types of vulnerabilities [18,21,23,38,48,51,54,57,79,83,147–152,160,166]. These techniques take as input software artifacts (binaries and/or source code) and outputs a list of security violations (vulnerabilities) and their locations. These works perform *static analysis* [160,166], or *dynamic analysis* [48,83,159], or a hybrid analysis [23] to find vulnerabilities.

The literature has extensively analyzed specific instances of improper input validation, such as *SQL Injection*, *XML Injection*, *Cross-Site Request Forgery* (CSRF), *Cross-Site Scripting* (XSS), and *Server-Side Includes*. However, one of type of tactical vulnerability got neglected, which is the **untrusted deserialization of objects** [108, 128]. Unlike other vulnerability types rooted at improper input validation, *untrusted deserialization vulnerability* exhibits a set of challenges that are hard to handle for effective[1] vulnerability detection (these challenges are described in Section 6.1).

There is been an uptick of severe deserialization vulnerabilities reports widely discussed by security practitioners since 2015 [59], with the first instances occurring dating back 2006 [127]. Given the pervasive and severity nature of object deserialization vulnerabilities, this tactical vulnerability type is listed in rank #13 in the 2021 Common Weakness Enumeration (CWE™) Top 25 Most Dangerous Software Weaknesses (CWE Top 25) [145]. Although untrusted object deserialization is increasingly occurring in software systems, we currently lack a robust approach that can detect untrusted deserialization problems. Current exploitation tools [15,60] are limited in the sense that they leverage *prior* reports to verify whether the system is prone to deserialization attacks.

Therefore, this dissertation presents a research work that aims to achieve three major goals: (i) to **understand** what are the common types of tactical vulnerabilities; (ii) to **investigate** key aspects of tactical vulnerabilities through in-depth empirical studies; and (iii) to develop a technique to **detect** tactical vulnerabilities rooted at untrusted deserialization of objects in Java programs.

[1] By effective we mean an approach that balances precision, recall, and scalability while detecting untrusted object deserialization vulnerabilities.

1.1 Research Goals

Goal 1 ***Identify common types of tactical vulnerabilities.***

We aim to promote the awareness and of common security architecture weaknesses. To achieve this goal, we systematically enumerated types of architectural vulnerabilities known by the software security community. It ensued in the creation of the *Common Architectural Weakness Enumeration* (the CAWE catalog). Chapter 4 discusses the creation process for this catalog. The CAWE catalog is important for creating a knowledge base that could later be used for detecting such issues. Using the CAWE catalog, we answered two research questions.

Research Questions

RQ1 *What are the types of tactical vulnerabilities?*

RQ2 *What architectural security tactics are more likely to have associated vulnerabilities?*

Goal 2 ***Understand tactical vulnerabilities***.

Tactical vulnerabilities have multiple characteristics, such as their consequences, root causes and mitigation techniques. Therefore, a second goal of this research work is to better understand the previously identified tactical vulnerability types through empirical studies. We conducted an empirical investigation of tactical vulnerabilities across multiple open-source systems to accomplish this goal. This study encompassed the extraction of vulnerability reports and related artifacts for three large-scale software systems, namely Chromium, PHP, and Thunderbird. By using the CAWE catalog and performing a detailed analysis of vulnerability artifacts, we identify the most occurring tactical vulnerability types on these projects, the security tactics mostly affected by these tactical vulnerabilities and their root causes and fixes. This empirical study is then used to answer three research questions.

Research Questions

RQ3 *What are the most common tactical vulnerabilities in real software systems?*

RQ4 *What security tactics are most affected by tactical vulnerabilities?*

RQ5 *What are the root causes of the most frequently occurring types of tactical vulnerabilities?*

Goal 3 ***Detect tactical vulnerabilities rooted at untrusted deserialization***

As it will be discussed in Chapters 2 and 3, existing techniques fall short in uncovering untrusted deserialization vulnerabilities due to unsoundness of their analysis with respect to serialization/deserialization features. As such, the last goal is to develop DODO (untrusteD Object Deserialization detectOr) a hybrid program analysis technique to detect object deserialization vulnerabilities. It encompasses the development of novel (i) *taint-based* and *downcast-based* call graph construction algorithms for soundly handling object deserialization in a program, an (ii) *exploit generation*, and (iii) a *dynamic analysis* engine. While evaluating DODO, we answer two research questions.

Research Questions

RQ6 *Does* DODO*'s call graph algorithm handle object deserialization soundly?*

RQ7 *Is* DODO *useful for finding object deserialization vulnerabilities?*

1.2 Thesis Statement

Understanding and identifying tactical vulnerabilities can help the upfront design of a security architecture and aid the correct implementation of design decisions in code.

1.3 Contributions

The **contributions** of this Ph.D. work are:

- A catalog of common architectural weaknesses (the CAWE catalog) to promote the awareness of architectural issues that result in vulnerabilities [125].
- A series of empirical studies on tactical vulnerabilities in real software systems to understand the characteristics and pervasiveness of tactical vulnerabilities [123, 124, 126].
- The development of an automated technique for detecting untrusted deserialization vulnerabilities [121, 122].

1.4 Publications

This Ph.D. research resulted in the following publications:

SOAP'21 **J. C. S. Santos**, R. A. Jones, M. Mirakhorli, and C. Ashiogwu. "Serialization-aware Call Graph Construction". In: *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP'21).*

ICSA'21 A. Shokri, **J. C. S. Santos**, and M. Mirakhorli. "ArCode: Facilitating the use of application frameworks to implement tactics and patterns". In: *2021 IEEE 18th International Conference on Software Architecture (ICSA'21).*

SCAM'20 S. Moshtari, **J. C. S. Santos**, M. Mirakhorli, and A. Okutan. "Looking for Software Defects? First Find the Nonconformists - An Outlier-Based Defect Prediction Approach". In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM'20).*

WoSoCer'20 **J. C. S. Santos**, A. Shokri, and M. Mirakhorli. "Towards Automated Evidence Generation for Rapid and Continuous Software Certification". In *10th IEEE International Workshop on Software Certification (WoSoCer'20).*

FTfJP'20 **J. C. S. Santos**, R. A. Jones, and M. Mirakhorli. "Salsa: Static Analysis of Serialization Features". In *22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs (FTfJP'20).*

ICSEW'20 **J. C. S. Santos**, S. Suloglu, J. Ye, and M. Mirakhorli. "Towards an Automated Approach for Detecting Architectural Weaknesses in Critical Systems". In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20) – 1st International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS).*

ICSA-C'20 **J. C. S. Santos**, S. Moshtari, and M. Mirakhorli. "An Automated Approach to Recover the Use-case View of an Architecture". In *Proceedings of the 2020 IEEE International Conference on Software Architecture Companion (ICSA-C) - New and Emerging Ideas (NEMI) Track.*.

ESEC/FSE'19 **J. C. S. Santos**, A. Sejfia, T. Corrello, S. Gadenkanahalli, and M. Mirakhorli. "Achilles' heel of plug-and-Play software architectures: a grounded theory based approach". In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 671-682).

JSS'19 **J. C. S. Santos**, K. Tarrit, A. Sejfia, M. Mirakhorli, and M. Galster. "An Empirical Study of Tactical Vulnerabilities". *Journal of Systems and Software.* 149, 263-284.

ICSA'17 **J. C. S. Santos**, A. Peruma, M. Mirakhorli, M. Galster, J. V. Vidal and A. Sejfia. "Understanding Software Vulnerabilities Related to Architectural Security Tactics: An Empirical Investigation of Chromium, PHP and Thunderbird" In: *Proceedings of the 2017 IEEE International Conference on Software Architecture* (ICSA) (pp. 69-78). **Best Paper Award**

ICSAW'17 **J. C. S. Santos**, K. Tarrit, and M. Mirakhorli. "A Catalog of Security Architecture Weaknesses". In *Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 220-223).

1.5 Dissertation Structure

The remainder of this dissertation is organized as follows: Chapter 2 introduces terminology relevant to understand this research. Chapter 3 considers related work in the area of software security, software architecture design, and program analysis. Chapters 4 to 6 describes the methodology followed to create the CAWE catalog, conduct the empirical studies and develop Dodo in order to answer our research questions. Chapter 7 presents and discusses current results. Chapter 8 concludes this dissertation with a summary of current achievements along with ideas for extending the work.

Chapter 2

Background

This chapter defines the main concepts which are needed to ensure that this work can be understood by a broader audience.

2.1 Vulnerabilities and Vulnerability Databases

Software vulnerabilities are defects that affect a system's intended security properties. These security problems are typically disclosed and discussed across online forums, and many other websites, as well as tracked by vulnerability databases. A well-known vulnerability database is the **National Vulnerability Database (NVD)**, which currently tracks over 167,000 vulnerabilities that exist in a variety of software products, both open and closed source. Vulnerabilities disclosed in NVD are assigned a unique identifier known as "**CVE ID**" (Common Vulnerabilities and Exposure Identifier). Besides this identifier, NVD also includes details about the security issue. An instance of vulnerability recorded in NVD is shown below:

> **CVE ID**: CVE-2020-6988
> **Overview**: A remote, unauthenticated attacker can send a request from the RSLogix 500 software to the victim's MicroLogix controller. The controller will then respond to the client with used password values to authenticate the user on the client-side. This method of authentication may allow an attacker to bypass authentication altogether,

disclose sensitive information, or leak credentials.
References:
• `https://www.us-cert.gov/ics/advisories/icsa-20-070-06`
Affected Software Configurations:
Rockwell Automation's:
• MicroLogix 1400 Controllers Series B (v21.001 and prior) and Series A (all versions).
• MicroLogix 1100 Controller (all versions).
• RSLogix 500 Software (v12.001 and prior).
Vulnerability Type (s)
• CWE-603: Use of Client-Side Authentication (source: ICS-CERT)
• CWE-287: Improper Authentication (source: NIST)

As demonstrated in this excerpt, each entry in NVD includes a short *description* of the problem as well as a list of *references* that are links to other Web sites (such as issue tracking systems) that may contain more details about the CVE instance. NVD also indicates the software's *releases* affected by the vulnerability. In this example, multiple firmware versions developed by this vendor were affected.

Some of the CVE instances may also include *CWE tags* that indicate the *vulnerability type*. These tags are assigned by security analysts from the entities that reviewed the vulnerability report. The CWE tag refers to an entry from the **Common Weakness Enumeration (CWE)** dictionary [41], which enumerates common weaknesses in a software system that may lead to vulnerabilities. A *weakness* 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*) [101]. Thus, the CWE tag is used by the NVD as a way to classify vulnerabilities.

It is important to highlight the distinction between a **weakness** and a **vulnerability**. A **weakness** (or **vulnerability type**) is a security class of problems in a software system that share common characteristics (root cause, consequence and mitigations). In contrast, a **vulnerability** is an *instance* of a weakness (an actual occurrence of the weakness).

2.2 Architectural Security Tactics

Architectural Security Tactics are means of achieving security properties through a series of inter-related design decisions [19]. They are the building blocks of a security architecture [62] and provide reusable solutions for satisfying security requirements, even when the system is under attack. A comprehensive list of security tactics has been provided by Bass *et al.* [27]. These tactics are classified into four categories, as shown in Table 2.1.

2.3 Weaknesses and Tactical Vulnerabilities

Since security tactics are the building blocks of a security architecture, mistakes in the adoption or implementation of these tactics can result in **weaknesses** in the security architecture [125]. We can classify these "tactical weaknesses" (*i.e.*, **tactical vulnerability types**) into three categories [126]:

- **Omission weaknesses** are caused by *not adopting* a proper security tactic to address a security requirement. An example of an omission weakness is to store credentials *without encryption*. The lack of the "Encrypt Data" tactic in this scenario allows attackers to steal sensitive data, 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

Table 2.1: Architectural 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

and implementation issues related to architectural tactics. More specifically, these vulnerabilities occur due to:

(i) a lack of security tactics (***omission weakness***) in the application's architecture; or

(ii) adoption of less suitable security tactics for a given design problem or context (***commission weakness***); or

(iii) an incorrect implementation of security tactic principles which results in an incorrect transition from design to code (***realization weakness***).

2.4 Object Serialization and Deserialization

Object serialization (also known as "*marshaling*") is a mechanism in which an object is converted to an ***abstract representation*** (e.g., bytes, XML, JSON, etc). Such abstract representations are suitable for network transportation, storage, and inter-process communication. The receiver of a serialized object has to parse the abstract representation in order to reconstruct a new object. This reconstruction process is called **object deserialization** (or "*unmarshalling*").

Although object serialization and deserialization seems a innocuous mechanism, it can introduce serious vulnerabilities [95]. The problem stems from allowing arbitrary object types to be deserialized and invoking methods from the objects' classes during their reconstruction. Deserialization libraries may invoke default constructors, getter/setter methods, or methods with specific signatures when reconstructing the object. These are the **callback methods** of the deserialization mechanisms. As a result, attackers could leverage these callback methods invoked during object deserialization to perform a malicious action, such as resource consumption (denial-of-service attacks), application crashes and remote code execution [49, 108].

Each programming language have their own serialization/deserialization protocol. The Java's default serialization and deserialization protocol is thoroughly described at their specification page [106]. We briefly present this mechanism in the next subsection.

2.4.1 Java Serialization API

The default Java's Serialization API converts a snapshot of an object graph into a *byte stream*. During this process only *data* is serialized (*i.e.*, the object's fields). The code associated with the object's class (*i.e.*, methods) is within the classpath of the receiver of this serialized object [128]. For an object to be serializable, its class has to implement the `java.io.Serializable` interface. The specific class fields that should be serialized can be defined in two ways:

- *Implicitly*: all the *non-transient* and *non-static* fields of the class are serializable fields by default.
- *Explicitly*: all the fields that are listed during the initialization of a special field (`serialPersistentFields`) in the class. This particular field (which has to be declared as a private static final field) is initialized with an array of `ObjectStreamField` objects that indicate the names and types of the attributes that should be serialized.

The example in Listing 1 shows two classes with implicit and explicit declarations. The `User` class explicitly indicates that only one of its fields (`name`) must be serialized. For the `Book` class, all of its non-static fields (*i.e.*, `totalPages` and `title`) are serialized.

```
1  import java.io.ObjectStreamField;
2  import java.io.Serializable;
3
4  public class User implements Serializable {
5    private String name; private int id;
6    private static final ObjectStreamField[] serialPersistentFields = { new ObjectStreamField("name", String.class) };
7  }
8
9  public class Book implements Serializable{
10   private int totalPages; private String title;
11 }
```

Listing 1: Explicit and implicit declaration of serializable fields

The classes `ObjectOutputStream` and `ObjectInputStream` can be used to serialize and deserialize an object, respectively. They can only serialize/deserialize objects whose class implements the `java.io.Serializable` interface. If implemented by `Serializable` classes, the following methods are invoked by Java during deserialization **(1-4)** or serialization **(5-6)**:

(1) `void readObject(ObjectInputStream)`: it customizes the retrieval of an object's state from the stream.

(2) `void readObjectNoData()`: in the exceptional situation that a receiver has a subclass in its classpath but not its superclass, this method is invoked to initialize the object's state.

(3) `Object readResolve()`: this is the inverse of `writeResolve`. It allows classes to replace a specific instance that is being read from the stream.

(4) `void validateObject()`: it validates an object after it is deserialized. For this callback to be invoked, the class has to implement the `ObjectInputValidation` interface and register the validator by invoking the method `registerValidation` from `ObjectInputStream` class.

(5) `void writeObject(ObjectOutputStream)`: it customizes the serialization of the object's state.

(6) `Object writeReplace()`: this method replaces the actual object that will be written in the stream.

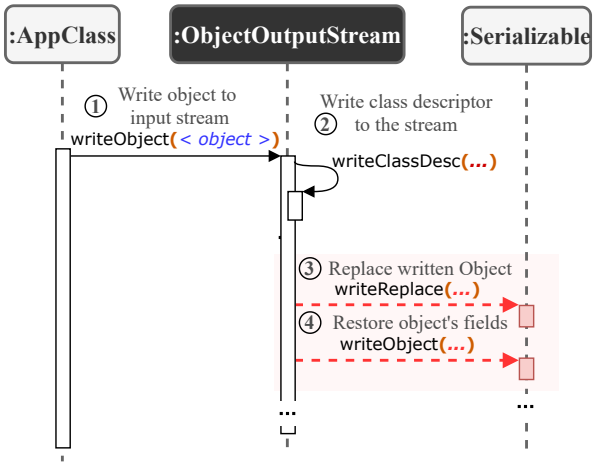


Figure 2.1: UML sequence diagram for Java's serialization

Figures 2.1 and 2.2 depicts the sequence of these callback methods invocations, highlighted in red boxes. Classes with a dark background are part of the Java's API, whereas the classes with a light gray background are application classes.

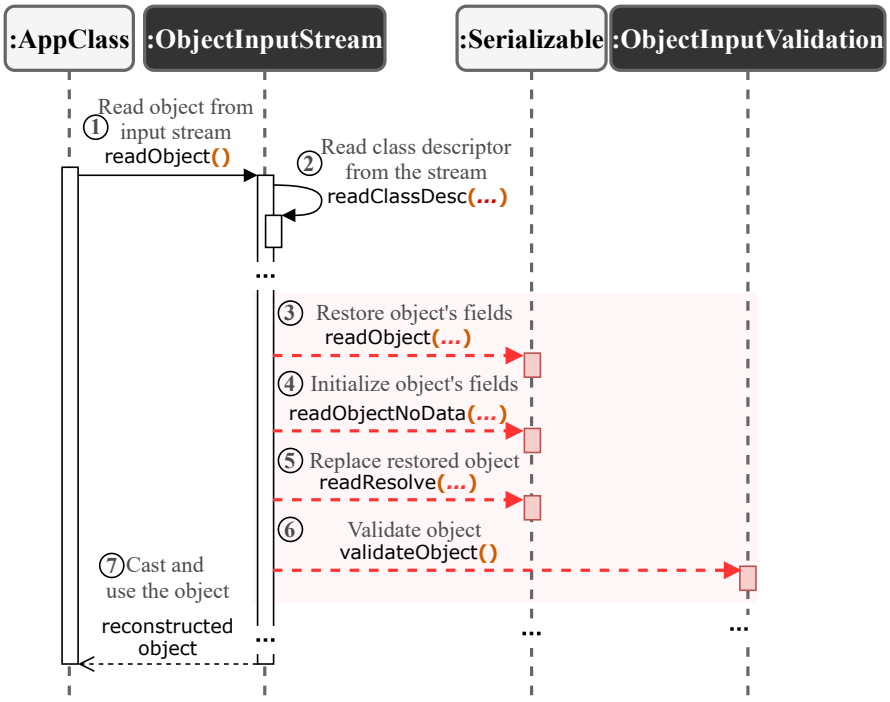


Figure 2.2: UML sequence diagram for Java's deserialization

2.4.2 Untrusted Deserialization Vulnerabilities

Untrusted object deserialization is listed as **CWE-502** in our CAWE catalog [125]. This tactical vulnerability type is sometime discussed in the literature as "object injection vulnerabilities" [131, 135]. In this section, we describe how this tactical vulnerability occur by walking through a vulnerable example program.

As shown in Figure 2.2, the `ObjectInputStream` class invokes callback methods from the Serializable classes as the stream is parsed and class descriptors are read. These callback methods are commonly referred as ***magic methods*** [128] and can be used in an exploit (details in Section 2.4.2).

Vulnerabilities can occur during or after deserialization:

- During: dangerous operations are *within* (or *reachable from*) magic methods.
- After: the object is later used by the application, which invokes its common methods (e.g. `.toString()`, `.finalize()`, etc) and they contain (or reach to) dangerous code. In this case, the object shall be of expected type (otherwise, a `ClassCastException` is thrown, which means the object will not be used any further).

The deserialization vulnerability is not specific to the `java.io.Serializable` interface, but it can also work for `java.io.Externalizable`'s interface (through the magic method `readExternal()`). In this dissertation, we are focusing on vulnerabilities achieved via the ***Serializable*** interface ***during*** deserialization. However, extending it can be viable by adding rules that abstract the inner workings of the Externalizable interface.

To illustrate how a seemingly harmless mechanism can lead to serious vulnerabilities, consider the example illustrated in Listing 2. It showcases a Web application that reads a `User` object stored in a cookie named as "user". On line 5, the application retrieves the "user" cookie from the HTTP request. This cookie is expected to contain a serialized `User` object encoded using *Base64*. On line 8, the application reconstructs the received `User` object from this base64-encoded stream of bytes stored in the cookie.

An attacker could leverage the deserialization process to remote code execute as follows. Consider that the application has in its classpath two serializ-

```
1  public class IndexServlet extends HttpServlet {
2    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
3      Cookie cookie = getCookieByName(req, "user");
4      if (cookie != null) {
5        byte[] bytes = Base64.getDecoder().decode(cookie.getValue());
6        ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
7        ObjectInput in = new ObjectInputStream(bis);
8        User u = (User) in.readObject();
9        /* ... */
10     } else { /* ... */ }
11   } //...
12 }
```

Listing 2: A servlet that reads a serialized object from a cookie.

able classes: `CacheManager` and `CommandTask` (as shown in Listing 3). An attacker would create a `CacheManager` object (`cm`) as shown in Figure 2.3. Then, the attacker serializes and encodes this malicious object (`cm`) in base64 and sends it as a cookie to the Web application. When the web application receives that cookie, it triggers the chain of method calls depicted in Figure 2.3. This sequence of method calls ends in an execution sink (`Runtime.getRuntime.exec()` on line 8 of the `CommandTask` class in Listing 3).

```
1  public class CacheManager implements Serializable {
2    private Runnable initHook;
3    public CacheManager(Runnable initHook) {
4      this.initHook = initHook;
5    }
6    private void readObject(ObjectInputStream ois) {
7      ois.defaultReadObject(); // populate initHook
8      initHook.run();
9    }
10 }
```

```
1  public class CommandTask implements Runnable,
2      Serializable {
3    private String command;
4    public CommandTask(String command) {
5      this.command = command;
6    }
7    public void run() {
8      Runtime.getRuntime().exec(command);
9    }
10 }
```

Listing 3: "Gadget classes" that can be used to trigger a remote code execution.

Although this request with a malicious serialized object results in a `ClassCastException` (when the execution returns to line 8 in Listing 2), the malicious command will be executed anyway, because the type cast check occurs ***after*** the deserialization process took place.

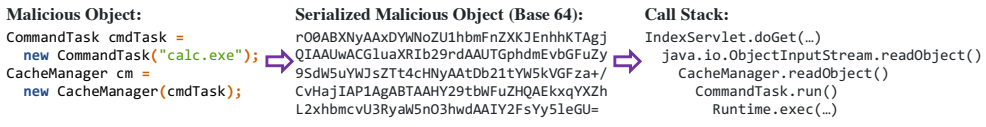


Figure 2.3: Malicious serialized object used to trigger a remote code execution.

As we can see from this vulnerability example, objects from serializable

classes can be specially combined to create a chain of method calls. These classes are called as "gadget" classes. They are used to bootstrap a chain of method calls that will end in an execution sink.

Thus, to exploit an untrusted deserialization vulnerability the conditions are: (a) the application performs serialization; (b) there are serializable classes in the classpath implementing custom deserialization routines (e.g. `readObject()`); (c) these magic methods perform dangerous tasks. (d) these gadget classes are in the scope of the application (available to be invoked). These conditions are *necessary*, but *not sufficient*. Uncovering untrusted deserialization, therefore, involve finding gadget classes in the classpath. These gadget classes can be from external APIs/libraries, the application itself, or even Java's standard classes.

2.5 Program Analysis Techniques

Program analysis techniques reason over multiple program elements (e.g, code, execution traces, inputs, test cases, architectural descriptions, etc.) to verify certain *properties* [104]. **Properties** are facts about the program that holds true for one or more executions (or even all of them). An example of a program property is that all type casts in a program are safe (for all executions).

There are two major techniques for analyzing program properties: *dynamic* and *static* analysis. On one hand, **dynamic analysis** is the process of reasoning over properties as the program is *executed* [22]. On the other hand, **static analysis** performs property verification by solely inspecting the program's source code, without actually executing it [36]. As a consequence, static analysis can reason about properties over multiple possible execution paths.

Property verification, however, is an **undecidable problem** [86], *i.e.*, it is not guaranteed that the verification algorithm terminates with an answer. As stated in Rice's theorem [116], it is unfeasible to create an automated approach that can verify whether any non-trivial property holds for a program[1]. Despite the undecidability of property verification, static analysis can

[1] A property ϕ is non-trivial if there is a version of the program P in which the property holds, and there is another part of P that ϕ does not.

make the problem algorithmically decidable by creating **abstractions** of the program's semantics and then verifying whether the property holds for this abstraction [42]. The underlying abstraction will, therefore, dictate whether a static analysis framework is *sound* or *complete*, which are defined as follows [22, 104]:

- A ***sound*** static analysis creates an **over-approximation** of the program's behavior. Hence, if a sound static analyzer claims that a program is error-free (*i.e.*, it does not violate a property ϕ) then this is guaranteed. However, the existence of a violation to ϕ is not guaranteed to be true (*i.e.*, false positives).
- A ***complete*** static analyzer yields to an **under-approximation** of the program's behavior. As a consequence, any detected violation to a property ϕ is a true positive. However, this type of analyzer is not guaranteed to find all violations to ϕ.

Soundness of a static analysis is a coveted property, but hard to achieve in practice. As a consequence, certain static analyses favor precision over soundness. These approaches are referred as **"soundy" analyses** described as *"[an] analysis* [that] *aims to be as sound as possible without excessively compromising precision and/or scalability"* [92].

The main advantage of static analysis is that it can provide high coverage. However, given the challenge of determining the semantics of programs (abstraction), this class of analyses yields to spurious results. They are also prone to miss execution paths that are not handled by the underlying abstraction. For instance, dynamic programming features (such as reflection) is a wreak havoc on static program analysis approaches because the actual behavior can only be determined at runtime [30, 87, 91, 138]. For this reason, some analyses simply ignore these constructs or create unsound abstractions [92, 140].

2.5.1 Taint Analysis

Taint analysis is a program analysis technique that verifies information flows from **sources** to **sinks** [155]. Since *sources* provide untrusted data (*i.e.*, **tainted data**) into the program, their data should not flow into sinks, which expect trusted (untainted) data. Hence, a taint analysis technique tracks

sources of tainted data, and how they are propagated across the program.

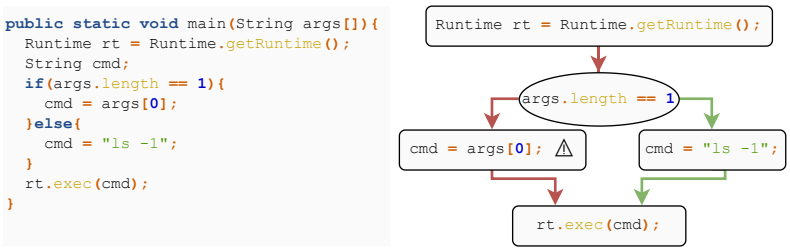


Figure 2.4: Example of a vulnerable program

A taint analysis technique relies on a **taint policy** which specifies how taint is *introduced* and *propagated* throughout operations. The adopted policy is tailored to the underlying client analysis, such as vulnerability detection, test case generation, etc. A taint analysis policy can also include *sanitizers* [149, 151] which are operations that remove taint from a variable. After analyzing the program, and propagating/removing the taint according to the underlying policy, a taint analysis technique reports a program as **vulnerable** if there is **an information flow from a source to a sink** [129].

For example, Figure 2.4 shows an a vulnerable Java program in which the vulnerability could be detected using taint analysis. Since the `args` is an input that can be tampered with, the taint policy marks it as *tainted*. Moreover, given that `rt.exec(...)` is a method that executes any given command, it is a security-sensitive operation marked as a *sink*. Since there is a flow from a tainted variable to the sink (highlighted in red), the analysis report the program as vulnerable.

Chapter 3

Related Work

Existing research have acknowledged the importance of developing secure software systems and have proposed many methods and approaches for achieving security by design. This chapter discusses relevant works in the field.

3.1 Software Architecture and Security

Existing research on software architecture for security has proposed methods for facilitating the analysis and evaluation of a security architecture [68, 119, 143] and techniques for reverse engineering security design decisions from source code [33]. Although these works can help architects to identify threats and to appropriately adopt security tactics into a system, such activities may not be enough to avoid vulnerabilities because the implementation of design decisions may be incorrect or erode over the time.

Similar to this PhD work, previous research focused on identifying potential threats and vulnerabilities from the underlying architecture [13, 29, 137, 164]. However, they currently can detect very specific instances of architectural flaws related to race conditions [13], anomalous component interactions [164] and multi-tier business applications implemented in Java [29].

3.2 Empirical Studies on Software Vulnerabilities

Current empirical studies analyze vulnerabilities for a diverse set of goals, such as understanding when vulnerabilities were first introduced in the software project [97], verifying what are the impacts of a public disclosure of vulnerabilities [17,144] as well as investigating how these vulnerabilities are discovered, reported and the associated delays when fixing them [31, 32, 66, 165].

Regarding the works that empirically investigated the relationship between vulnerabilities with respect to its architecture, Feng *et al.* [55] analyzed how architectural structure violations (e.g. improper inheritance) are related to vulnerabilities. Their findings suggested that files involved in structural violations are correlated with occurrences of vulnerabilities and have a higher code churn and a greater number of changes for fixing their vulnerabilities.

3.3 Vulnerability Prediction

The works in this category are concerned with creating prediction models from historical vulnerability data to help the identification of the parts of the system that are more likely to be prone to vulnerabilities. In this context, existing works have been using a variety of metrics derived from software artifacts (such as code churn, lines of code and imported libraries) or organizational metrics (e.g., the number of engineers) to build prediction models [37,103,132,136,167].

Researchers are not only working on developing vulnerability prediction models, but also in investigating more fundamental questions. For instance, Hovsepyan *et al.* [74] verified to which extent prediction models benefit from using older releases or newer releases for training the prediction algorithm. Through investigating vulnerabilities from two real systems, they observed that using older releases to create the prediction models increases the recall of these predictions, at the cost of high false positive rates. Walden *et al.* [158] compared different prediction models created using traditional software metrics as features against a model that was based on text features (*i.e.*, keywords extracted from source files). Through a set of cross-validation experiments, they found that using text features had a higher recall than using software metrics as features.

3.4 Formal Methods for Security Architecture

Formal methods are techniques that mathematically models a complex system, which allows reasoning over its properties in a more rigorous fashion [78]. Given that formal methods can model systems, regardless of their domain and technologies, existing works used them for helping the design of secure systems.

In this context, Kang *et al.* [81] developed an approach for facilitating security analysis. It allowed software architects to explore and evaluate alternative design solutions and verify potential security implications of each of these alternative design decisions. Through a study conducted over two applications [81], they were able to uncover two vulnerabilities in the design in each of these two systems, which were later confirmed by their software vendors. Similarly, Gacek *et al.* [61] proposed a framework that automatically generates assurance cases[1] from a system model specified in an Architecture Analysis and Design Language (AADL) and a set of rules expressed in a domain specific language proposed by the authors. Furthermore, Heyman *et al.* [71] presented reusable formal models of security patterns[2]. These proposed models were devised to be reused by architects to formally describe a security architecture and uncover potential architectural security flaws.

3.5 Automated Vulnerability Detection

Current state-of-the-art techniques that automatically detects vulnerabilities leverage on two approaches: *static analysis* and *dynamic analysis*. On one hand, static analysis techniques inspects only the source code against a set of predefined rules to find vulnerabilities (*i.e.*, it finds security violations without running the code). On the other hand, dynamic analysis approaches finds vulnerabilities by executing the code and observing/monitoring its behavior. Currently, there are a variety of open source and commercial tools for performing automated vulnerability detection. Examples of such tools are Flaw Finder, FindBugs and Splint (open source) as well as Fortify 360, AppScan and Coverity (commercial) [105].

[1] Assurance cases are claims and supporting evidence about the security of a given system.

[2] A Security pattern is a documented solution for a common security problem that happens in a specific context.

In this scenario, previous research has focused on developing new techniques that leverage on static [160, 166], dynamic analysis [48, 83, 159], or both approaches [23] to detect security issues. Since these tools and techniques are different in terms of the vulnerability types they can detect and the programming language(s) they support, numerous studies have compared different tools in terms of advantages/disadvantages and the quality of their vulnerability detection [18, 45, 57, 58, 85, 89, 109].

3.5.1 Untrusted Object Deserialization Detection

Many works explored the problem of performing pointer analysis of programs [28, 56, 70, 72, 82, 88, 117, 139]. These approaches focus on computing over- or under-approximations in order to improve one or more aspects of the static analysis, such as its soundness, precision, performance, and scalability. In this dissertation, however, we focus on aiding points-to analyses to soundly handle serialization-related features in a program, which are currently not well-supported because the deserialization API heavily relies on non-trivial reflection [87, 114].

Prior works on static analysis also explored the challenges involving supporting reflection features [30, 90, 91, 138]. These approaches involve making certain assumptions when performing the analysis, in order to create analyses that are not overly imprecise. Sharp and Rountev [134] discussed an approach to statically analyze RMI-based programs, which is a mechanism that also relies reflection but has a set of unique challenges. Analyzing RMI-based programs require reasoning over client and server code and their inter-process communication via objects/messages. Thus, the work presented an abstraction to soundly infer data flows across processes in RMI-based programs [133].

In the past few years, there was a spike of vulnerabilities associated with deserialization of objects [39, 73]. Thus, existing works also studied vulnerabilities rooted at untrusted deserialization vulnerabilities [49, 108]. Pele *et al.* [108] conducted an empirical investigation of deserialization of pointers that lead to vulnerabilities in Android applications and SDKs. Dietrich *et al.* [49] demonstrated how seemingly innocuous objects trigger vulnerabilities when deserialized, leading to denial of service attacks.

There is a line of research that explored call graph's soundness of Java

(or JVM-like) programs [14, 113, 114]. In particular, recent empirical studies [113,114] show that although serialization-related features are widely used, they are not well-supported in existing approaches. Currently, to the best of our knowledge, we could not find an approach that aims to enhance existing points-to analysis to support serialization-related features to enable the detection of untrusted object deserialization statically.

More recently there were approaches published that aimed at detecting untrusted object deserialization for PHP [84, 131] and .NET [135]. Shcherbakov and Balliu [135] described an approach to semi-automatically detect and exploit object injection vulnerabilities .NET applications. It relies on existing publicly available gadgets to perform the detection and exploitation. Koutroumpouchos *et al.* described ObjectMap [84] which is tool that performs black-box analysis of Web applications to pinpoint potential insecure deserialization vulnerabilities. It works by inserting payloads into the parameters of HTTP GET/POST requests and then monitoring the target web application for errors to infer whether the application is vulnerable or not.

More similar to our research, we can mention two recent works (that were published while this work was already being conducted) by Rasheed and Dietrich [111] and Haken [67]. Both works focused on deserialization vulnerabilities in Java programs [67, 111].

Rasheed and Dietrich [111] described a hybrid approach that first performs a static analysis of a Java program to find potential call chains that can lead to sinks, where reflective method calls are made. It then uses the results of the static analysis to perform fuzzing in order to generate malicious objects. Unlike our work (Dodo), their work is inherently unsound [111]. Dodo first makes the call graph sound with respect to serialization features, then it relies on symbolic execution and constraint solving to generate exploits.

Gadged Inspector [67] is a tool that analysis the program using CHA-based call graph construction algorithm. It outputs a list of gadget chains (*i.e.*, a sequence of method invocations from a deserialization callback method to a sink). Unlike Dodo, this approach only outputs *gadget chains* whereas our work goes further by *generating exploits* and using these exploits to *instrument the program* to verify whether the vulnerability exists or not.

Chapter 4

Identifying Tactical Vulnerability Types

This chapter discusses the creation of the *Common Architectural Weakness Enumeration* (CAWE catalog). This catalog documents and categorize known weaknesses in design or implementation of security tactics.

4.1 Creating the CAWE Catalog

We follow the following steps to create the CAWE catalog:

1. We compiled an extensive list of architectural security tactics previously published [27,96]. For each tactic, we collected its *description* and *keywords* that summarize the security tactic. It resulted in a list of 18 security tactics (see Table 2.1) which helped us to identify the common building blocks of security architectures.
2. We extracted all entries from the MITRE's CWE dictionary (version 2.9) [41]. Entries in the CWE dictionary can be of four types:
   - *View*: it groups weaknesses from a given perspective (*e.g.*, types of errors);
   - *Category*: it aggregates entries based on a common attribute. For exam-

ple, a shared environment (J2EE, .NET), a functional area (authentication, cryptography), *etc.*;

- *Weakness*: it is an actual vulnerability type;
- *Compound Element*: it defines a sequence of weaknesses that can lead to security issues (*i.e.*, it is a chain of weaknesses that results in another weakness).

Based on the descriptions above, we have not included entries that were a *View* or a *Category* because they group other weaknesses rather than representing one vulnerability type. Thus, we analyzed a total of **727** entries (*Weakness* or *Compound Elements*) out of the **1,004** entries available in the CWE dictionary version 2.9. A *Weakness* or *Compound Element* contains a variety of information such as a *description*, *mitigation techniques*, *common consequences*, *code examples*, *etc.* [1].

3. We performed a keyword-based search over the 727 entries' metadata. We used the tactics' keywords collected in the first step. The output of this is a set of *tuples*; each pair indicates a *potential* connection between a security tactic and a CWE entry.

4. The keyword-based search above gives an initial set for us to build upon. However, this automated process can render to inaccurate mappings. Hence, we manually analyzed all the 727 CWEs to confirm whether these potential connections indeed existed. We also checked these 727 CWEs for missing connections between a tactic and a weakness/compound element. The systematic process for this manual analysis was as follows:

   - 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).
   - we applied the criteria defined in Section 2.3 to consider a CWE entry to be a *architectural weakness*. If the CWE entry is either caused by (i) a lack of a design decision (*omission*); or (ii) an incorrect choice of security

[1] 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`

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*), the entry would be considered as an *architectural 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 an *architectural weakness*. We tagged each of these weaknesses with the *security tactic* affected by the weakness and the *type of impact* (*i.e.*, , commission, omission or realization weakness).

Example: Consider as an example the CWE-354 ("Improper Validation of Integrity Check Value"). It contains some keywords related to the security tactic "Verify Message Integrity". Hence, after performing the keyword-based search (3rd step), this entry is marked as *potentially* related to the "Verify Message Integrity" tactic. When we subsequently manually inspected it, we observed that this weakness is caused by an incorrect verification of the checksums [2] of messages. As a consequence, the software system may accept corrupted or intentionally modified messages. When we inspected 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. Therefore, we then considered the CWE-354 to be a "realization weakness" affecting the "Verify Message Integrity" tactic because it occurs due to an incorrect implementation of the tactic.

Minimizing Inconsistencies and Biases

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 Crit-

[2] Checksums are extra data that is attached to messages to detect errors and modifications in the message.

ical 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. These four individuals agreed on the categorization of 88.4% CWEs. For the entries with disagreements (84 CWEs in total), they discussed their rationale and reached a consensus of what would be the appropriate classification.

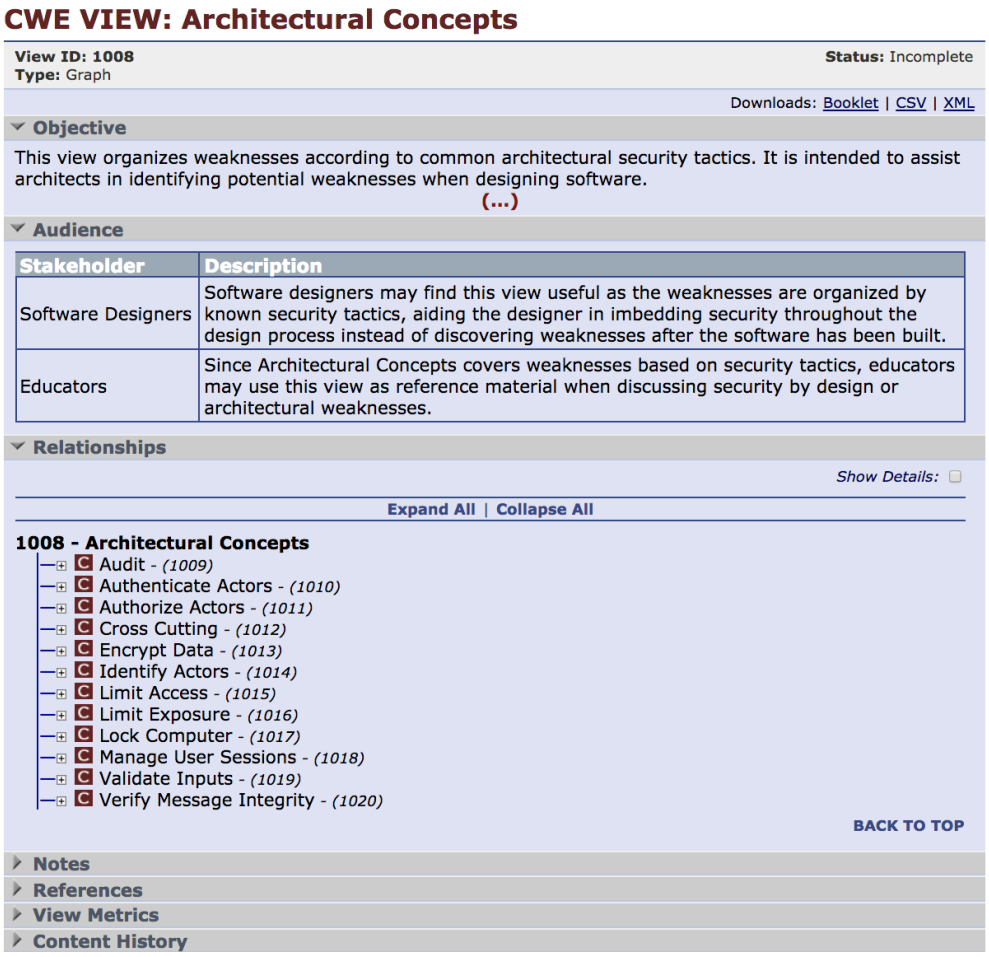
4.2 Overview of the CAWE Catalog

The CAWE catalog is currently integrated intro MITRE's list of software weaknesses as a *View* (CWE-1008 in Figure 4.1). This view has an ID equals to *1008*, and it is named as "Architectural Concepts". It can be accessed by following this link: `https://cwe.mitre.org/data/definitions/1008.html`. This URL brings to a Web page containing the list of affected security tactics (collapsed). It then displays the associated architectural weaknesses when each of these tactics are expanded.

The CAWE catalog currently has 223 architectural weaknesses categorized based on 11 security tactics. The CAWE catalog also contains a category called "Cross-Cutting", which encompasses weaknesses that can impact multiple security tactics (see category #1012 in Figure 4.1). An example of an architectural weakness is presented in Figure 4.2. 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).

One aspect worth mentioning is that MITRE's Website had a view that encompasses "mistakes made during the design and/or architecture phase"[3]. Unlike the CWE-701, our definition and purposes for the CAWE view are slightly broader. Our goal was to promote the awareness of oversights related to the security architecture itself (as an artifact). Therefore, weaknesses are then either omission/commission (that occur during the design process) or realization (that occur during the transition of a correct architecture to code).

[3]"CWE-701: Weaknesses Introduced During Design": http://cwe.mitre.org/data/definitions/701.html



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

Show Details:

Expand All | Collapse All

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)*

BACK TO TOP

Notes
References
View Metrics
Content History

Figure 4.1: The CAWE catalog integrated into MITRE's website as a view

Figure 4.2: CWE-291 "Reliance on IP Address for Authentication" with the added metadata from our work (impact type and affected tactic)

Chapter 5

Understanding Tactical Vulnerabilities

We used the CAWE catalog to study vulnerabilities in three large-scale open-source projects, namely Chromium, PHP, and Thunderbird. In this empirical study, we identified the most occurring architectural vulnerability types on these projects. Moreover, we conducted a qualitative analysis of each architectural vulnerability and their fixes. Through this qualitative analysis, we characterized their root causes and investigated the way the original developers of each system fixed these vulnerabilities.

We conducted an in-depth case study with three cases [118]. We followed the guidelines for industrially-based multiple-case studies [156]. In our study, each of the three systems was one case and the analysis unit was a *software project*. In each case (Chromium, PHP, and Thunderbird), we investigated **RQ3**, and **RQ4**.

5.1 Case Selection

We used the following criteria while selecting projects for our study: the system shall be (i) widely utilized by many users, (ii) among the top 50 software projects with the highest number of disclosed vulnerabilities [43], (iii) imple-

menting a wide range of security tactics, (iv) using a public issue tracking system for managing and fixing vulnerabilities, and (v) from different software domains. By applying these criteria, we aimed to ensure 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** [1] (a Web browser), **Mozilla Thunderbird** [2] (an email and news feed client) and **PHP** [3] (the interpreter of the PHP programming language) as case studies. These projects are diverse in size, age, and domain. However, they are similar with respect to their underlying programming language (they were mostly written in C/C++), as shown in Table 5.1.

Table 5.1: Details about the studied projects (statistics collected as of January 2017).

	Chromium	**PHP**	**Thunderbird**
Size (LOC)	>14 MLOC	>4 MLOC	>1 MLOC
# 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

[1] http://www.chromium.org/
[2] http://mozilla.org/thunderbird/
[3] http://php.net/

5.2 Data Collection and Analysis

We performed three steps while conducting this study. First, we identified the security tactics adopted in each project. Second, we retrieved each project's disclosed vulnerabilities in the NVD. Finally, we manually classified vulnerabilities as tactical and non-tactical. The collected artifacts and their relationships are portrayed in Figure 5.1 to make it easier to follow subsequent explanations.

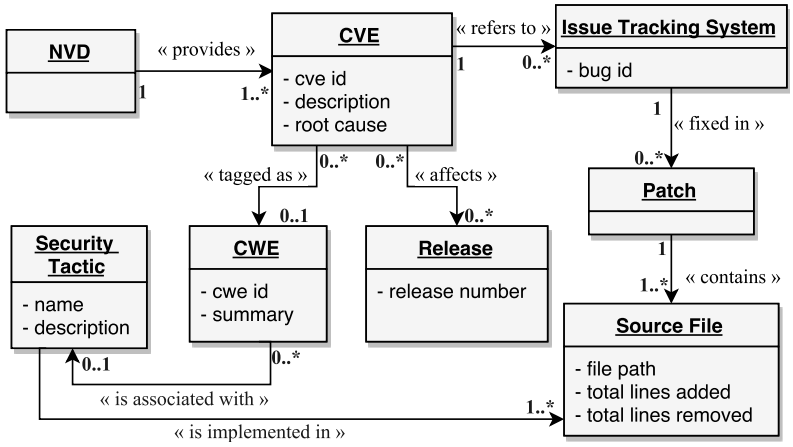


Figure 5.1: Data extraction information model

5.2.1 Step #1: Identifying Security Tactics in each Project

We identified in this first step the security tactics used in the three projects. Since this is a manual analysis, we performed the following complementary activities to ensure accuracy:

- We analyzed existing literature and technical documentation for each project [26] to see if any specific security tactic was adopted. We then manually checked if these tactics were implemented in their code.
- We searched tactic-related keywords (e.g. "authenticate") on the source code of the projects and browsed their source code to identify tactic-related files.
- We leveraged a prior work that automatically reverse-engineers architectural tactics from source code [96, 100].

The results of these steps were compiled into a single document. This document indicates the set of tactics adopted in each project and where in the code they are implemented. We contacted the developers involved in these projects if they agree with the identified tactics. Table 5.2 lists the security tactics identified for each project.

Table 5.2: Security tactics in Chromium, PHP, and Thunderbird

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

5.2.2 Step #2: Extracting Disclosed Vulnerabilities for each Project

As shown in Figure 5.1, CVEs are the starting point to collect the required artifacts. Therefore, we first extracted all these CVEs from NVD. Subsequently, we performed the following actions to enforce the accuracy and completeness of our dataset:

- **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). Thus, we manually analyzed each collected CVE to verify whether the corresponding entries in the issue tracking system of the three studied projects were included in the NVD. If NVD did not provide this information, we searched the CVE ID in the project's bug 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. In the end, we obtained a total of 2,386 CVEs spanning across the lifetimes of these projects until January 2016. Our dataset looked as follows: 1,252 CVEs were related to the Chromium project, 430 CVEs were associated with the PHP project, and 704 CVEs were in the Thunderbird project.

- **Removal of invalid CVEs:** We discarded *invalid vulnerabilities* which are those that were tagged as deprecated or as a duplicate of another CVE in the NVD. We also discarded CVEs that were not related to Chromium, PHP or Thunderbird (including applications written in PHP rather than in PHP itself). In addition, we excluded 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.
- **Tracing CVEs to patches:** We used the corresponding defect entry in the project's *Issue Tracking System* to extract the *patch* that was released to fix the vulnerability. From the *patch* we obtained the *source files* that were modified as part of the fix.

5.2.3 Step #3: Identification of Tactical and Non-Tactical Vulnerabilities

We used two approaches to classify vulnerabilities: a ***bottom-up*** approach and a ***top-down*** approach.

- **Bottom-up classification:** We read all the included vulnerability reports (CVEs) to classify them as *tactical* or *non-tactical*. To reduce inherent biases on this classification, we conducted 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 (SMEs) inspected all the collected CVEs and provided a *rationale* for how they classified CVE reports.

  We gave to each SME a set of instructions for classifying CVEs, as shown in Table 5.3. These instructions request these experts to read the CVE reports and its associated artifacts to identify: (i) where the issue is located, (ii) its

Table 5.3: 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:	

root causes, and (iii) a rationale and evidence for tactical vulnerabilities.

Both subject-matter experts also conducted detailed code reviews to classify the CVEs. We provided the source files that implement tactics in these projects and a matrix indicating the overlap of CVEs and tactical files. As described in Section 5.2.1, we reverse-engineered security tactics in the source code.

Once each subject-matter expert had finished their classification, they discussed their disagreements (based on each person's rationale) and resolved them. The percentage of vulnerability reports (CVEs) with disagreements as well as their Cohen's Kappa coefficients [40] in each project is shown in Table 5.4.

Table 5.4: Cohen's kappa coefficients and percentage of agreement when classifying vulnerability reports as tactical and non-tactical CVEs

	Chromium	**PHP**	**Thunderbird**
Cohen's Kappa coefficient (κ)	0.75	0.80	0.93
% Agreed	0.88	0.90	0.86

- **Top-down classification:** We used our CAWE catalog (Chapter 4) as a guidance to differentiate tactical and non-tactical vulnerabilities across the three systems. Since each CVE may have a CWE tag (Figure 5.1), we relied on that to obtain clues whether the problem is related to a security tactic or not. We used 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*). Since a few vulnerabilities did not have a CWE tag [4], we 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 5.1).

We then **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

[4]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.

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). Hence, 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.

Table 5.5: 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

Overview of the Vulnerability Dataset: Table 5.5 shows an overview of our vulnerability dataset, indicating the total number of collected vulnerabilities (# **CVEs**), the number of **discarded** instances (as explained in Section 5.2.3), the remaining CVEs that we **analyzed**, and how many **tactical** and **non-tactical CVEs** we found in each system. From this table, we observe 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.

Chapter 6

Detecting Deserialization Vulnerabilities

When we conducted the empirical study described in Chapter 5, we found that *improper input validation* is the most occurring class of tactical vulnerabilities [123, 126]. Although prior research widely explored certain instances of input validation flaws, such as XSS and SQL/XML/LDAP/Path injection [18, 21, 23, 38, 48, 51, 54, 57, 79, 83, 147–152, 160, 166], untrusted object deserialization is currently under-explored [113, 114].

When we look at CVEs related with serialization[1] (Figure 6.1), we can see a steep increase of deserialization vulnerabilities starting in 2015. In fact, deserialization vulnerabilities is top #13 in the "CWE Top 25 Most Dangerous Software Weaknesses" [145]. It went up 8 positions compared to the CWE Top 25 list published last year (2020). This is also consistent with existing analysis that found deserialization vulnerabilities among the top 10 vulnerability types across OSS projects, affecting a high number of open-source projects [11].

Given the pervasive and severity nature of deserialization vulnerabilities, our third goal is to develop Dodo, an approach that detects realization weak-

[1] This trend line was extracted by searching CVEs with the keyword *"serializ"* and grouping them by their published year.

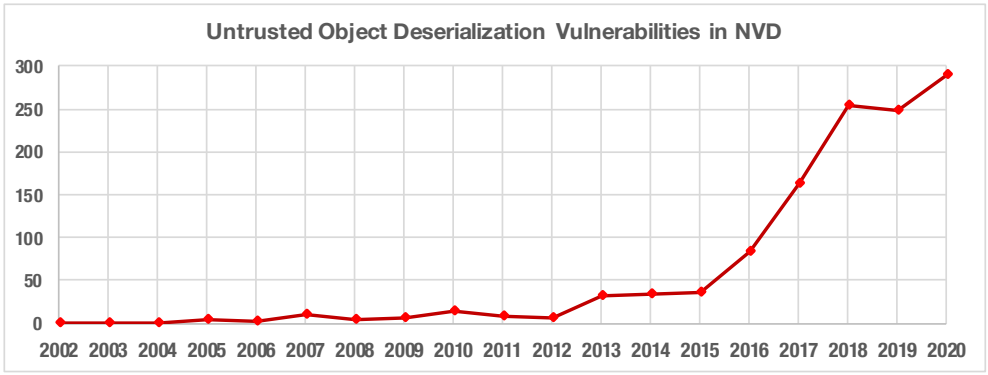


Figure 6.1: Trend line of untrusted object deserialization CVEs in the NVD

nesses caused by untrusted object deserialization. Before we introduce Dodo, we describe next the challenges involved in performing program analysis to find this type of vulnerability.

6.1 Challenges on Detecting Untrusted Object Deserialization

As mentioned in Section 2.5, untrusted object deserialization can be detected via *taint analysis*. Performing an effective taint analysis requires three key elements [129]: (i) a ***taint analysis policy*** that dictates how taint is *introduced*, *propagated* and *removed* in the system; (ii) a comprehensive list of ***sinks***; and (iii) a **pointer analysis** method that is precise enough to not inadvertently tags a value as tainted when it is not (*over-tainting*) and that does not miss tainted flows (*under-tainting*).

Devising an effective static taint analysis for detecting untrusted object deserialization has the following challenges:

- **"Dangerous code" can be anywhere in the classpath**: An application is composed of three main parts: the language's built-in classes, library classes, and the application code itself. Any class in the classpath has the potential to be leveraged in an exploit[2]. Hence, sinks may not be "dis-

[2]In Java, the class has to implement the `java.io.Serializable` interface.

coverable" by simply analyzing the application's source code because the application typically only calls a subset of classes from Java or external API classes. Thus, if a class with a dangerous operation is not being *used* in the application's code, the sink is deemed as unreachable by a traditional static analyzer. Thus, automatically detecting the problem requires a whole-program analysis of the source code that goes beyond the application's code itself (*i.e.*, analyzing classes even if they are not being used in the program).

- **Serialization is built on top of reflection**: The serialization/deserialization mechanism is implemented on top of reflection, a dynamic programming language feature. Reflection introduces method calls, aliases, and object allocations that are not directly observable from the source code; instead, these instructions are only determined at runtime [87]. Therefore, typical static analyses are not enough to reason over the presence of vulnerabilities as they often under-approximate the program's behavior in face of the uncertainty added by these dynamic features [140]. Consequently, they would miss vulnerabilities.

- **Specially crafted object**: Unlike other classes of injection problems in which the input is a primitive or string, the input provided by the attacker is an *abstract representation* of an object. This representation can have many formats (binary, XML, JSON, etc). This representation itself is not the problem; but rather the chain of method calls that are called during the reconstruction of the object described in this representation. Hence, the static analysis should abstract the object reconstruction which is uncertain; it does not know in advance the object provided by the attacker. Objects can have complex recursive structures due to non-primitive fields [49], and its primitive field's values are also uncertain. This characteristic makes it hard to analyze possible reachable code.

- **There is no "one-size-fits-all" approach to fix untrusted deserialization**: As previously described in the literature, taint sanitization (*i.e.*, determining when taint may be removed from a value) is a challenging problem to solve [129]. Properly identifying the safe removal of taint plays a crucial role in preventing the taint analysis to over-tainting variables.

  As it will be described in Section 6.2, there are multiple different ways developers can mitigate untrusted object deserialization and these fixes often

require more than just invoking a method call to sanitize a primitive data type [108]. Hence, it is difficult to define suitable taint sanitization rules that can abstract *all* possible ways an application can fix this vulnerability in order to identify when a tainted variable can be marked as safe. Therefore, a taint analysis approach that assumes that data that flows through certain methods are validated (*e.g.*, [149–151]) will not suffice for solving this problem.

These challenges wreak havoc on static analysis techniques for vulnerability detection, leading to over- or under-tainting of variables (depending on the underlying abstraction used to perform the static analysis). To overcome the challenges above, one can perform *dynamic taint analysis* [129], in which the taint analysis is performed as the system is executed. However, dynamic taint analyses require a predefined input and can explore only *one* execution path at a time. Since objects can have complex structures (*e.g.*, circular references), there are many runtime paths that would need to be exercised, which makes the analysis time-consuming.

6.2 Mitigating Untrusted Object Deserialization

By scrutinizing multiple vulnerability reports [102], white papers [80, 130], talks [59, 128], books [94], and previous papers [39, 49, 108, 135], we observed that there are three main ways to fix untrusted deserialization: (i) by preventing untrusted data to reach a sink (***unreachable sinks***); (ii) by enforcing the integrity of serialized and deserialized objects (***enforcing integrity***); or (iii) ***compartmentalization***. The mitigation techniques and their corresponding category is presented in Table 6.1.

6.2.1 Group 1: Unreachable sinks

This category contains mitigation techniques that make the sink *unreachable*. These mitigation techniques are:

M1.1 Allowed/Blocked list of classes: It maintains a list of classes that *may* or *may not* be deserialized (*allow list* and *block list*, respectively). This is achieved by overriding the `resolveClass(ObjectStreamClass o)` method in `ObjectInputStream` and throwing an exception when the

Table 6.1: Mitigation techniques for untrusted object deserialization

Category	Mitigation
Group 1 *Unreachable Sinks*	M1.1 Allowed/Blocked list of classes M1.2 Prevent Deserialization of Domain Objects
Group 2 *Enforcing Integrity*	M2.1 Adding the "transient" to a sensitive field M2.2 Sign and Seal Serialized Objects
Group 3 *Compartmentalization*	M3.1 Deserialize within a sandbox M3.2 Reduce privileges before deserializing from a privileged context

object type is either in the *block list* or not in the *allow list* [130].

M1.2 Prevent deserialization of domain objects: This mitigation is typically used when the application has a class that extends another serializable class (directly or indirectly) which provides concrete implementations to callback methods (*i.e.*, "magic methods"). Therefore, to prevent malicious uses of these subclasses, the application breaks the chain of method calls by throwing an exception. Hence, the dangerous sink is unreachable because the chain of calls from `readObject()` to the sink method is broken due to a thrown exception.

6.2.2 Group 2: Enforcing object integrity

This category encompasses mitigation approaches that enforce the integrity of the object:

M2.1 Add transient to a "sensitive" field: To prevent serializing fields with sensitive information (*e.g.*, passwords), applications enforce that these fields are not included when the object is serialized. This is achieved by adding the keyword `transient` to the field [108].

M2.2 Sign and seal serialized objects: This mitigation is used when: (i) the application has to transmit sensitive data, (ii) it does not have a secure transport channel (*e.g.*, SSL), and (iii) sensitive data must persist over some time (*e.g.*, on a hard drive). In this case, marking fields as "transient" would not fulfill the application's needs [94]. This mitigation is based on the Obfuscated Transfer Object security pattern [142]. It

involves two steps: the generation of a signature for the object (*signing*) followed by encrypting the object together with its signature (*sealing*).

6.2.3 Group 3: Compartmentalization

This category includes mitigation approaches in which the system enforces policies at runtime to prevent object deserialization misuse.

M3.1 Deserialize within a sandbox: The application creates a sandbox with a set of policies that are enforced at runtime. Thus, if the deserialized object triggers an operation forbidden by the policy, the object reconstruction is stopped [8, 10].

M2.2 Minimize privileges before deserializing from a privileged context: The program creates a child process with dropped privileges prior to deserializing an object [94]. The child process contains the minimum required privileges to reconstruct the object (*e.g.*, the child process may not have permission to read/write to files).

6.3 Taint Analysis for Detecting Untrusted Object Deserialization

As previously discussed in Section 6.1, the key challenge when performing static taint analysis is to create an *abstraction* that is suitable for the client analysis (in our work, that is untrusted object deserialization detection). We need to accurately track intra- and inter-procedural data flows. For inter-procedural data flows, it is crucial to correctly compute the possible targets for each method call (*i.e.*, method dispatch). However, some programming features, such as polymorphism and reflection [87], makes it difficult to determine the dispatch at a given call site. Consequently, dangerous paths would be missed as a result of call graphs that unsoundly handle such features.

To overcome these challenges and taking into account the specific characteristics of deserialization vulnerabilities, we developed **Dodo (untruste*D* *O*bject *D*eserialization detect*O*r)**, a ***novel hybrid approach*** that combines the strengths of static and dynamic analyses. Dodo encompasses three steps

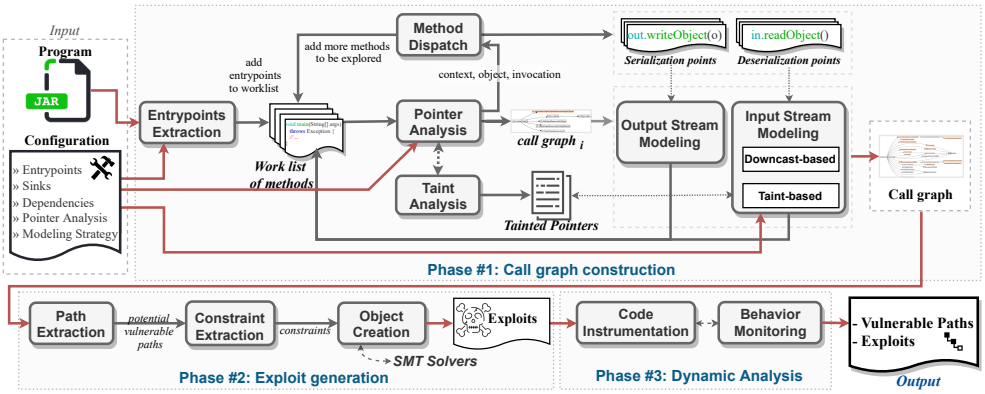


Figure 6.2: DODO Overview

which are depicted in Figure 6.2. ① It performs an imprecise but sound[3] static analysis of the system to find *potential vulnerable paths*. ② It *generates exploits* for each potential vulnerable path by leveraging symbolic analysis and constraint solving. ③ It executes the system providing the crafted exploit as input. It then verifies whether the system indeed contain the vulnerability or it has implemented correct mitigation procedures to counteract it. This is achieved by instrumenting the program and observing the occurrence of exceptions, crashes and other vulnerability occurrence indicators.

The key insight for this hybrid approach is that the imprecision introduced by the static analyzer ① will be removed on our final stage that involves dynamic analysis ③ . Thus, if any of the found potential vulnerable paths turns out to be infeasible paths at runtime, they will be disregarded during the dynamic analysis. This minimizes the occurrence of false positives. Each phase shown in Figure 6.2 is detailed in the next sections.

6.4 Phase 1: Call Graph Construction

DODO takes as input both the *system's binary* (Java bytecode) and a *configuration file* that contains:

[3]"Sound" in this context means it is sound with respect to deserialization features.

- *Entrypoint methods*: a CSV file that contains the signatures for the methods that start the execution of a program. Traditional Java desktop-based applications have a single entrypoint method (*i.e.*, the `main(String [])` method). However, Android applications and Web applications may have multiple entrypoints. Thus, this configuration gives a leeway for us to accommodate multiple types of systems written in Java. Moreover, this allows us to perform partial program analysis if needed (*i.e.*, starting the analysis from an inner method, other than an actual entrypoint).
- *Sinks*: a CSV file with signatures for methods that perform security-critical operations, such as executing a command or sending data over a network.
- *Dependencies* (optional): list of paths to libraries' JAR files in which the program depends upon (and are part of the program's classpath).
- *Pointer Analysis*: which pointer analysis algorithm to use to compute the call graph. DODO currently supports two kinds of pointer analyses: `0-n-CFA` or `n-CFA` (where n can be specified).
- *Modeling Strategy*: as we describe in Section 6.4.2, the DODO's call graph construction employs two modeling to abstract the inner implementation details of Java's deserialization API. Thus, a user can specify which modeling strategy to use (*downcast-based* or *taint-based*).

Based on this configuration, DODO constructs the ***program's call graph***. A call graph is a directed graph in which the nodes are *methods* in the program and the edges indicates *caller-callee relationships* [64]. A system's call graph is a core data structure for performing many interprocedural analyses, including taint analysis.

As discussed in Chapter 2, object deserialization relies on reflection to make invocations to callback methods (*i.e.*, "magic methods"). Hence, call graphs that are built using existing pointer analysis algorithms (e.g., `n-CFA`, `n-l-CFA`, `n-object-sensitive`, *etc.*) do not include these callbacks, resulting in an unsound call graph with respect to deserialization [113, 114]. Therefore, if we rely on these call graphs to perform vulnerability detection, then we miss possible vulnerable paths due to their unsoundness.

To construct sound call graphs one could argue that conservative algorithms, such as Class Hierarchy Analysis (CHA) [47] and Rapid Type Analysis

(RTA) [20], would suffice because they compute call graphs based solely on the class hierarchy and implemented methods. However, performing taint analysis over these call graphs will not scale well because the call graph is much larger (number of nodes and edges) in order to encompass all the possible behaviors [122]. Furthermore, since these conservative algorithms are highly imprecise it results in over-tainting of variables.

Hence, we need an approach that is sound with respect to deserialization, yet precise enough to counteract performance issues that will be introduced by attempting to find vulnerable paths within these call graphs. To achieve this goal, DODO employs an iterative call graph construction technique [64]. It involves two major steps: (1) a set of iterations over a worklist of methods to create an initial call graph using an underlying pointer analysis method; and (2) a refinement of the initial call graph by applying a set of assumptions performed iteratively until a fixpoint is reached.

6.4.1 Step 1: Initial Call Graph Construction

DODO first extracts a set of ***entrypoint methods*** $m \in E$ added to our ***worklist*** $\mathcal{W}$. This worklist tracks the methods m under a context c that have to be traversed and analyzed, i.e. $\langle m, c\rangle \in \mathcal{W}$, where a ***context*** c is an abstraction of the program's state. Since the worklist $\mathcal{W}$ tracks methods within a context, the entrypoints methods added to $\mathcal{W}$ are assigned a global context, which we denote as $\emptyset$. As a result of this first step, the worklist is initialized as:

$$\mathcal{W} = \{\langle m, \emptyset\rangle \mid \forall m \in E\}$$

Starting from the entrypoint methods identified, DODO constructs an ***initial (unsound) call graph*** (*i.e.*, call graph_0) using the underlying pointer analysis algorithm selected by the client analysis (*e.g.*, `n-CFA`, etc). Each method in the worklist $\langle m, c\rangle \in \mathcal{W}$ is converted into an Intermediary Representation (IR) in Single Static Assignment form (SSA) [44]. Instructions i in a method's IR have a ***scope***. The scope is based on where the method m is declared. It can either be *application*, *extension* (code from imported libraries/APIs), or *primordial* (Java's standard API classes).

Each instruction i in a method's IR is visited following the rules by the underlying pointer analysis algorithm. We point the reader to the work by

Sridharan *et al.* [140] which provides a generic formulation for multiple points-to analysis policies.

When visiting instance invocation instructions i in a method m (*i.e.*, `x = o.g(`a_1`,`a_2`,...,`a_n`)`), the static analysis computes the possible dispatches (call targets) for the method g as follows:

$$\texttt{targets} = \mathit{dispatch}(\mathit{pt}(\langle o, c\rangle), g)$$

The dispatch mechanism computes the possible method targets by querying the current points-to set for the object o at the current context c ($\mathit{pt}(\langle o, c\rangle)$)as well as the declared target g (method signature). If the invocation instruction occurs at a ***serialization*** or ***deserialization point***, then the *dispatch* function implemented by our approach creates a ***synthetic method***. A serialization point is the callsite of an invocation instruction i within the application scope that calls `ObjectOutputStream`'s `writeObject(Object)` method. Similarly, the callsites of an invocation i within the application scope that calls `ObjectInputStream`'s `readObject()` are deserialization points.

In this scenario, the computed dispatch is a ***synthetic method*** m_s that aims to model the runtime behavior for the `readObject()` and `writeObject()` from the classes `ObjectInputStream` and `ObjectOutputStream`, respectively. These synthetic models $m_s \in M_s$ are initially created *without* instructions. Their instructions are computed and added during the call graph refinement step (Step 2 described in Section 6.4.2).

Calls to synthetic methods (models) are *n-callsite-sensitive* [140]. We use this context-sensitiveness policy to account for the fact that one can use the same `Object(In|Out)putStream` instance to read/write multiple objects. Thus, we want to disambiguate these paths in the call graph.

As a result of this first step, we obtain the ***initial callgraph*** (g_i) and a list of the call sites at the ***serialization and deserialization points***.

6.4.2 Step 2: Call Graph Refinement

In this second step, DODO takes as input the current call graph g_i resulting from the iterations in the prior step. The call graph g_i contains nodes that are either *actual* methods in the application or *synthetic methods* created in the

previous phase.

Since synthetic methods are empty (*i.e.*, without instructions), in this second phase, we iteratively refine them by adding instructions to model the semantics of Java's serialization and deserialization API. In the next subsections we outline the process of performing code modeling for the `ObjectOutputStream` and `ObjectInputStream` classes.

Modeling the ObjectOutputStream Class (Serialization)

The modeling for the `ObjectOutputStream` class is presented in Algorithm 1. This algorithm takes as input the set of instructions I at the *serialization points*, *i.e.*, invocations to the `ObjectOutputStream.writeObject(Object)` method.

For each invocation $i \in I$, it obtains the points-to set $pt(\langle o_i, c \rangle)$ for the object o_i passed as the first argument to `writeObject(Object)` (line 2). These points-to set contains all the allocated types t for o_i under the context c. Since the `writeObject`'s argument is of type `java.lang.Object`, it first adds to m_s a type cast instruction that refines the first parameter to the type t. In case the class type t implements any callback method invoked during serialization (see Section 2.4.1 for their signatures), then it adds an invocation instruction from m_s targeting this callback method (line 8).

Subsequently, it iterates over all non-static fields f from the class t and compute their points-to sets (see the `foreach` in line 10). If the concrete types allocated to the field contains callback methods, it adds the following instructions: (i) an instruction to get the instance field f from the object; (ii) a downcast to the field's type; and (iii) invocation(s) to the callback method from the field's declaring class. After adding all these instructions to the synthetic method m_s, Dodo re-adds the synthetic method to its worklist (as depicted in Figure 6.2).

Modeling the ObjectInputStream Class (Deserialization)

Dodo uses two distinct approaches to model the `ObjectInputStream` class: (1) a **downcast-based** approach that relies on type casts in the program to infer potential objects that can be deserialized and (2) a **taint-based** approach

Algorithm 1: Object serialization modeling

Input: Set of invocation instructions to `writeObject`: I; `/* (serialization points) */`
Project's initial call graph: G;
Output: Set of refined synthetic models M_s

```
1  foreach instruction ∈ I do
2      o_i ← argument(1, instruction)
3      c ← context(instruction)
4      m_s ← target(instruction)
5      foreach t ∈ pt(⟨o_i, c⟩) do
6          addTypeCast(m_s,t)
7          foreach callback ∈ callbacks(t) do
8              addInvoke(m_s, callback)
9          end
10         foreach f ∈ fields(t) do
11             foreach fieldType ∈ pt(⟨o_i.f, c⟩) do
12                 if fieldType has any callback then
13                     addGetField(m_s, f)
14                     addTypeCast(m_s, fieldType)
15                     foreach callback ∈ callbacks(fieldType) do
16                         addInvoke(m_s, callback)
17                     end
18                 end
19             end
20         end
21     end
22     addToWorkList(m_s,c)
23 end
```

that performs taint analysis alongside pointer analysis to refine the call graph based on the variables' taint states.

— **Downcast-Based Deserialization Modeling:** Many classes in a classpath (*e.g.*, Java's Swing classes) implement the `Serializable` interface. Hence, there is a high amount of possible calls that could be included in the resulting call graph. Thus, this modeling strategy uses these assumptions:

Assumption #1
There is no dynamic loading of remote classes. Only the classes in the classpath are available to be (de)serialized (closed-world assumption [93]);

Assumption #2
All fields in serializable classes are not null. These fields can be instantiated with *any* type that is safe. This assumption ensures that we can soundly infer the possible targets for invocations within callback methods made via inner fields (*e.g.*, line 8 of the `CommandTask` class in Listing 3).

Assumption #3
All type refinements (downcasts) are safe. Thus, downcasts can be used to infer the possible callback methods invoked during the serialization/deserialization and points-to sets for fields within serializable classes. This assumption aims to reduce the points-to sets for these fields since many classes in the classpath implement the `java.io.Serializable` interface.

Given these assumptions, DODO traverses the def-use chains [12] of the caller's IR to find any downcasts for the returned deserialized object (line 4 in Algorithm 2):

$$\mathbf{o_{ret}} \texttt{ = in.readObject();}$$

$$\ldots$$

$$x \texttt{ = (t) } \mathbf{o_{ret}}\texttt{;}$$

For each downcast type t (line 4), DODO adds an allocation instruction to m_s followed by invocations to callbacks implemented by t (if any exists). Next, it iterates over all instance fields of the `type` and compute the possible serializable classes that are type-safe for the field (lines 9-10). For each possible

safe type, it adds a field allocation. Then, if the possible type has a callback method, it adds to m_s: a cast to the possible type (line 15), and an invocation to the callback (line 16).

The `readObject()` method from `ObjectInputStream` that DODO is modelling, returns an object instance. The returned value can either be the result from the `readObject()` or `readResolve()`. Thus, DODO adds to m_s an instructions that returns a value from a ϕ function (line 20). This ϕ function is used to indicate that the return value can either be from `readObject()` or `readResolve()` callbacks. Finally, the synthetic method m_s is re-added to the worklist $\mathcal{W}$ (line 21).

Algorithm 2: Downcast-based object deserialization modeling

```
   Input: Set of invocation instructions to ObjectInputStream.readObject: I;
          Project's initial call graph: G;
          Serializable classes in the classpath: S;
   Output: Set of refined synthetic models M_s
 1 foreach instruction in I do
 2     ⟨o_ret, c⟩ ← getPointerForReturnValue(instruction)
 3     m_s ← declaredTarget(instruction)
 4     foreach t ∈ downcasts(o_ret) do
 5         o_i ← addAllocation(m_s, t)
 6         foreach callback ∈ callbacks(t) do
 7             addInvoke(m_s, callback)
 8         end
 9         foreach f ∈ fields(t) do
10             foreach type ∈ possibleTypes(f) do
11                 addAllocation(m_s,o_i.f, type)
12                 foreach callback ∈ callbacks(type) do
13                     addGetField(m_s, o_i.f)
14                     addTypeCast(m_s, o_i.f, type)
15                     addInvoke(m_s, callback)
16                 end
17             end
18         end
19     end
20     addReturnInstruction(m_s)
21     addToWorkList(m_s, c)
22 end
```

Handling object array/collection fields: To soundly handle fields that are

collections of objects (*e.g.*, arrays, sets, lists, etc), we apply an extra assumption:

Assumption #4

All array/collection fields contains at least one object of each possible type. This ensures that we soundly infer possible targets for calls whose receiver object is an element from an array/collection field. To ensure DODO keeps its soundness promises, it is not *container-sensitive*, *i.e.*, it does not keep different points-to sets for `a[i]` and `a[j]` ($i \neq j$).

— **TAINT-BASED DESERIALIZATION MODELING:** In this code modeling approach, DODO performs *pointer analysis* in parallel with *taint analysis* to compute the taint state of variables and points-to sets. Each instruction in the method's IR is visited following the rules by the underlying pointer analysis and our taint analysis algorithm [140].

Each pointer `x` in a program has a ***taint state*** $\tau(x)$, which can either be `true` (tainted) or `false` (untainted). We provide below the formulation of our taint analysis policy.

Taint Introduction: As listed below, DODO marks as tainted all the allocations to a gadget class G_c (1), all the *fields* of these instances (2), as well as the "this" pointer for a call back method invocation, *i.e.*, magic method $m_c \in M_c$ (3):

(1) The pointer for `x` in the instruction `x = new` G_c`()`

$$\tau(x) = \texttt{true}$$

(2) The pointers for all the non-static fields of `x`

$$\forall f_i \in fields(x) : \tau(x.f_i) = \texttt{true}$$

(3) The `this` pointer in m_c

$$\tau(m_c.this) = \texttt{true}$$

Taint Propagation Rules: As DODO parses the method's instructions, it uses the rules listed in Table 6.2 to compute the taint states of the program's variables. Taint ***is never removed*** from a variable. Although this will make the underlying call graph more imprecise, our goal with this approach is to

soundly reason over *all* possible runtime paths, and remove the spurious paths during the dynamic analysis (Phase #3).

Table 6.2: Taint propagation rules

Instruction at method m under a context c	**Taint Propagation Rule**	
`x = T.f`	$\tau(x) = \tau(x) \vee \tau(T.f)$	[Load-Static]
`x = y.f`	$\tau(x) = \tau(x) \vee \tau(y) \vee \tau(y.f)$	[Load-Instance]
`x.f = y`	$\tau(x.f) = \tau(x.f) \vee \tau(y)$	[Store-Instance]
`T.f = y`	$\tau(T.f) = \tau(T.f) \vee \tau(y)$	[Store-Static]
`x = o.g(a1,⋯,an)`	$\forall a_i \in A_j, \forall p_i \in P_g : \tau(p_i) = \tau(p_i) \vee \tau(a_i)$ $\tau(g_{this}) = \tau(g_{this}) \vee \tau(o)$	[Instance-Call-Args]
	$\tau(x) = \tau(x) \vee \tau(g_{ret})$	[Instance-Call-Return]
	Side Effect: $\tau(o) = true \rightarrow$ $pt(\langle o, c\rangle) = pt(\langle o, c\rangle) \cup targetTypes(o, c, g)$	[Call-Side-Effect]
`x = T.g(a1,⋯,an)`	$\forall a_i \in A_j, \forall p_i \in P_g : \tau(p_i) = \tau(p_i) \vee \tau(a_i)$	[Static-Call-Args]
	$\tau(x) = \tau(x) \vee \tau(g_{ret})$	[Static-Call-Return]
`return x`	$\tau(m_{ret}) = \tau(m_{ret}) \vee \tau(x)$	[Return]
	Side Effect: $\mathcal{W} = \mathcal{W} \cup C_m$	[Return-Side-Effect]
`x = y[i]`	$\tau(x) = \tau(x) \vee \tau(y)$	[Array-Load]
`x[i] = y`	$\tau(x) = \tau(x) \vee \tau(y)$	[Array-Store]
`ϕ = v1,v2,...,vn`	$\tau(\phi) = \tau(v_1) \vee \tau(v_2) \vee \cdots \vee \tau(v_n)$	[Phi]
`x = (TypeCast) y`	$\tau(x) = \tau(x) \vee \tau(y)$	[Checkcast]

Assignment Instructions

As shown in Table 6.2, the rules for assignment instructions are:

$$\texttt{lhs = rhs} \longrightarrow \tau(lhs) = \tau(lhs) \vee \tau(rhs)$$

The pointer for the left-hand side (`lhs`) is tainted if the pointer for the right-hand side (`rhs`) is also tainted (or the left-hand side itself was already previously tainted). This is the case for the rules Load-Static, Load-Instance, Store-Instance, Store-Static, Instance-Call-Return, Static-Call-Return, Return, Array-Load, Checkcast, and Array-Store.

Phi Functions

Phi functions (ϕ) are special statements that are inserted into a method's

SSA form to represent possible values for a variable depending on the control flow path taken. The taint for the pointer of phi $\tau(\phi)$ will be tainted if *any* of the possible variables' pointers are tainted.

Method Invocations

When there is a method invocation, it can either be a static invocation or an invocation to an instance method. In both cases, each passed parameter p_i is assigned to the corresponding argument a_i from the invoked method. Consequently, the rules INSTANCE-CALL-ARGS, and STATIC-CALL-ARGS are propagated similar to assignment instructions. Notice, however, that for instance methods there is a special variable m_{this} denoting the "this" pointer for that method. Hence, the rule INSTANCE-CALL-ARGS propagates the taint from the caller object to the "this" pointer $\tau(g_{this})$.

Side Effects to the Pointer Analysis Engine: *Method invocations* and *return instructions* introduce ***side-effects*** to the static analysis engine state, labelled in Table 6.2 as CALL-SIDE-EFFECT and RETURN-SIDE-EFFECT, respectively.

Side-effects from Tainted Instance Method Invocations

When there is an instance method invocation $o.g(...)$ and the object o is tainted, then DODO computes the possible method targets for the call $o.g(...)$ *soundly*. The method dispatch is performed very similar to the Class Hierarchy Analysis (CHA) [47] with the difference that it considers only classes that implements the `Serializable` interface. The dispatch (*i.e.*, `targetTypes(o,g)`) is computed as follows:

1. it obtains the static type t for o, i.e. $t = type(o)$;
2. it extracts the set of classes based on the inheritance hierarchy for T (*i.e.*, $T = cone(t)$, where cone returns the list of all descendants of t, including t itself [154].
3. it computes the subset $C \subseteq T$ that includes only the types (classes) which provide a concrete implementation matching the signature of the invoked method g.
4. the possible target methods are all the methods from the set A_t in which their classes are serializable (*i.e.*, implements the serializable

interface directly or via inheritance).

Once the dispatch is computed, the points to set for $pt(\langle o, c \rangle)$ adds all the elements from `targetTypes(o,g)`.

Side-effects from Method Return Values

In a scenario where a method m has a tainted return value $\tau(m_{ret}) =$ `true`, all the callers of m are re-added to the $\mathcal{W}$. Since the return is tainted, we need to back propagate this information to all the callers of m to ensure that the rule INSTANCE-CALL-RETURN and STATIC-CALL-RETURN would correctly propagate the taint state.

Context-sensitivity for Tainted Method Calls: Our taint-based call graph construction algorithm is *agnostic* to the pointer analysis policy (*e.g.*, `0-1-CFA`). This means that a client analysis could choose to use a context insensitive analysis (e.g. `0-CFA`). Since tainted pointers are likely to have a large points-to set and we use a sound analysis to compute all possibilities, we should avoid merging point-to-sets of these tainted variables. Otherwise, the resulting pointer analysis would be too imprecise for us to further generate exploits. Therefore, we use `n-CFA` sensitive for tainted method calls (even if we use an insensitive analysis for all the other pointers).

6.5 Phase 2: Exploit Generation

This second phase takes as input the system's *call graph* and the locations of *sinks* generate exploits. Exploits are a set of objects that can potentially trigger a vulnerability at runtime if the system does not implement proper mitigation procedures. To generate exploits, this second phase leverages symbolic analysis and off-the-shelf SMT solvers by performing three steps, described in the next subsections.

6.5.1 Step 1: Path extraction

We compute a *backward slice* [163] for each sink. These slices contain all the statements in which the sink is *control dependent*. For each slice previously computed, we reconstruct the paths between *sources* and *sinks*. The output of this step is a set of *potential* vulnerable paths $p \in P$ from a magic method to a sink.

6.5.2 Step 2: Path Constraint Analysis and Solving

Each path computed in the previous path may include constraints that arise due to conditional structures, such as `if/else` or loops. Moreover, recall that our taint-based call graph construction algorithm introduces "fake" allocations to the points-to set of tainted pointers (CALL-SIDE-EFFECT). These "fake" allocations interfere with the shape of the call graph (nodes and edges) and, as such, with the reachability to sinks.

Thus, DODO extracts two kinds of constraint during this phase:

- **Path constraints**: they are a result of conditional structures within the code. They strict the actual values for the ***primitive*** fields of the objects in the exploit. Thus, DODO collects constraints in a conditional such as `a <operator> b` only if the variables `a` and `b` are primitives, string objects, or objects from primitive wrapper classes (*e.g.*, `java.lang.Integer`).
- **Type constraints**: these constraints can arise in two scenarios: (i) due to method invocations within the vulnerable path's call sites, and (ii) as a result of object-related conditional structures (*e.g.*, `obj1 == obj2`). These constraints strict the ***runtime types*** for reference fields (*i.e.*, fields that are other objects themselves). Table 6.4 enumerates how each instruction generates a constraint over the runtime type for a pointer $p = \langle o, c \rangle$ that points to an object o under a context c. Noticed that for `!obj1.equals(obj2)` and `obj1 != obj2` nothing can be concluded about the runtime types for obj_1 or obj_2. For example, if we have obj_1 `= "hello"` and obj_2 `= "world"` they have the same type (String) but fail both conditions. Similarly, if obj_1 `= "hello"` and obj_2 `= "hello"` then they would have the same type and would make the condition `obj1.equals(obj2)` equals to `true`, but `obj1 != obj2` might be equals to `false` if they are not the same *reference* (*i.e.*, same memory location). Thus, we denote with a * when DODO does not infer a type constraint over a given instruction. For invocation instructions such as `obj.g(`$\mathtt{a}_1$`,`$\cdots$`,`$\mathtt{a}_n$`)`, DODO first queries what is the method dispatch that was computed in the vulnerable path p ($callTargetType(g, p)$). Then, DODO constrains the type for the object's pointer to be equals to the class that declares the method in the dispatch.

To solve these constraints, DODO relies on Z3 [46], a Satisfiability Modulo Theories (SMT) solver. Notice, however, that *code* constraints are not directly

Table 6.3: Constraint generation rules

Instruction at method m under a context c within a vulnerable path p	**Type constraint**
`obj1 == obj2`	$type(\langle obj_1, c\rangle) = type(\langle obj_2, c\rangle)$
`obj1 != obj2`	$*$
`obj instanceof T`	$type(\langle obj, c\rangle) \in cone(T)$
`!(obj instanceof T)`	$type(\langle obj, c\rangle) \notin cone(T)$
`obj == null`	$type(\langle obj, c\rangle) = \emptyset$
`obj != null`	$type(\langle obj, c\rangle) = cone(staticType(\langle obj, c\rangle))$
`obj1.equals(obj2)`	$type(\langle obj_1, c\rangle) = type(\langle obj_2, c\rangle)$
`!(obj1.equals(obj2))`	$*$
`obj.g(a1,⋯,an)`	$type(\langle obj, c\rangle) = callTargetType(g, p)$

```
v5.equals("windows") AND
      !v3.isEmpty()
```

```
(declare-fun v1795_3 () String)
(declare-fun v1720_5 () String)
(assert (let ((a!1 (not (= (= (str.len v1795_3) 0) false))))
             (and (not (= (= v1720_5 "windows") false))
             (not a!1))
        )
)
```

Figure 6.3: An example of a path constraint converted into the SMT-Lib format [24]

handled by SMT solvers. Thus, we convert these code-level constraints to first-order predicates that follow the syntax established in the SMT-Lib standard [24]. Figure 6.3 contains an example on how we encode a *path constraint* using the SMT-Lib format [24]. Notice that each variable identifier encompasses two parts. The first part is the *scope*, which is the id for the call graph node in which the (SSA) variable is used. The second part is the *variable identifier*, which is its unique id within the method's IR (that is in SSA form). For example, the function identifier `v1720_5` indicates a String with an identifier equals to `5` declared inside the call graph node that has an id equals to `1720`.

When Dodo passes these encoded ***constraints*** to Z3, it gets back a result that can either be *satisfiable* (`SAT`) or *unsatisfiable* (`UNSAT`). If the constraint

is satisfiable, Z3 also returns a solution. We trace these values back to their corresponding primitive fields such that we can instantiate them with proper values and types.

6.5.3 Step 3: Malicious Objects Instantiation

DODO takes the constraints and their solutions (if satisfiable) in the previous step to create malicious objects (*i.e.*, ***potential exploits***). This is a top-down instantiation process, in which the top-level object is of the type of the class that contains the magic method (*i.e.*, the top-level "gadget class").

First, DODO creates an object for this top-level gadget class and initialize all its fields to default values: integers are set to 0, booleans are equals to false, and object fields are null. Subsequently, DODO instantiates all object fields in the class according to the type constraints previously solved. Notice that this process is inherently recursive, since objects can have back references. The exploit generation engine keeps track of back references to avoid getting stuck.

Each malicious object is serialized and saved into a text file. In the next step, we feed this exploit to the program to verify whether the vulnerability exists or not.

6.6 Phase 3: Dynamic Analysis

The exploits generated in the previous phase were based on *potential* vulnerable paths computed from a static analysis, but they cannot confirm whether the vulnerability is a true positive or not. Moreover, unlike our downcast-based and taint-based call graph algorithm can introduce imprecision for a subset of pointers to ensure that parts of the program involved in deserialization are handled soundly. Therefore, the goal of Phase 3 is to verify whether the exploits previously generated can effectively expose a vulnerability or the system has proper mitigation procedures that would prevent an exploitation.

To achieve this goal, we generate a JAR file that includes the program under analysis and a `Driver` class in the classpath, as shown in Figure 6.4. This driver class reads a file whose path is provided as a program argument.

DODO starts the program execution from the `Driver`'s main method and

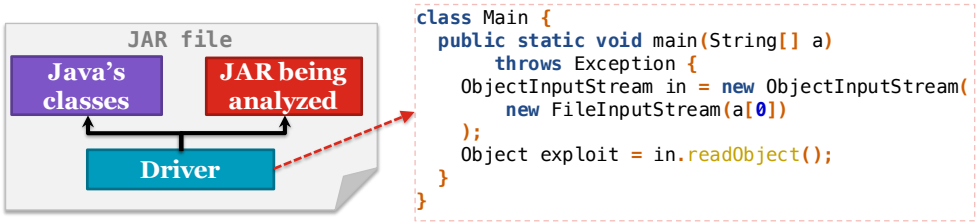


Figure 6.4: Details of the dynamic analysis performed by DODO

passes as program argument the path to where the exploit was saved. DODO then monitors the program's execution. It considers the program to be vulnerable if:

1. the sink statement is executed, and
2. the arguments that reach the sink are tainted.

When DODO detects that the program is vulnerable using the heuristic above, it notifies the developer. The notification includes the exploit as an evidence for the vulnerability, which could aid the process of debugging to fix the errors.

6.7 Demonstrative Example

Consider the code snippet in Listing 4. The class `Main` has a `main` method that reads an object from a file, whose path is provided as a program argument. This program contains other four classes (`CacheManager`, `TaskExecutor`, `CommandTask`, and `Config`). The `CacheManager` class has a magic method (implementation highlighted) whose internal method calls can reach to the sink located on `TaskExecutor` class (highlighted). Similarly, the `Config` class has a magic method that invokes a sink (implementation highlighted). In the next subsections, we walk through how DODO analyzes this program to find vulnerabilities. We demonstrate DODO considering that we selected `0-1-CFA` [157] as the main pointer analysis method. As described in Section 6.4.1, calls to the model methods are *n-callsite-sensitive*. In this example, we use `n=1` (*i.e.*, *1-callsite-sensitive*).

```
1 class Main {
2   public static void main(String[] a)
3       throws Exception {
4     FileInputStream f=new FileInputStream(a[0]);
5     ObjectInputStream in=new ObjectInputStream(f);
6     Config obj = (Config) in.readObject();
7   }
8 }
9 class CommandTask
10     implements Runnable, Serializable {
11   private String cmd;
12   private TaskExecutor taskExecutor;
13   @Override
14   public void run() {
15     if (!cmd.isEmpty() && taskExecutor != null)
16       taskExecutor.executeCmd(cmd); /* site @24 */
17   }
18 }
19 class TaskExecutor implements Serializable {
20   public void executeCmd(String cmd) {
21     try {
22       Runtime rt = Runtime.getRuntime();
23       rt.exec(cmd);
24     } catch (IOException e) {  }
25   }
26 }
27 class Config implements Serializable {
28   private String page;
29   public void readObject(ObjectInputStream ois)
30       throws IOException, ClassNotFoundException {
31     ois.defaultReadObject();
32     Runtime rt = Runtime.getRuntime();
33     rt.exec("open http://localhost/" + page);
34   }
35 }
```

```
35 class CacheManager implements Serializable {
36
37   private Runnable task;
38   private Runnable[] taskArray;
39   private List<Runnable> taskList;
40   private Set<Runnable> taskSet;
41   private Map<String, Runnable> taskMap;
42   private String os;
43   private long timestamp;
44
45   public void readObject(ObjectInputStream ois)
46       throws IOException, ClassNotFoundException {
47     ois.defaultReadObject();
48     Runnable r;
49     if(os.equals("windows") && task instanceof CommandTask){
50       r = getInitHook();     /* site @32 */
51       r.run();
52     }else {
53       r = getFromArray();
54       r.run();               /* site @46 */
55       r = getFromList();
56       r.run();               /* site @57 */
57       r = getFromSet();
58       r.run();               /* site @68 */
59       r = getFromMap();
60       r.run();               /* site @79 */
61     }
62   }
63
64   Runnable getInitHook(){ return task; }
65   Runnable getFromArray() { return taskArray[0]; }
66   Runnable getFromList() { return taskList.get(0); }
67   Runnable getFromSet() { return taskSet.iterator().next(); }
68   Runnable getFromMap() { return taskMap.get("xyz"); }
69 }
```

Listing 4: Walk-through example to demonstrate DODO's approach

6.7.1 Phase 1: Call Graph Construction

DODO first computes the program's call graph. DODO includes two modeling strategies when constructing call graphs: a ***downcast-based*** approach and ***taint-based*** approach. The strategy used is specified via a configuration file.

In both strategies, DODO first extracts the program's entrypoints. These entrypoints are provided as part of the analysis configuration. In this example, the `Main.main(String a[])` is specified as the main method. Therefore, the DODO's worklist is initialized as: $\mathcal{W} = \{\langle Main.main(String\ a[]), \emptyset\rangle\}$. DODO then proceeds to iteratively compute the call graph by traversing each instruction for each method in the worklist.

There are three method invocations on `Main.main()`: two invocations to the

constructors (`<init>`) of `FileInputStream` and `ObjectInputStream` classes followed by a call to the `readObject()` method from the `ObjectInputStream` class. The invocation to `ObjectInputStream.readObject()` is replaced by DODO, *i.e.*, instead of dispatching to Java's implementation of `ObjectInputStream.readObject()`, DODO creates a model (synthetic) method that has the same signature but it is initialized without any instructions. At this stage, the call graph for this program after traversing the main method looks like as shown in Figure 6.5. All these three call graph nodes discovered after parsing `Main.main()` are added to the worklist to be processed (*i.e.*, , `FileInputStream.<init>()`, `ObjectInputStream.<init>()`, and `ObjectInputStream.readObject()`.

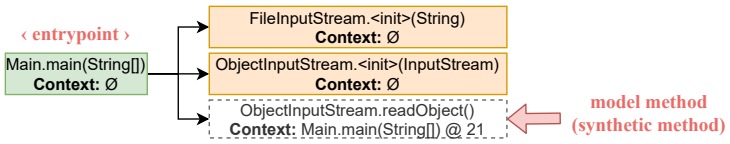


Figure 6.5: Initial call graph after parsing the `Main.main()` method in Listing 4

The instructions that are added to `ObjectInputStream.readObject()` during refinement depend on the modeling strategy used. In the next subsections, we describe how the refinement is performed when using the ***downcast-based*** approach and the ***taint-based*** approach.

Downcast-based Call Graph Construction

This strategy relies on downcast information to infer the possible callback methods that can be invoked during serializaton/deserialization. The snippet in Listing 4 has a downcast to `Config` type on line 6. Consequently, this modeling strategy infers that ***only*** objects from this type will be deserialized.

The resulting call graph is partially[4] shown in Figure 6.6. As indicated, the `readObject()` model method is added with four instructions: an instantiation to `Config` class, an invocation to `Config`'s default constructor, an invocation to `Config.readObject()`, and a return statement to the `Config`'s object.

[4]In these call graphs, we hide other nodes that are called by primordial classes, *e.g.*, from `FileInputStream.<init>()`

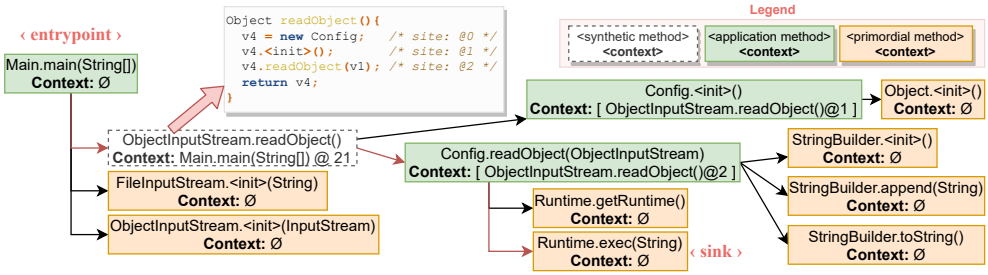


Figure 6.6: Call graph for Listing 4 when using the downcast-based strategy

Taint-based Call Graph Construction

The **taint-based** strategy relies on taint states to infer callback methods that might by invoked during deserialization. Thus, when refining a method model it considers that *all* serializable classes in the classpath could have its callbacks invoked. Since there are multiple Java's default classes that implement callback methods, we consider callbacks that are declared either within the application or extension scope.

By using this strategy, there are two possible callbacks can be invoked: one from `Config` and one from `CacheManager`. As a result, all of its instance fields are marked as *tainted* per the taint introduction rule described in Line 22 (these are highlighted in red on Listing 4). Based on the taint progration rules specified on Table 6.2, variables are then marked as tainted (these variables that are tainted due to propagation are highlighted in cyan on Listing 4).

Recall that tainted invocations (*i.e.*, an instruction such as `obj.aMethod()` in which `obj` is tainted) are handled differently. Whereas the dispatch of non-tainted invocation will follow the rules from the underlying pointer analysis policy, the dispatch for tainted invocations are computed using a modified version of the CHA algorithm (described in Line 22). Therefore, the computed call graph when using the taint-based approach looks like as Figure 6.7[5]. As shown in this image, the model method includes the following instructions: an object instantiation for `Config` as well as `CacheManager`, their constructors invoca-

[5]Due to space constraints, we elide the "getter" calls as well as inner calls from primordial nodes (*e.g.*, `String.isEmpty()`)

tion, and invocations to their callback methods. Finally, the model method returns a value that can either be an instance of `Config` or `CacheManager`. Notice that the phi function (ϕ) added to indicate this possibility.

It is worth mentioning that the context for tainted invocations can be specified by the client analysis. In this example, we use `1-CFA` for tainted invocations.

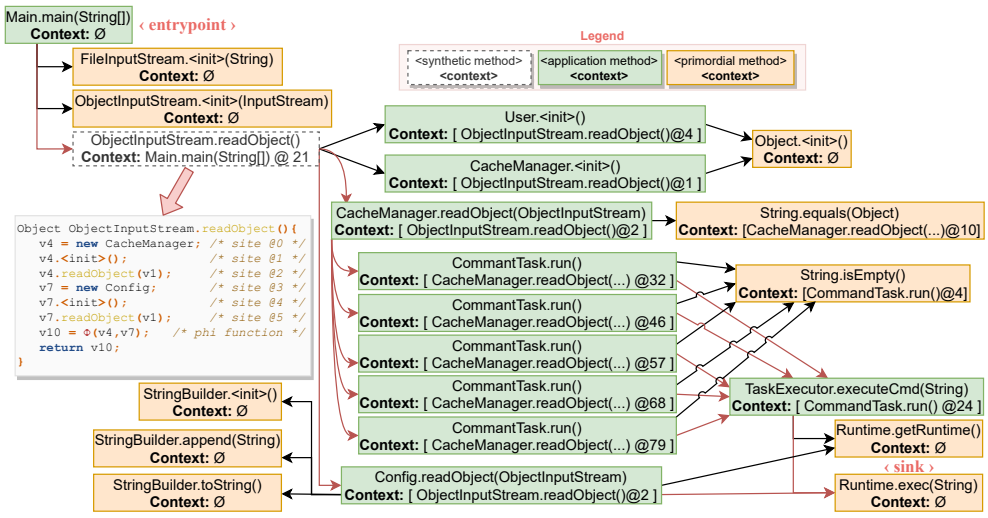


Figure 6.7: Call graph for Listing 4 when using the taint-based strategy

6.7.2 Phase 2: Generating Exploits

First, Dodo computes a *backward slice* [163] that contains all the statements in which the sink is *control dependent*. Subsequently, Dodo extracts the paths between *magic methods* and *sinks*. Dodo finds **1** potential vulnerable path when using the downcast-based approach and **10** vulnerable paths when using the taint-based approach. Figure 6.8 enumerates the possible paths. The path `#1` is the only one that could be found by Dodo when using the call graph computed using the downcast-based strategy. All the paths `#1-#10` are found by Dodo when using the taint-based approach.

After computing these *potential* vulnerable paths from magic methods to

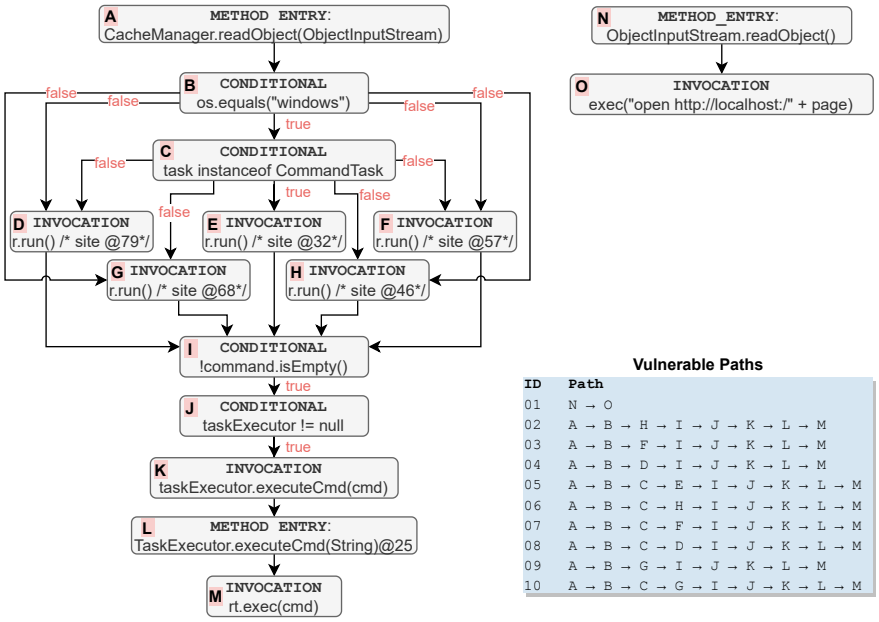


Figure 6.8: Vulnerable paths found by DODO for the program in Listing 4

sinks, DODO extracts their constraints (path constraints and type constraints). Table 6.4 enumerates the path and type constraints extracted for each path. Path `#1` does not have any path constraint because the sink is not preceded by a conditional structure. The other paths (`#2-#9`) do have path constraints that must be true/false for the sink to be executed. As we can observe in this example, the instructions in the `else` block on Listing 4 (lines `52-59`) can be executed in two scenarios `A = false` or both `A = false and B = false`. Thus, DODO treats these as distinct paths (*e.g.*, both paths `#5` and `#6` flows through the `r.run()@32` but the path `#5` considers only `A = false` whereas path `#5` considers both `A` and `B` equals to `false`.

Finally, DODO converts these constraints to the SMT-Lib format [24] that can be handled by Z3 [46]. These constraints are solved by Z3 and deemed as satisfiable. Using the solved constraints, DODO then generates the exploits as shown in Figure 6.9.

Table 6.4: Path and type constraints extracted for the program in Listing 4

#	Path	Path Constraint	Type Constraint
1	N→O	Ø	rt = {Runtime}
2	A→B→C→E→I→J→K→L→M	os.equals("windows") !command.isEmpty()	taskExecutor != null task instanceof CommandTask r@32 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
3	A→B→F→I→J→K→L→M	!os.equals("windows") !command.isEmpty()	taskExecutor != null r@57 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
4	A→B→D→I→J→K→L→M	!os.equals("windows") !command.isEmpty()	taskExecutor != null r@79 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
5	A→B→H→I→J→K→L→M	!os.equals("windows") !command.isEmpty()	taskExecutor != null r@46 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
6	A→B→C→H→I→J →K→L→M	os.equals("windows") !command.isEmpty()	!(task instanceof CommandTask) taskExecutor != null r@46 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
7	A→B→C→F→I→J →K→L→M	os.equals("windows") !command.isEmpty()	!(task instanceof CommandTask) taskExecutor != null r@57 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
8	A→B→C→D→I→J →K→L→M	os.equals("windows") !command.isEmpty()	!(task instanceof CommandTask) taskExecutor != null r@79 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
9	A→B→G→I→J→K →L→M	!os.equals("windows") !command.isEmpty()	taskExecutor != null r@68 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}
10	A→B→C→G→I→J →K→L→M	os.equals("windows") !command.isEmpty()	!(task instanceof CommandTask) taskExecutor != null r@68 = {CommandTask} taskExecutor = {TaskExecutor} rt = {Runtime}

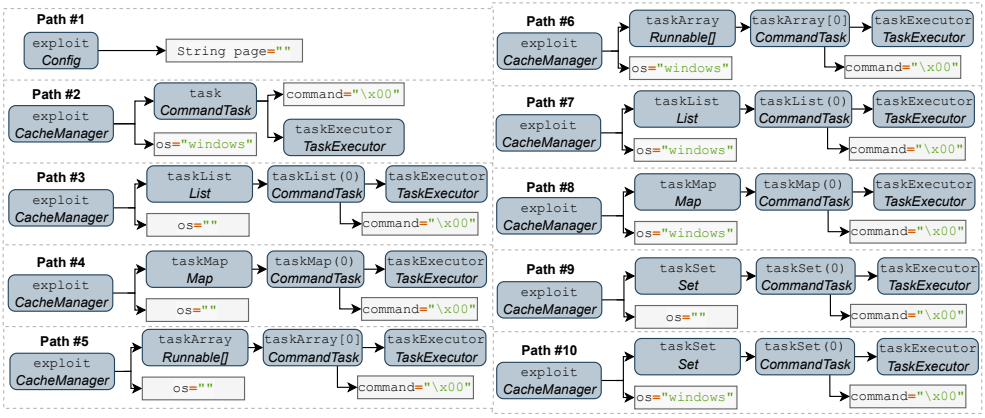


Figure 6.9: Exploits generated by DODO for the program in Listing 4

6.7.3 Phase 3: Instrumentation

In this final phase, DODO saves all the exploits in Figure 2.3 into a file. Subsequently, it executes the program with the paths to these files passed as program arguments. DODO instruments each program's instruction to verify whether (a) the program's execution reaches the sink and (b) the sink contains tainted data (*i.e.*, variables from the exploit).

In this example, all the exploits are able to exercise the vulnerability except for one. The exploits that use the `taskMap` field (*i.e.*, paths `#4` and `#8`. This is due to a limitation on DODO's exploit generation algorithm: when instantiating collections, DODO place one instance on the collection's first index (`0`). For non-indexed collections, such as *maps* it adds an instance with a dummy key equals to "0".

6.8 Answering the Research Questions

This section explains how we answer each research question related with our last work's goal.

6.8.1 RQ6: Does Dodo's call graph algorithm handle object deserialization soundly?

We first aim to verify whether Dodo improves a call graph's ***soundness* with respect to deserialization callbacks** and how it compares with existing approaches. The soundness of a call graph construction algorithm corresponds to being able to create a call graph that incorporate ***all*** possible paths (nodes and edges) that can arise at runtime **during deserialization** [14].

We answer this question using the Java Call Graph Test Suite (JCG) dataset [52]. It was released as part of recent empirical studies [113, 114] to investigate the soundness of the call graphs computed by existing algorithms with respect to particular programming language constructs. The JCG test suite was derived by an extensive analysis of real projects to create test cases that are representative of common ways that projects use these language constructs (*e.g.*, lambdas, reflection, serialization, etc).

In case of serialization-related constructs, the dataset includes **9** test cases for verifying the soundness of call graphs during serialization and deserialization of objects. Each test case is a Java program with annotations that indicate the expected target for a given method call. Table 6.5 provides an overview of the test cases that are available in the JCG dataset and what aspects they aim to probe.

To answer this RQ, we run Dodo using two pointer analysis configurations (`0-1-CFA`, and `1-CFA`) for both our downcast-based and taint-based algorithms. Then, we compare against the same algorithms used in a prior empirical study [113]: Soot (`CHA`, `RTA`, `VTA`, and `Spark`), Wala (`RTA`, `0-CFA`, `1-CFA`, and `0-1-CFA`), Doop (`context-insensitive`), and Opal (`RTA`).

6.8.2 RQ7: Is Dodo useful for finding object deserialization vulnerabilities?

We evaluate Dodo using two open-source projects (Commons File Upload and C3P0) with known disclosed deserialization vulnerabilities. We selected these two projects because their exploits have been discussed by practitioners and are available on the YSoSerial repository [60]. This GitHub repository that contains exploits to trigger well-known "gadget chains" in previously disclosed

Table 6.5: Test cases from the JCG Test Suite [52] and which soundness aspect they aim to verify on a call graph

ID	Description
Ser1	The code serializes an object whose class contains a custom `writeObject` method. It tests whether the call graph creates a node for the `writeObject(...)` callback method that can be invoked by the `writeObject(...)` method from the `ObjectOutputStream` class.
Ser2	Tests whether the call graph has nodes and edges for the `writeObject` callback method under the scenario that the call *may* be invoked if a condition is true.
Ser3	Tests whether the call graph construction algorithm considers inter-procedural flow to soundly infer that the object's `writeObject(...)` callback method will be invoked by the `writeObject` method from the `ObjectOutputStream` class.
Ser4	The code deserializes an object (without performing a downcast) whose class contains a custom `readObject` method. It tests whether the call graph creates a node for the `readObject(...)` callback method that can be invoked by the `readObject()` method from the `ObjectInputStream` class.
Ser5	The code deserializes an object whose class contains a custom `redObject` method. It tests whether the call graph creates a node for the `readObject(...)` callback method that can be invoked by the readObject method from the ObjectInputStream class.. Unlike Ser4, this test case has a downcast to the expected type of the read object.
Ser6	Tests whether the call graph has nodes and edges for the `writeReplace` callback method that will be invoked during serialization.
Ser7	Tests whether the call graph has nodes and edges for the `readResolve` callback method that will be invoked during deserialization.
Ser8	Tests whether the call graph has nodes and edges for the `validateObject` callback method that will be invoked during deserialization.
Ser9	Tests whether constructors of serializable classes are handled soundly. It checks whether the call graph models the runtime behavior which invokes the first default constructor that is not from a serializable superclass.

vulnerabilities.

We answer this RQ by using Dodo to compute the call graphs of these two OSS projects, generate the exploit(s), and verify whether the vulnerability exists.

Chapter 7

Results

This chapter introduces the results obtained throughout the development of this dissertation. Specifically, RQ1 and RQ2 are answered using the CAWE catalog, described in Chapter 4. Research questions RQ3 to RQ5 are answered through the two empirical studies described in Chapter 5. The remaining research questions (RQ6 and RQ7) are answered using a prototype that implements the technique described in Chapter 6.

7.1 Using the CAWE Catalog to Answer RQ1 and RQ2

7.1.1 RQ1: What are the types of tactical vulnerabilities?

Out of the 727 software weaknesses we inspected from the CWE list version 2.9, we found a total of 223 are tactic-related weaknesses (*i.e.*, corresponding to different types of vulnerabilities rooted in the design/implementation of security tactics). Figure 7.1 presents a high-level hierarchical view of these types of tactical vulnerabilities from the CAWE catalog per tactic. Due to space constraints, this figure only shows the higher-level entries (since some tactic-related weaknesses are children of other weaknesses).

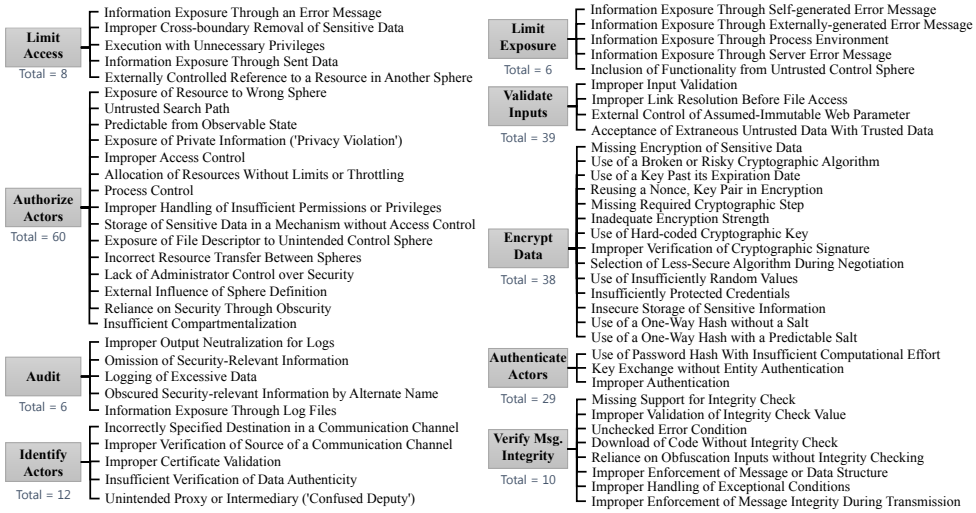


Figure 7.1: High-level overview of the CAWE catalog [125]

Key Finding for RQ1

– There are **223** different types of tactical vulnerabilities (*i.e.*, tactical weaknesses).

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

We answered this question by computing 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 7.1 shows the number of tactical vulnerabilities per security tactic along with a breakdown by the impact type (omission, commission and realization weaknesses).

We noticed 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 may affect multiple security

tactics (*i.e.*, cross-cutting).

Table 7.1: 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

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

7.2 Empirical Studies on Tactical Vulnerabilities

7.2.1 RQ3: What are the most common types of architectural vulnerabilities in real software systems?

We answer this question by identifying the most frequently occurring types of tactical CVEs in each project and their underlying security tactics. Table 7.2 lists the tactical vulnerability types in Chromium, PHP, and Thunderbird, their related tactics, and the total number of CVEs caused by the given vulnerability type.

We found 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. We also verified that many tactical vulnerabilities *in all three projects* were related to the lack of the "Validate Inputs" tactic or its incorrect implementation in code.

One reason as to why input validation problems are more pervasive than other types can be explained by the fact they are not specific to a given software domain. Software will always be provided with inputs, which may be valid or malformed (intentionally by attackers or due to mistakes made by legitimate users). Consequently, input validation issues can occur in any software system leading to *injection attacks*, *data manipulation*, *Cross-Site Scripting*, *etc.*.

Another finding was that Chromium has significantly more vulnerabilities linked to the "Limit Exposure" tactics compared with the other two projects. This is partially due to Chromium being a Web Browser. The "Limit Exposure" tactic aims to minimize the number of entry points through which an attacker can try to breach into the system. This is particularly relevant for a web browser that has built-in processes for interacting with multiple client-side scripts and extensions (add-ons).

In Table 7.2, we observe that PHP's and Chromium's second most common vulnerability type was the *Inclusion of Functionality from Untrusted Control Sphere (CWE-829)*. These results also highlight the importance of tracking external components to understand how these components interacting with the system, if they have any vulnerable code and whether they are used in security-critical operations.

Key Findings for RQ3

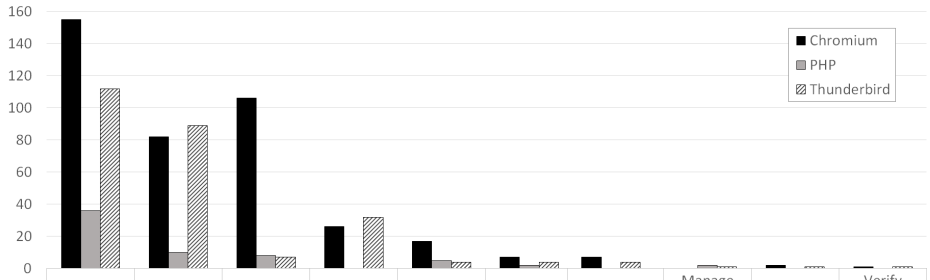
- Improper Input Validation (CWE-20) and Improper Access Control (CWE-284) are the most occurring tactical vulnerability types in Chromium, PHP and Thunderbird.
- The security of the studied projects was compromised by reusing or importing vulnerable versions of third-party libraries. In the case of Chromium such vulnerabilities occurred **106** times, while in Thunderbird and PHP, **7** and **8** times, respectively.

Table 7.2: Most common tactical vulnerability types in **C**hromium, **P**HP, and **T**hunderbird

Tactic	Vulnerability Type	C	P	T	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

7.2.2 RQ4: What security tactics are most affected by architectural vulnerabilities in real software systems?

We answer this question by identifying the tactics associated with the CWE tags of the vulnerabilities across the three projects. We then computed how many times each security tactic was incorrectly adopted in the three systems (Figure 7.2). The majority of the tactical vulnerabilities were related to a failed mechanism that validates inputs consistently and correctly, *i.e.*, the "Validate Inputs" tactic (CWE-20, CWE-59, CWE-74, CWE-77, CWE-79, CWE-89, and CWE-94 in Table 7.2). Failing to validate user inputs can lead to a variety of consequences, such as 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 7.2) are common across 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 7.2: Total number of vulnerabilities (CVEs) per security tactic for each system

Key Finding for RQ4

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

7.2.3 RQ5: What are the root causes of the most frequently occurring types of architectural vulnerabilities?

To answer RQ5, we used our qualitative analysis results. Specifically, we elaborated on the specific root causes that lead to tactical vulnerabilities. For each root cause, we provide an *example*, the *impact* of the associated vulnerabilities on the system's security, and a brief *explanation* of how these vulnerabilities were mitigated.

For "*omission*" or "*commission*" vulnerabilities, our root cause analysis indicates which aspects of the associated security tactics were not chosen (omission) or incorrectly adopted during the software design process (commission). It is worth mentioning that the majority of tactical vulnerability types are cases of "*realization*" weaknesses (see Table 7.2). As such, most of our root causes occurred during the implementation/maintenance of these tactics.

A summarized view of the results is presented Figures 7.3 to 7.7. The detailed description of these root causes, their mitigations, consequences and examples are provided in Appendix A.

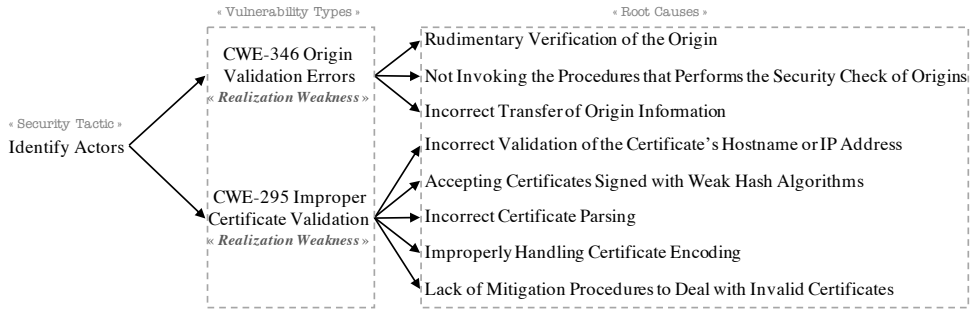


Figure 7.3: Root cause analysis of tactical vulnerabilities related to the "Identify Actors" tactic

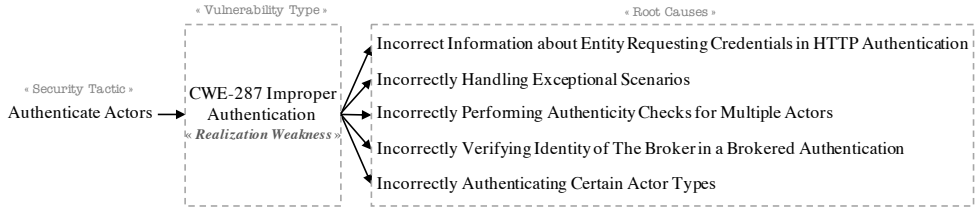


Figure 7.4: Root cause analysis of tactical vulnerabilities related to the "Authenticate Actors" tactic

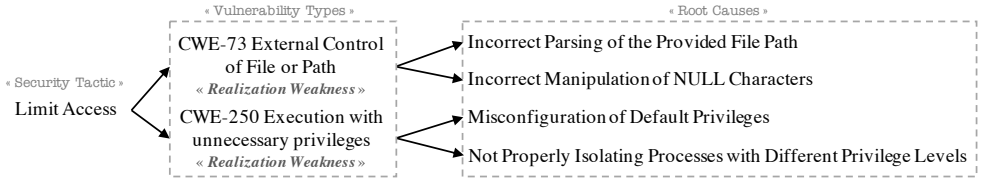


Figure 7.5: Root cause analysis of tactical vulnerabilities related to the "Limit Access" tactic

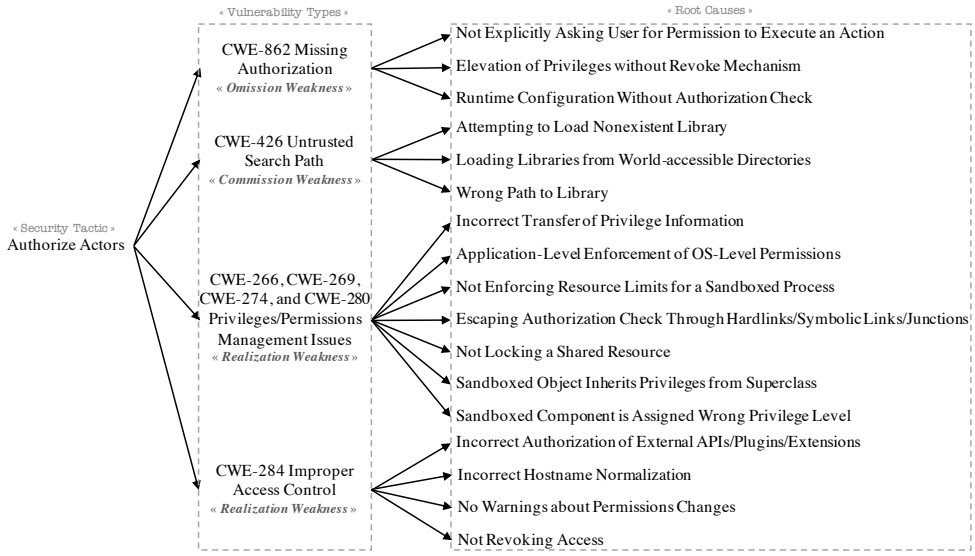


Figure 7.6: Root cause analysis of tactical vulnerabilities related to the "Authorize Actors" tactic

Key Finding for RQ5

– While the studied projects 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 (*i.e.*, *commission* weaknesses).

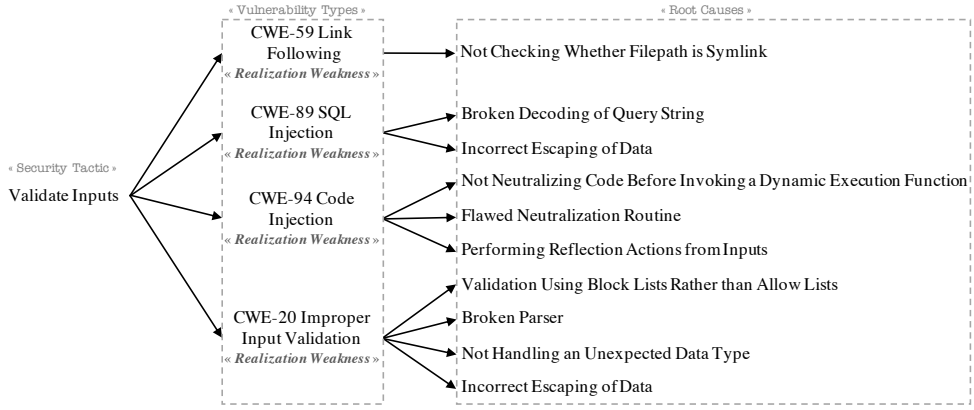


Figure 7.7: Root cause analysis of tactical vulnerabilities related to the "Validate Input" tactic

7.3 Untrusted Object Deserialization Detection

We developed a prototype for Dodo in Java using IBM's T. J. Watson Libraries for Analysis (WALA) [76] to answer our research questions. It allows client analyses to select a pointer analysis method that can either be `0-n-CFA` or `n-CFA`, where `n` is provided. In the next subsections, we investigate how Dodo performs in constructing sound call graphs as well as in detecting object deserialization vulnerabilities.

7.3.1 RQ6: Does Dodo's call graph construction algorithms handle object deserialization soundly?

We examine in this question whether Dodo improves a call graph's ***soundness*** with respect to serialization and deserialization callbacks and how it compares with existing algorithms. The soundness of a call graph construction algorithm corresponds to being able to create a call graph that incorporate ***all*** possible paths (nodes and edges) that can arise at runtime ***during deserialization*** [14].

Using our developed prototype [121, 122], we run Dodo with the Java Call Graph Test Suite (JCG) [113, 114]. Table 7.3 reports the programs in which

each approach soundly inferred the call graph (✓) and the ones it failed to do so (✗). As shown in this table, we built call graphs using two different pointer analysis policies: `0-1-CFA`, and `1-CFA`. For the sake of comparison, this table also includes the same algorithms investigated by Reif *et al.* [113]. The released artifacts of their study [113] includes adapters for constructing call graphs using Soot (`CHA`, `RTA`, `VTA`, and `Spark`), Wala (`RTA`, `0-CFA`, `1-CFA`, and `0-1-CFA`), Doop (`context-insensitive`), and Opal (`RTA`).

Table 7.3: Results from running the test cases from JCG

Approach	Ser1	Ser2	Ser3	Ser4	Ser5	Ser6	Ser7	Ser8	Ser9
Taint-based$_{0\text{-}1\text{-}CFA}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Taint-based$_{1\text{-}CFA}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Downcast-based$_{0\text{-}1\text{-}CFA}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Downcast-based$_{1\text{-}CFA}$	✓	✓	✓	✓	✓	✓	✓	✓	✓
Soot$_{CHA}$	✗	✗	✗	✗	✗	✗	✗	✓	✓
Soot$_{RTA}$	✗	✗	✗	✗	✗	✗	✗	✓	✓
Opal$_{RTA}$	✓	✓	✗	✗	✓	✗	✓	✗	✓
Soot$_{VTA}$	✗	✗	✗	✗	✗	✗	✗	✗	✗
Soot$_{Spark}$	✗	✗	✗	✗	✗	✗	✗	✗	✗
Wala$_{RTA}$	✗	✗	✗	✗	✗	✗	✗	✗	✗
Wala$_{0\text{-}CFA}$	✗	✗	✗	✗	✗	✗	✗	✗	✗
Wala$_{1\text{-}CFA}$	✗	✗	✗	✗	✗	✗	✗	✗	✗
Wala$_{0\text{-}1\text{-}CFA}$	✗	✗	✗	✗	✗	✗	✗	✗	✗
Doop$_{Context\text{-}Insensitive}$	✗	✗	✗	✗	✗	✗	✗	✗	✗

As shown in Table 7.3, Dodo passed ***all*** of the nine test cases. Only three other algorithms partially provided support for callback methods, namely Soot$_{RTA}$ and Soot$_{CHA}$ (2 out of 9) and OPAL$_{RTA}$ (5 out of 9) [113]. The remaining algorithms (Soot$_{VTA}$, Soot$_{Spark}$, Wala$_{RTA}$, Wala$_{0\text{-}CFA}$, Wala$_{1\text{-}CFA}$, Wala$_{0\text{-}1\text{-}CFA}$, and Doop$_{context\text{-}insensitive}$) did not provide support at all for callback methods.

It is also important to highlight that the frameworks that provided partial support for serialization-related features (Soot$_{RTA}$, Soot$_{CHA}$, and Opal$_{RTA}$) use ***imprecise*** call graph construction algorithms (Class Hierarchy Analysis - CHA [47] or Rapid Type Analysis - RTA). Figures 7.8 and 7.9 shows a chart comparison of call graphs' sizes in terms of nodes and edges, respectively. Our approaches (Downcast-based and Taint-based) constructed call graphs with a number of nodes ranging from 549 to 3,786 and number of edges ranging from 944 to 12,199. The other algorithms ranged from 6650 to 7208 (Opal),

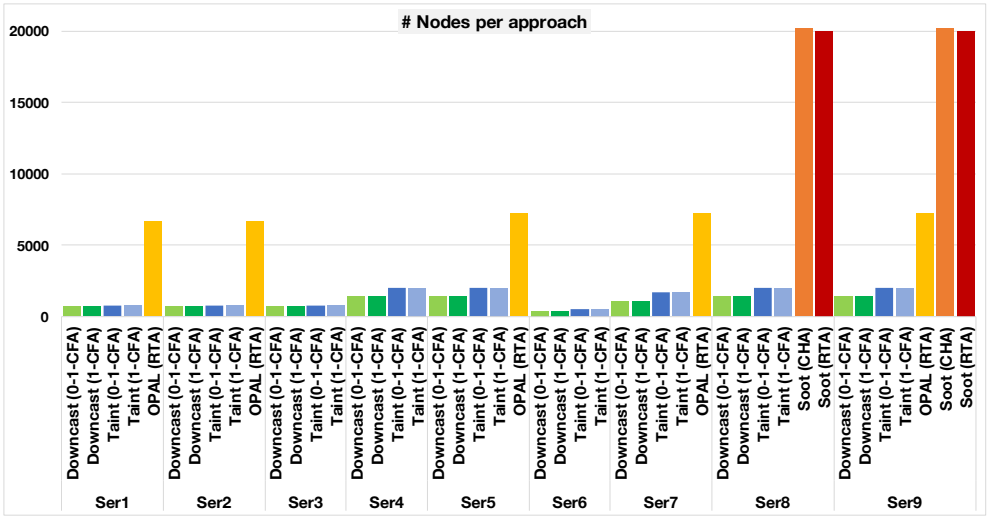


Figure 7.8: Number of call graph nodes per approach

from 20027 to 20168 (Soot) in terms of nodes and from 59,039 to 66,175 (Opal) and 327,530 to 329,815 (Soot) in terms of edges.

As we can infer from these charts, the only call graph construction algorithms used by Soot, and Opal that provided **partial** support for serialization create much larger call graphs (in terms of the number of nodes and edges). Since these algorithms only rely on static types when computing the possible targets of a method invocation, they introduce spurious nodes and edges, thereby increasing the call graph's size. Therefore, our approach enhances the underlying pointer analysis policy in order to strike a balance between improving soundness while not greatly affecting the call graph's precision by adding spurious nodes and edges.

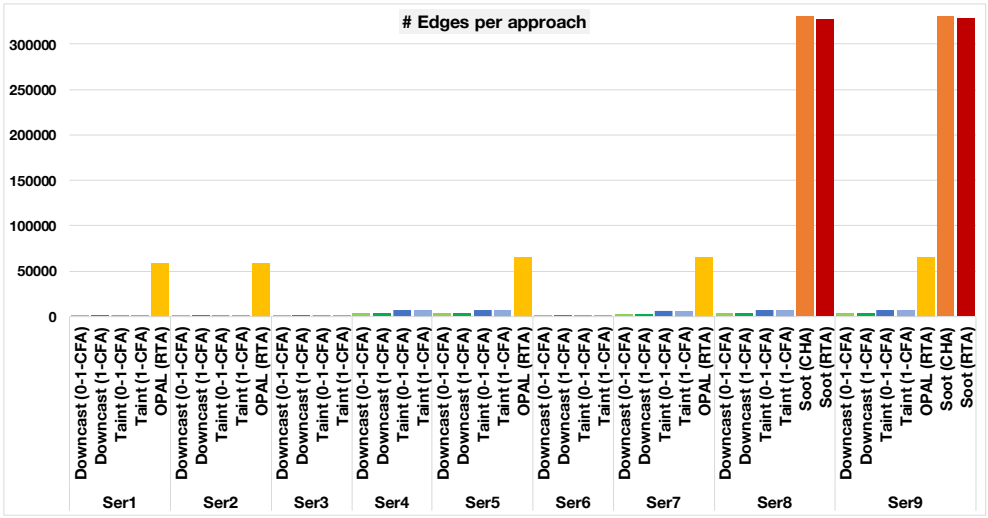


Figure 7.9: Number of call graph edges per approach

Key Findings for RQ6

- Our experiments showed that approach **improved** call graphs' soundness with respect to serialization-related features. It added nodes and edges in the call graph that could infer possible calls that could arise at runtime during serialization and deserialization of objects.
- Our approach (both downcast-based and taint-based) **passed all test cases** in the CATS dataset. Other approaches, namely Soot_{RTA}, Soot_{RTA} passed only 2, and OPAL_{RTA} passed 5.
- The only call graph construction algorithms used by Soot, and Opal that provided **partial** support for serialization used algorithms that only rely on the method's signatures for dispatch. Hence, they created much larger call graphs because they introduced spurious nodes and edges.

7.3.2 RQ7: Is DODO useful for finding object deserialization vulnerabilities detection?

We run DODO using two open-source projects: Commons File Upload (version 1.3.2) and C3P0 (version 0.9.2). These projects contain a total of 157 and 842 classes (including those in dependencies), respectively, as shown in Table 7.4.

Table 7.4: Number of classes in each OSS project and their dependencies

	# Classes	# Classes in dependencies
Commons File Upload (version 1.3.2)	49	108 (commons-io-2.2.jar)
C3P0 (version 0.9.2)	240	602 (mchange-commons-java-0.2.11.jar)

Since both of these OSS projects are libraries, we configured as entrypoints their *magic methods*. That is, all custom deserialization callback methods were treated as entrypoints to construct the call graph in Phase#1 (Figure 6.2). We also included their required dependencies that are listed in Table 7.4. We run DODO in a machine with 16Gb of RAM memory and a 2.9 GHz Intel Core i7 processor.

The results for this analysis is shown in Table 7.5. DODO took `7.32` seconds to perform the static analysis in order to build the program's call graph for the Commons File Upload project. For the C3P0 project, DODO took `11.49` seconds to compute its call graph. In both cases, DODO found **2** potentially vulnerable paths during the Phase 2, in Step 1 (see Section 6.5). After generating the exploits and instrumenting the program, these two vulnerable paths are ***actual*** vulnerabilities.

Table 7.5: Results when using DODO for detecting untrusted deserialization vulnerabilities in two OSS projects

Project	**# Potentially Vulnerable Paths**	**# True Vulnerabilities**	**# Sink Methods**
Commons File Upload (CVE-2016-1000031)	2	2	2
C3P0	2	2	1

Key Finding for RQ7

– By running DODO using two open-source projects with disclosed vulnerabilities, we could found two vulnerable paths in each (true positives).

7.4 Threats to Validity and Limitations

We discuss in this section potential threats to the validity of this work (*i.e.*, *construct*, *internal*, and *external* validity threats to the CAWE catalog and the empirical study) as well as DODO's limitations.

7.4.1 Validity Threats to the CAWE Catalog and the Empirical Study

Threats to the validity can be categorized into *construct*, *internal* and *external* validity threats [118]. Below, we describe each of these validity threat types and how we mitigated them.

- **Construct validity** concerns to what extent the operational measurements are suitable for the purpose of the research and its claims [118]. In our context, one threat is the manual inspection of CWE entries and CVE reports to enumerate tactic-related weaknesses to create the CAWE catalog (Section 4.1) and to identify tactical vulnerabilities in open-source projects (Section 5.2). To mitigate this threat, first we employed a *systematic process* that employed the collection of an extensive list of security tactics as well as their keywords and thorough investigation of CWE entries. Second, we leveraged a variety of software artifacts (*e.g.*, patches, security reports in issue tracking systems, *etc.*) besides CVE reports to identify tactical vulnerabilities in Chromium, PHP, and Thunderbird.

  Despite our efforts to minimize construct validity threats, we acknowledge that our analysis heavily depends on the accuracy of the collected reports (CVEs as well as associated artifacts, as shown in Figure 5.1). Moreover, our study might have missed tactical vulnerabilities that were not disclosed in NVD as well as tactical weaknesses that are not included in the CWE list. Besides, we disregarded vulnerabilities due to incompleteness (*e.g.*, missing a patch or the corresponding defect entry in the issue tracking system was private at the time of the study was conducted).

- **Internal validity** concerns to what extent systematic error or biases were reduced such that causal relations can be inferred [118]. One threat relates to the manual creation of the CAWE catalog. We conducted a peer review to mitigate potential biases and incorrect classification of weaknesses. Similarly, another threat is the manual identification of tactical vulnerabilities in order to observe the nature of tactical vulnerabilities, how they happened and were fixed. We mitigated this threat by performing both top-down and bottom-up vulnerability classification (see Section 5.2.3) and conducting a peer review process.
- **External validity** related to the generalizability of the findings of the work. In our context, there are two threats:
  - We inspected prior vulnerability reports (CVEs) across three open-source projects, which are mostly written in C/C++ and are Internet-based applications. Thus, the results may not generalize to projects with different characteristics. However, it is important to highlight that this work did not aim for *statistical generalization* but rather *analytical generalization*; the three projects were carefully chosen from a variety of software domains and with a high number of known vulnerabilities. Hence, we anticipate that the systems will reflect a normal large-scale software engineering environment. Moreover, we indicated which findings are peculiar to a system and which discoveries apply to all systems while describing our findings.
  - We identified the root causes of vulnerabilities based on the CAWE catalog. We acknowledge that it may not be complete, *i.e.*, it may not include all possible ways that developers can adopt/implement tactics incorrectly. However, it is important to highlight that the CAWE catalog derives from the CWE list, which is a community-established list of possible types of security issues that have been observed and documented in the real world and have been widely used by the security community.

7.4.2 Dodo's Limitations

Although the results discussed in Section 7.3 demonstrated that Dodo can be useful in finding deserialization vulnerabilities, Dodo exhibits the following shortcomings:

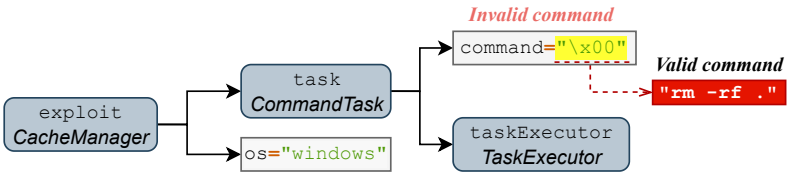


Figure 7.10: DODO's exploit generation limitation

- **Trivial Primitive Values**: DODO's exploit generation component relies on the primitive constraints solved by Z3 (see Section 6.5). These primitive values may not be meaningful. For instance, Figure 7.10 shows the exploit generated for the example described in Section 6.7. This generated exploit has a `command` attribute which is equals to the string `"\x00"`, which is not a valid OS command. To overcome this limitation, we can create a catalog of common attack commands per sink type. For example, creating common OS commands used in exploits. Prior research [151] had investigated the creation of attack patterns to help in solving complex constraints to detect Cross-Site Scripting and SQL/LDAP/XML/XPath injection vulnerabilities. Therefore, we can expect that following a similar approach would also help to overcome this limitation.

- **Exploits involving array/collection objects**: When generating exploits which involves collection, DODO's exploit generation algorithm uses a fixed index/key equals to "0". That is, it assumes that the element at the index/key `0` will be the one used in the exploit. This assumption can result in an exploit that is unable to exercise the sink (as shown in the Demonstrative Example in Section 6.7). We can overcome this problem by performing array/collection access analysis [107] in which we track the indexes in arrays/list as well as keys in maps. This way, DODO can generate an exploit that adds the malicious object at the right index/key.

- **File-based Dynamic Analysis**: As discussed in Section 6.6, DODO's employ a threat model in which it assumes that an attacker has found a deserialization point where an exploit can be injected. Therefore, DODO performs instrumentation over a *driver* class that reads a file that contains the generated exploit. However, certain mitigation techniques involve creating a subclass of the `ObjectInputStream` class to block certain class instances to

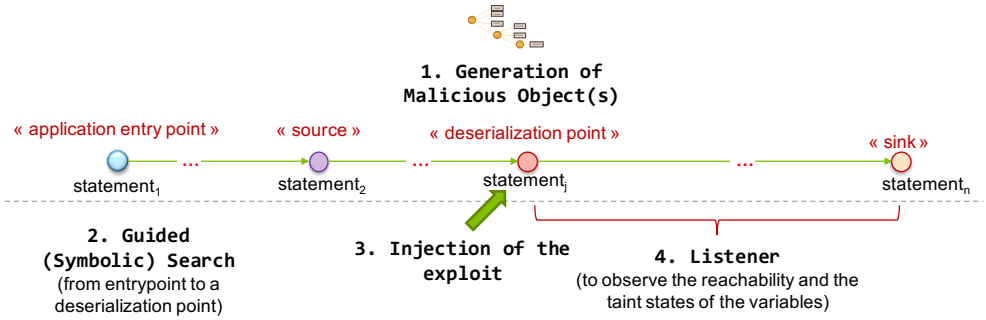


Figure 7.11: Future work: dynamic analysis with guided execution

be serialized [130]. Therefore, using a driver that uses `ObjectInputStream` could report the program as vulnerable when it is not.

This limitation can be solved by implementing a ***guided program execution***, as shown in Figure 7.11. First, DODO generates the exploit ① . Subsequently, it executes the program from its actual entrypoint (*e.g.*, `main()` method) guiding the execution to follow the path that leads to a *deserialization point*, where the object is read from an input stream source ② . At this stage, DODO injects the previously generated exploit in the program at runtime ③ . Next, it monitors the program's behavior to verify the sink's reachability and the taint states of the variables that reach the sink ④ . Based on whether the sink is reached with tainted variables, then DODO reports the program as vulnerable.

- **Not all constraints are automatically solvable**: DODO leverages Z3 [46] to solve primitive constraints (see Section 6.5.2). In this context, there are two limitations. First, our component that converts program instructions to the SMT-Lib format [24] does not provide full support to all string operations in the `java.lang.String` class involving regular expressions. Second, Z3 [46] may timeout for certain constraints [161] since we imposed a limit of 10 minutes. When a constraint times out, DODO generates an *incomplete* exploit in which not all values (primitives/references) are initialized.

Chapter 8

Conclusions

This dissertation presented a Ph.D. research work focused on achieving three goals: (i) to identify common tactical vulnerability types, (ii) to understand real tactical vulnerabilities, and (ii) to develop an approach to detect one tactical vulnerability type caused by untrusted object deserialization.

In the first goal, we aimed to promote the awareness to software engineers and architects of tactic-related weaknesses that can lead to vulnerabilities. As a result, we created a catalog of 223 tactical weaknesses [125], which is currently available to the public in CWE's Website [41]. This catalog could be used as part of security training materials, as well as a guidance during architecture risk analysis. While the omission weaknesses can be used as a checklist to identify missing security tactics, the commission and realization weaknesses can help pinpoint problems in the current architecture design and implementation during architecture risk analysis.

In the second goal, we aimed to better understand these weaknesses related to security tactics, we conducted multiple empirical studies of instances of tactical weaknesses from large and widely used open-source systems [123, 124]. The results are openly available and can be used as starting point for other empirical analysis. Among our findings, we observed that a sheer amount of tactical vulnerabilities were related to the "Validate Inputs" tactic. Moreover, reusing/importing vulnerable libraries can also lead to severe vulnerabilities.

Finally, we developed DODO which encompasses (1) a call graph construction approach that provides full support of serialization-related callbacks and can be used for detecting untrusted object deserialization [121, 122], (2) an exploit generation technique and (3) code instrumentation. Our experiments showed that DODO's computed call graphs are sound with respect to deserialization features. Moreover, using DODO we could successfully detect previously vulnerabilities in two OSS projects (Commons File Upload and C3P0).

8.1 Future Work

There are multiple venues for extending this work:

- While our empirical study identified tactical vulnerability types, their root causes, and mitigations, we need further investigation of their severity and cost associated to fixing them. In addition, we need to further investigate how/why these issues are introduced in order to develop techniques/recommendations to detect and/or prevent them from occurring.
- In Section 7.3, we evaluated DODO's call graph construction algorithm and its ability to find known vulnerabilities in two open-source projects. As future work, we will evaluate DODO's precision, recall and scalability using other open-source projects. Specifically, looking at projects that are not vulnerable and verify how well DODO can find *unknown* vulnerabilities.
- Expand the technique to other programming languages besides Java. We provide in Appendix B a road map to achieve this goal.
- DODO relies on symbolic analysis and constraint solving to generate exploits. Hence, as future work, we aim to compare DODO against a fuzzing-based approach [111].

Bibliography

[1] Getting started · wala/WALA wiki · GitHub. https://github.com/wala/WALA/wiki/Getting-Started. (Accessed on 07/05/2021).

[2] Jython. https://www.jython.org/. (Accessed on 07/03/2021).

[3] Prototyping with python - the fuzzing book. https://www.fuzzingbook.org/beta/html/PrototypingWithPython.html. (Accessed on 07/11/2021).

[4] The ruby programming language on the JVM. https://www.jruby.org/. (Accessed on 07/03/2021).

[5] Which programming language is fastest? the computer language benchmarks game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/. (Accessed on 07/03/2021).

[6] CWE-502: Deserialization of untrusted data. https://cwe.mitre.org/data/definitions/502.html, 2015.

[7] Instrumentation (Java SE 9 & JDK 9). https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/Instrumentation.html, 2021. (Accessed on 07/11/2021).

[8] jenkinsci/workflow-support-plugin- pipeline: Supporting apis plugin. https://github.com/jenkinsci/workflow-support-plugin/commit/a9b071025b5eea33176cefddc1928bce9904c0ef, Jul 2021. [Accessed 07/17/2021].

[9] PHP parser. `https://github.com/nikic/PHP-Parser`, 2021. (Accessed on 07/11/2021).

[10] Self-Protecting Sandbox using SecurityManager · Terse Systems, Jun 2021. [Online; accessed 17. Jul. 2021].

[11] The State of Open Source Security 2020 | Snyk, Jul 2021. [Online; Accessed 22. Jul. 2021].

[12] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

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

[14] Karim Ali, Xiaoni Lai, Zhaoyi Luo, Ondrej Lhoták, Julian Dolby, and Frank Tip. A study of call graph construction for JVM-hosted languages. *IEEE Transactions on Software Engineering*, 2019.

[15] ambionics. ambionics/phpggc - PHPGGC: PHP Generic Gadget Chains. `https://github.com/ambionics/phpggc`, Jul 2021. (Accessed on 07/01/2021).

[16] Iván Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, et al. Avoiding the top 10 software security design flaws. `http://cybersecurity.ieee.org/center-for-secure-design/`, 2014. (Accessed on 10/06/2016).

[17] Ashish Arora, Ramayya Krishnan, Anand Nandkumar, Rahul Telang, and Yubao Yang. Impact of vulnerability disclosure and patch availability-an empirical analysis. In *Third Workshop on the Economics of Information Security*, volume 24, pages 1268–1287, 2004.

[18] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 97–106. IEEE, 2011.

[19] Felix Bachmann, Len Bass, and Mark Klein. Deriving architectural tactics: A step toward methodical architectural design. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2003.

[20] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, 1996.

[21] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. Safewapi: web api misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 507–517, 2014.

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

[23] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (SP 2008)*, pages 387–401, May 2008.

[24] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[25] Adam Barth. RFC 6454 - the web origin concept. 2011.

[26] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser, 2008.

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

[28] Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. Eventually sound points-to analysis with specifications. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

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

[30] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE'11, pages 241–250, New York, NY, USA, 2011. ACM.

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

[32] Larissa Braz, Enrico Fregnan, Gül Çalikli, and Alberto Bacchelli. Why don't developers detect improper input validation?'; drop table papers;–. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 499–511. IEEE, 2021.

[33] Michaela Bunke and Karsten Sohr. An architecture-centric approach to detecting security patterns in software. In *International Symposium on Engineering Secure Software and Systems*, pages 156–166. Springer, 2011.

[34] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. *Electronic Proceedings in Theoretical Computer Science*, 254:15–28, Aug 2017.

[35] Humberto Cervantes, Rick Kazman, Jungwoo Ryoo, Duyoung Choi, and Duksung Jang. Architectural approaches to security: Four case studies. *IEEE Computer*, 49:60–67, 2016.

[36] B. Chess and G. McGraw. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, 2004.

[37] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[38] Angelo Ciampa, Corrado Aaron Visaggio, and Massimiliano Di Penta. A heuristic-based approach for detecting sql-injection vulnerabilities in web applications. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, pages 43–49, 2010.

[39] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 7–12, 2015.

[40] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[41] The MITRE Corporation. Common weakness enumeration, 2014.

[42] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, pages 269–282, 1979.

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

[44] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[45] José Carlos Coelho Martins da Fonseca and Marco Paulo Amorim Vieira. A practical experience on the impact of plugins in web security. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30. IEEE, 2014.

[46] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[47] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[48] Anthony Dessiatnikoff, Rim Akrout, Eric Alata, Mohamed Kaaniche, and Vincent Nicomette. A clustering approach for web vulnerabilities detection. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 194–203. IEEE, 2011.

[49] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:32, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[50] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–10, 2018.

[51] F. Duchene, R. Groz, S. Rawat, and J. Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 815–817, 2012.

[52] Michael Eichberg. Jcg - serializableclasses. `https://bitbucket.org/delors/cats/src/master/jcg_testcases/src/main/resources/Serialization.md`, March 2020. (Accessed on 06/01/2020).

[53] Stefan Esser. Utilizing code reuse/rop in php application exploits. *Black-Hat USA*, 2010.

[54] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, volume 58, 2010.

[55] Qiong Feng, Rick Kazman, Yuanfang Cai, Ran Mo, and Lu Xiao. Towards an architecture-centric approach to security analysis. In *13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 221–230, 2016.

[56] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for java. In *Asian Symposium on Programming Languages and Systems*, pages 465–484. Springer, 2015.

[57] Jose Fonseca, Marco Vieira, and Henrique Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 365–372. IEEE, 2007.

[58] National Security Agency Center for Assured Software. On analyzing static analysis tools. `https://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf`, July 2011. (Accessed on 04/29/2017).

[59] Chris Frohoff. Marshalling pickles: how deserializing objects will ruin your day. `http://frohoff.github.io/appseccali-marshalling-pickles/`, January 2015. (Accessed on 05/26/2018).

[60] Chris Frohoff. frohoff/ysoserial - a proof-of-concept tool for generating payloads that exploit unsafe java object deserialization. `https://github.com/frohoff/ysoserial`, 2018. (Accessed on 05/26/2018).

[61] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. Resolute: An assurance case language for architecture models. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, pages 19–28. ACM, 2014.

[62] Matthias Galster, Mehdi Mirakhorli, Jane Cleland-Huang, Janet E. Burge, Xavier Franch, Roshanak Roshandel, and Paris Avgeriou. Views on software engineering from the twin peaks of requirements and architecture. *SIGSOFT Softw. Eng. Notes*, 38(5):40–42, August 2013.

[63] Tom Gerencer. Extremely useful JVM programming guide for creating stellar software - digital.com. `https://digital.com/web-hosting/java/jvm-programming/`, 2021. (Accessed on 07/10/2021).

[64] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, sys-*

tems, languages, and applications, OOPSLA'97, pages 108–124, New York, NY, USA, 1997. ACM.

[65] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187, 2011.

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

[67] Ian Haken. Automated discovery of deserialization gadget chains, 2018.

[68] Spyros T Halkidis, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and George Stephanides. Architectural risk analysis of software systems based on security patterns. *IEEE Transactions on Dependable and Secure Computing*, 5(3):129–142, 2008.

[69] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[70] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. *ACM SIGPLAN Notices*, 36(5):24–34, 2001.

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

[72] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, 2001.

[73] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 779–790, 2016.

[74] Aram Hovsepyan, Riccardo Scandariato, and Wouter Joosen. Is newer always better?: The case of vulnerability prediction models. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 26:1–26:6. ACM, 2016.

[75] J.H. Howard, D.S. Schiappa, K.E. Ahmed, and K.S. Young. Authentication broker service, October 20 2009. US Patent 7,607,008.

[76] IBM. T.j. watson libraries for analysis (wala). `http://wala.sourceforge.net/wiki/index.php/Main_Page`. (Accessed on 06/05/2020).

[77] Clemente Izurieta and James M Bieman. How software designs decay: A pilot study of pattern evolution. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM).*, pages 449–451. IEEE, 2007.

[78] Daniel Jackson. Lightweight formal methods. In *International Symposium of Formal Methods Europe*, pages 1–1. Springer, 2001.

[79] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[80] Matthias Kaiser. Pwning your java messaging with deserialization vulnerabilities. *Black Hat (White paper)*, 2016.

[81] Eunsuk Kang, Aleksandar Milicevic, and Daniel Jackson. Multi-representational security analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 181–192. ACM, 2016.

[82] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices*, 48(6):423–434, 2013.

[83] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering (ICSE)*, pages 199–209, May 2009.

[84] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. Objectmap: detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*, pages 67–72, 2019.

[85] Benjamin A Kuperman, Carla E Brodley, Hilmi Ozdoganoglu, TN Vijaykumar, and Ankit Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11):50–56, 2005.

[86] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[87] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of java reflection: Literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE'17, pages 507–518. IEEE Press, 2017.

[88] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *International Conference on Compiler Construction*, pages 47–64. Springer, 2006.

[89] Peng Li and Baojiang Cui. A comparative study on software vulnerability static analysis techniques and tools. In *2010 IEEE International Conference on Information Theory and Information Security (ICITIS)*, pages 521–524. IEEE, 2010.

[90] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for java. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, page 27–53, Berlin, Heidelberg, 2014. Springer-Verlag.

[91] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(2):1–50, 2019.

[92] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[93] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, page 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[94] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional, 2011.

[95] Dustin Marx. JDK 11: Beginning of the end for java serialization? `https://dzone.com/articles/jdk-11-beginning-of-the-end-for-java-serialization`. (Accessed on 04/07/2020).

[96] Jane Cleland-Huang Mehdi Mirakhorli. Detecting, tracing, and monitoring architectural tactics in code. *IEEE Trans. Software Eng.*, 2015.

[97] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodríguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[98] M Mirakhorli. *Preserving the quality of architectural decisions in source code*. PhD thesis, PhD Dissertation, DePaul University Library, 2014.

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

[100] Mehdi Mirakhorli, Ahmed Fakhry, Artem Grechko, Mateusz Wieloch, and Jane 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.

[101] Martin 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, pages 234–242, 2014.

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

[103] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[104] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[105] NIST - SAMATE. Source code security analyzers. `https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html`. (Accessed on 04/29/2017).

[106] Oracle. Java object serialization specification - (version 6.0). `https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`, 2010. (Accessed on 04/07/2020).

[107] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(1):65–109, January 2002.

[108] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[109] Marco Pistoia, Satish Chandra, Stephen J Fink, and Eran Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.

[110] Corina S Păsăreanu, Peter C Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 15–26, 2008.

[111] Shawn Rasheed and Jens Dietrich. A hybrid analysis to detect java serialisation vulnerabilities. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1209–1213, 2020.

[112] Adam Rehn. AST instrumentation (examples by language). `https://adamrehn.com/articles/ast-instrumentation-examples-by-language/`, January 2015. (Accessed on 07/11/2021).

[113] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 251–261, New York, NY, USA, 2019. Association for Computing Machinery.

[114] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA'18, pages 107–112. ACM, 2018.

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

[116] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[117] Atanas Rountev, Ana Milanova, and Barbara G Ryder. Points-to analysis for java using annotated constraints. *ACM SIGPLAN Notices*, 36(11):43–55, 2001.

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

[119] J. Ryoo, R. Kazman, and P. Anand. Architectural analysis for security. *IEEE Security Privacy*, 13(6):52–59, 2015.

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

[121] Joanna C. S. Santos, Reese A. Jones, Chinomso Ashiogwu, and Mehdi Mirakhorli. Serialization-aware call graph construction. In *10th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis*, 2021.

[122] Joanna C S Santos, Reese A. Jones, and Mehdi Mirakhorli. Salsa: Static analysis for serialization features. In *Proceedings of the 22nd Workshop on Formal Techniques for Java-like Programs*, FTfJP '20, 2020.

[123] Joanna C S Santos, Anthony Peruma, Mehdi Mirakhorli, Matthias Galster, Jairo Veloz Vidal, and Adriana Sejfia. Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, PHP and thunderbird. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 69–78. IEEE, 2017.

[124] Joanna CS Santos, Adriana Sejfia, Taylor Corrello, Smruthi Gadenkanahalli, and Mehdi Mirakhorli. Achilles' heel of plug-and-play software architectures: a grounded theory based approach. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 671–682, 2019.

[125] Joanna CS Santos, Katy Tarrit, and Mehdi Mirakhorli. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 220–223. IEEE, 2017.

[126] Joanna CS Santos, Katy Tarrit, Adriana Sejfia, Mehdi Mirakhorli, and Matthias Galster. An empirical study of tactical vulnerabilities. *Journal of Systems and Software*, 149:263–284, 2019.

[127] Christian Schneider. Java Deserialization Security FAQ, Apr 2019. [Online; Accessed 22. Jul. 2021].

[128] Christian Schneider and Alvaro Muñoz. Java deserialization attacks. `https://owasp.org/www-pdf-archive/GOD16-Deserialization.pdf`, 2016. (Accessed on 11/15/2019).

[129] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[130] Robert Seacord. Combating java deserialization vulnerabilities with look-ahead object input streams (laois), 2017.

[131] Hossain Shahriar and Hisham Haddad. Object injection vulnerability discovery based on latent semantic indexing. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 801–807, New York, NY, USA, 2016. Association for Computing Machinery.

[132] L. K. Shar and H. B. K. Tan. Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1293–1296, June 2012.

[133] M. Sharp and A. Rountev. Static analysis of object references in rmi-based java software. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 101–110, 2005.

[134] M. Sharp and A. Rountev. Static analysis of object references in rmi-based java software. *IEEE Transactions on Software Engineering*, 32(9):664–681, 2006.

[135] Mikhail Shcherbakov and Musard Balliu. Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web. In *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*, 2021.

[136] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.

[137] Laurens Sion, Katja Tuma, Riccardo Scandariato, Koen Yskout, and Wouter Joosen. Towards automated security design flaw detection. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 49–56, 2019.

[138] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems*, pages 485–503, Cham, 2015. Springer International Publishing.

[139] Yannis Smaragdakis and George Kastrinis. Defensive points-to analysis: Effective soundness via laziness. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[140] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, 2013.

[141] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

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

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

[144] Rahul Telang and Sunil Wattal. An empirical analysis of the impact of software vulnerability announcements on firm stock price. *IEEE Transactions on Software Engineering*, 33(8):544–557, 2007.

[145] The MITRE Corporation. 2021 CWE Top 25 Most Dangerous Software Weaknesses, Jul 2021. [Online; Accessed 22. Jul. 2021].

[146] The PHP Group. Magic methods. `https://www.php.net/manual/en/language.oop5.magic.php`, 2021.

[147] Julian Thomé, Alessandra Gorla, and Andreas Zeller. Search-based security testing of web applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, SBST 2014, page 5–14, New York, NY, USA, 2014. Association for Computing Machinery.

[148] Julian Thome, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Security slicing for auditing common injection vulnerabilities. *Journal of Systems and Software*, 137:766–783, 2018.

[149] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel C Briand. Joanaudit: A tool for auditing common injection vulnerabilities. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 1004–1008, 2017.

[150] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand. Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 198–208, 2017.

[151] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand. An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. *IEEE Transactions on Software Engineering*, 46(2):163–195, 2020.

[152] J. Thomé, L. K. Shar, and L. Briand. Security slicing for auditing xml, xpath, and sql injection vulnerabilities. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 553–564, 2015.

[153] Daniil Tiganov, Jeff Cho, Karim Ali, and Julian Dolby. Swan: a static analysis framework for swift. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1640–1644, 2020.

[154] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN*

conference on Object-oriented programming, systems, languages, and applications, pages 281–293, 2000.

[155] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[156] June Verner, Jennifer Sampson, Vladimir Tosic, Nur Azzah Abu Bakar, and Barbara Kitchenham. Guidelines for industrially-based multiple case studies in software engineering. In *Third IEEE International Conference on Research Challenges in Information Science*, pages 313–324, 2009.

[157] Jan Vitek, R Nigel Horspool, and James S Uhl. Compile-time analysis of object-oriented programs. In *International Conference on Compiler Construction*, pages 236–250. Springer, 1992.

[158] J. Walden, J. Stuckman, and R. Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 23–33, Nov 2014.

[159] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512, May 2010.

[160] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering (ICSE)*, pages 171–180. IEEE, 2008.

[161] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019.

[162] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346, 2013.

[163] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

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

[165] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *International Conference on Database and Expert Systems Applications*, pages 217–231. Springer, 2011.

[166] Y. Zheng and X. Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 652–661, May 2013.

[167] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428, April 2010.

Appendices

Appendix A

Root Cause Analysis (RQ5)

This appendix section elaborates on the details of each root cause identified in our qualitative analysis (see Section 7.2.3). These results are thoroughly explained in a publication derived form this PhD work [126].

A.1 "Identify Actors" Tactic

This tactic was affected by **Origin Validation Errors** (**CWE-346**) and **Improper Certificate Validation** (**CWE-295**) in Chromium and Thunderbird (previously shown in Figure 7.3). These tactical vulnerabilities occurred in these projects as follows:

A.1.1 CWE-346 Origin Validation Errors

This tactical vulnerability concerns issues caused by failing to correctly verify the validity of the source of data or communication. For two Chromium and Thunderbird, it was due to problems related with violations of the *Same-Origin Policy* (*SOP*) [25] and the *Content Security Policy* (*CSP*) [141], which are two complementary security policies commonly applied in Web applications to implement the "Identify Actors" tactic.

In both CSP and SOP, an origin of a Web resource is defined by the *scheme*, *host* and *port* of its URL [25]. On one hand, the *SOP* enforces that docu-

ments/scripts loaded from different sources (i.e. *origins*) do not interact with each other. Under this policy, an application allows scripts/documents to access data from another document/script only if they are from the same origin. On the other hand, the *CSP* is a complementary security control that allows Web servers to specify an "allow list" 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 list are ignored by the application.

Since Chromium is a Web browser, the Same-Origin Policy is needed to ensure that different tabs in the browser do not access other Web page's data. In case of Thunderbird, even though it is a mail client, it uses a Web browser engine (Gecko) to render HTML content in emails as well as the application's user interface itself (by rendering XUL).

Violations to these policies were caused by:

- **Rudimentary Verification of the Origin**: it is caused by an ad-hoc implementation of the checking of the origin of a request according to the software system's goal. In other words, it occurs when developers did not strictly follow the SOP/CSP specification when checking the origin to allow/deny a request.
  Example: Per the CSP specification [141], 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, when the *host* part of a content security policy started with a wildcard (e.g., "*.domain.com"), Chromium was mistakenly matching this host to resources originated from "domain.com" (CVE-2015-6785).
  Impact: It leads to a bypass of the tactic's protection mechanism. This can be used by intruders to steal data (e.g. authentication tokens) or inject code.
  Recommendations: Software engineers should strictly follow existing specifications (e.g. [25, 141]) when implementing the CSP/SOP. They also should careful test their SOP/CSP implementation. Moreover, the risk of inconsistent policy enforcement is reduced when such enforcement is performed by a centralized component. In fact, we saw numerous CVEs where developers performed extensive code refactorings, such as moving

scattered origin checks to a central point to implement these cross-origin regulations uniformly.

- **Not Invoking the Procedures that Perform the Origin Check**: during a cross-origin request to load, execute, or access a resource, the origin verification functions/methods are not invoked.
  Examples: Thunderbird's SOP implementation (CVE-2012-4192) violated the SOP policy by not verifying a request's origin prior to granting access to the properties of the `location` object. This vulnerability was introduced by an incorrect fix to an unrelated defect. Developers removed the invocations to the functions that performs origin check while they were fixing another bug.
  Impact: It leads to a policy-bypass, which results in a compromise of the application's confidentiality and integrity. Attackers are able to read and/or modify data within the software system.
  Recommendations: Extensive testing to ensure that the origin check functions are invoked in all the components that handle cross-origin requests.

- **Incorrect Transfer of Origin Information:** If an application has multiple components that handle cross-origin requests and only one component that performs the origin check, the latter needs to properly receive the origin information to perform the check. We observed instances where an application did not transmit origin information from one process to the forked process or from one object to another.
  Example: Thunderbird's SOP implementation did not expose the final URL to the component performing the origin check when handling Web page redirects (CVE-2008-5507). Thus, attackers could bypass the policy using a malicious JavaScript code that redirected a user to another domain.
  Impact: When using the *SOP* and *CSP* to identify actors, it is necessary to make the assumption that information about the origin is always available when the check is needed. Otherwise, unauthorized actors will be able to access the system's resources due to a policy bypass.
  Recommendations: The origin information must be forwarded to child processes and/or objects. This is especially critical in a chain of URL redirects, where the origin check should be based on the end URL rather than the original URL.

A.1.2 CWE-295 Improper Certificate Validation

Digital certificates are commonly used in Web-based software systems to identity the actors interacting with the system. Each certificate contains multiple fields, including an *expiration date*, *common name* (CN) and the *certification authority* (CA) that issued the certificate. Checking whether a certificate is *valid* is crucial in correctly implementing the tactic. Both Chromium and Thunderbird had realization vulnerabilities in their certificate validation. They were caused by:

- **Incorrect Validation of the Certificate's Hostname or IP Address:** The implementation only checked a portion of the certificate's *hostname* or *IP address* when verifying whether the certificate was issued to the actor making the request.
  Example: Thunderbird (CVE-2010-3170) accepted certificates with a *CN* attribute equals to "*.168.3.48" but it should have not been accepted because CN attributes with an IP address should not have wildcards (*).
  Impact: The identity of the actor making a request is in the the hostname/IP address in a certificate. Hence, incorrect hostname/IP address matching allows remote attackers to spoof trusted certificates, bypassing the tactic.
  Recommendations: Existing guidelines [115] shall be strictly followed by certificate validation routines. Specifically, the *CN* and *subjectAltNames* attributes need to be strictly checked before accepting the connection associated with the certificate.

- **Accepting Certificates Signed with Weak Hash Algorithms:** Certificates are signed with a secure hashing algorithm. We observed instances where the application accepted certificates that were signed using less secure hashing algorithms.
  Example: Chromium accepted SSL connections to a Web site that provided an X.509 certificate signed with either the MD2 or MD4 hashing algorithms (CVE-2009-2973). These hashing algorithms are not strong enough.
  Impact: The application is at risk of man-in-the-middle attacks.
  Recommendations: Less secure hash algorithms are at a higher risk of collision attacks. Thus, a certification validation routine needs to be tested to enforce they do not accept certificates signed with such algorithms.

- **Incorrect Certificate Parsing:** Prior to check the validity of a certificate,

the application needs to *parse* it. We found instances where the certificate parsing was incorrect causing certificate validation problems.
Example: Thunderbird incorrectly handled extra data in a signature that used an RSA key with exponent 3. Remote attackers were able to forge signatures for SSL/TLS and email certificates (CVE-2006-5462).
Impact: Parsing a certificate incorrectly can result in wrong values in the certificate's attributes. This, in turn, negatively affects the certification validation. It can either lead to crashes or misleading the procedure to accept malformed certificates.
Recommendations: Certificates can be provided in different file formats. Therefore, a good strategy to prevent this problem is to have dedicated parsers for each file format that parses these certificates according to the format's specifications.

- **Improperly Handling Certificate Encoding:** certificates can be provided using different encoding. Recognizing a certificate's encoding is crucial to ensure that the information can be properly extracted from the certificate. We observed cases where a certificate's encoding was incorrectly manipulated.
  Example: Thunderbird's implementation incorrectly assumed that any incoming X.509 certificate was encoded using UTF-8 if they were not in ASCII (CVE-2014-1559).
  Impact: It leads to an incorrect parsing of the certificate. Attackers can leverage this mistake to spoof their identity.
  Recommendations: The attributes of a certificate may be encoded using different character sets. Hence, certificate validation procedures should never assume their encoding, but rather infer it from the certificate's attributes.

- **Lack of Mitigating Procedures to Deal with Invalid Certificates:** this occurs when the implementation correctly parses and validates certificates, but it does not correctly handle invalid certificates.
  Example: Chromium's certificate validation implementation did not correctly handle the scenario where an actor provides an invalid certificate (CVE-2014-7948). Consequently, Chromium cached resources from Websites that provided invalid certificates.
  Impact: Attackers can be able to conduct successful man-in-the-middle attacks.

Recommendations: Avoiding this problem can be done by throwing an exception when invalid certificates are received. This exception should be later caught in the code.

A.2 "Authenticate Actors" Tactic

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

- **CWE-287 Improper Authentication:** During their operations, software systems deal with a variety of actors. A popular approach for ensuring a software's security is properly *authenticating all actors interacting with the system*. This is used to check whether an actor is who they say they are (*i.e.*, verify their identity). The following issues contributed to improper authentication issues:
  - Incorrect Information About Entity Requesting Credentials in HTTP Authentication: During HTTP authentication, the user is prompted to provide its credentials. Users then verify the entity requesting their credentials and provide their credentials in case they consider the entity to be trustworthy. Therefore, the application should correctly disply the entity requesting the information.
    Example: Chromium showed to the user the message provided by the server in the "*WWW-Authenticate*" HTTP header. This allowed an attacker to write a message that may 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 crucial aspect to correctly adopt Authenticate Actors tactic is to help users in making decisions. Doing otherwise opens the system to user-assisted attackers, where users are deceived into trusting a fake entity with their credentials.
    Recommendations: The application needs to show the whole entity's identity, including its domain and scheme. Any message provided by the requesting entity needs to be displayed in an unambiguous fashion to prevent deception. We observed that developers discussed a fix to this problem by displaying the server's origin and provided message with different labels.

– **Incorrectly Handling Exceptional Scenarios**: Authentication of actors typically is implemented in a 3-step fashion. First, the application requests the actor's credentials. Second, the actor provides its credentials. Finally, the application verifies whether these credentials are valid. However, the actor may stop the authentication request by canceling it. We found cases in Thunderbird and Chromium where a cancel request was not properly processed.
Example: Chromium still synchronized data even after a user canceled the sign-in request (CVE-2013-6643).
Impact: It can lead to an authentication bypass.
Recommendations: the application should include an error handling mechanism that catches exceptional scenarios involving failures or cancel requests.

– **Incorrectly Performing Authenticity Checks for Multiple Actors**: A tactic's implementation receives multiple authentication requests in parallel, but it only checks the authenticity for one of the actors in these parallel requests.
Example: Thunderbird allowed an attacker to bypass the authenticity check through registering multiple listeners to the same event (CVE-2008-5022).
Impact: Such an incorrect implementation can result in an authentication bypass.
Recommendations: Each received authentication request needs to be queued and processed individually.

– **Incorrectly Verifying the Identity of the Broker in a Brokered Authentication:** Brokered authentication [75] is a security pattern where an authentication broker in charge of granting tokens to actors. We found cases where an implementation to this pattern incorrectly verified whether the obtained token was issued by a trustworthy broker.
Example: After a successful authentication, the OAuth protocol allows URL redirection to a Website. When faced with a chain of redirects, Chromium used the wrong URL when checking the identity of the broker that issued the token in the authentication (CVE-2013-6634). Attackers could leverage this flaw to hijack user sessions.

Impact: It results in an authentication bypass.
Recommendations: An authentication broker implementation needs to handle the exceptional situation where there are multiple redirect scenarios. The last URL is the correct URL in a chain of redirects.

– **Incorrectly Authenticating Certain Actor Types**: Multiple actors (*e.g.*, end users, machines, plug-ins, *etc.*) may interact with a software system. In our study, we found realization weaknesses caused by not authenticating certain actor types.
Example: Chromium permitted plug-ins (an external actor) to be executed without checking their identity (CVE-2013-0910).
Impact: It can lead to an authentication bypass, where these actor types would be able to access the system's data.
Recommendations: Whereas some actor types are more obvious (*e.g.*, users) others might be more subtle and implicit (*e.g.*, plug-ins or extensions). Hence, to effectively prevent this issue, the software system should ensure that all actor types have their identity check prior to being allowed to interact with the software system.

A.3 "Limit Access" Tactic

"Limit Access" tactic concerns with restricting the amount of resources (*e.g.*, CPU, memory, etc) being accessed by an actor. Chromium and PHP had weaknesses in this tactic related to **External control of File or Path** (**CWE-73**) and **Execution with Unnecessary Privileges** (**CWE-250**).

- **CWE-73 External Control of File or Path:** Chromium and PHP perform a file-related operation (*e.g.*, create a compressed archive of a directory) based on a filepath provided as inputs. These requests are intended to be processed within a "safe area", that is, paths outside this logical container should not be accessed. However, we found tactical vulnerabilities that escaped this safe area, which were caused by the following mistakes:

  – **Incorrect Parsing of the Provided File Path:** The application did not correctly parse file paths that contained "*.*" or "*..*" characters in them or that were *symbolic links*.
  Example: Chromium opened/created databases based on a user-provided

filepath. The intended design was to allow the callee to access any file inside a dedicated database directory (isolated). However, Chromium's implementation of this design decision blindly followed *symbolic links* (*i.e.*, it did not check the filepath was a symbolic link to a resource outside the isolated area), resulting in an attacker accessing files from the user (CVE-2014-1715).
Impact: File paths can contain "*dot-dot*" ("." or "..") characters or be symbolic links. These can be leveraged by intruders to escape the safe area, 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:** We found scenarios where the system would not gracefully handle a provided path that contained NULL-related characters (*e.g.*, "\x00" or "%00" or "\0") while implementing the "Limit Access" tactic.
Example: File paths that contained a "\0" character were being incorrectly truncated by PHP (CVE-2015-4025).
Impact: It allows attackers to bypass the tactic and access restricted files/directories.
Recommendations: This problem is common in programming languages with null-terminated strings. This tactical vulnerability type can be avoided by using frameworks/APIs that can detect invalid characters in a file path while implementing the tactic.

- **CWE-250 Execution with Unnecessary Privileges:** Multiple components in Chromium run in different processes (*i.e.*, a multi-process architecture). Each of them have a set of specific capabilities which also dictate their privilege level. Chromium, therefore, has an Inter-Process Communication layer (IPC) layer to enable the communication among processes based also on their privileges. Likewise, PHP's interpreter run scripts with varying privilege levels. In our study, we observed tactical vulnerabilities where processes were executed with more privileges than intended. These were caused

by:

- **Misconfiguration of Default Privileges:** Occurred when the system's privileges default configuration is too loose, providing unnecessary privileges to processes.
  Example: In CVE-2014-0185, PHP's process manager default configuration allowed any user to execute arbitrary scripts with the same permission level of the process manager.
  Impact: An attacker can leverage this vulnerability to perform over-privileged operations.
  Recommendations: Remediating this problem involves following the *least privilege* secure design principle [120]. Hence, default configurations should only allow the minimum read/write access. For example, it can allow read/write to the file's owners and other users in the group (*e.g.*, `660` permission in Unix-based systems).

- **Not Properly Isolating Processes with Different Privilege Levels**: A sandboxed environment restricts which resources get accessed by code running in it. That is, processes in the sandbox only interact with resources/processes within the sandbox. However, we observed tactical vulnerabilities where the application allowed the communication between processes with different privilege levels (*i.e.*, outside the sandbox).
  Example: The `ptrace` system call allows one process to observe/control the execution of another process. Chromium used to allow sandboxed processes to use the *ptrace* command to manipulate the UI process, which allowed an attacker to execute arbitrary code (CVE-2012-2846).
  Impact: It results in privilege escalation, where a lower privileged process relies on a process outside its safe area to perform an operation at a higher privilege level.
  Recommendations: From the patches that fixed the vulnerabilities, we found that there are two ways to remediate this tactical vulnerability type. The first approach is to start the sandboxed process at a permission level that is required for initialization tasks, and then drop these privileges to the minimum after the initialization is completed. A second strategy is to include a sandbox policy in which a sandboxed process cannot invoke security-critical system calls, that is, commands

that are used to manipulate other processes (*e.g.*, `ptrace`).

A.4 "Authorize Actors" Tactic

This tactic had the following 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 is an unintended consequence of not adopting the "Authorize Actors" tactic, in which the application checks an actor's identity before an operation takes place. This weakness was caused by the following:
  - **Not Asking the User for Permission to Execute an Action:** It occurred due to an improper software design, which 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: In CVE-2011-3898, JRE applets were executed in Chromium without user permission.
    Impact: Intruders can inject arbitrary code or conduct other malicious activities without users' awareness.
    Recommendations: Fixing this problem involves requesting user's consent prior to performing security-critical actions (which require user mediation).
  - **Elevation of Privileges Without Revoke Mechanism:** In plugin-based architectures, the system is decomposed into a core component – also called as "host"- and a set of plug-ins (which communicate with the core using APIs). We found vulnerabilities caused by the core component allowing plugins to perform privileged actions without any configuration that could restrict or drop privileges.
    Example: In CVE-2012-0057, PHP allowed the *libxslt* extension to create/write to user files without the option to allow end-users to revoke this privilege.
    Impact: Data tampering and integrity violations.

Recommendations: Mitigating this issue involved creating configuration parameters that enable/disable specific types of operations (*e.g.*, reading files, accessing networks, creating directories, etc).

- **Runtime Configuration Without Authorization Check:** It is caused by changing security-sensitive settings at runtime without any authorization check.
  Example: PHP allowed attackers to overwrite protected configurations using the function "ini_set()" from the PHP language (CVE-2007-5900).
  Impact: Security-critical configuration parameters can be overwritten, leading to privilege escalation.
  Recommendations: The fixes involved identifying a subset of security-relevant configuration parameters that should be read-only at runtime. Hence, prior to modifying a configuration parameter at runtime, the application adopts the "Authorize Actors" tactic to check whether the entry is read-only at runtime.

- **CWE-426 Untrusted Search Path**: Chromium and Thunderbird are applications that load system libraries (*e.g.*, `.dll` on Windows) at runtime. The key issue we observed is that loaded libraries are outside the trust boundaries because they are not under the direct control of the system. Hence, an attacker could replace a genuine library by a malicious one. In fact, we found tactical vulnerabilities where attackers could execute an arbitrary library. They were caused by:

  - **Attempt to Load a Nonexistent Library:** Depending on the operating system version, certain libraries may not exist. We found scenarios where the application's design disregarded the underlying OS version when attempting to dynamically load a library. Consequently, it attempted to load a library that did not exist.
    Example: The "dwmapi.dll" library is only available on Windows versions after Windows XP. Thunderbird attempted to load this library on all Windows versions. Thus, attackers could create a malicious "dwmapi.dll" in the working directory of a machine with an older Windows version and have their library loaded and 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: Developers mitigated this problem by creating a list of libraries per OS version. Then, the tactic's implementation verifies whether such library would exist in the underlying OS prior to load it to the memory.

- **Loading Libraries from World-Accessible Directories:** We found cases where applications attempt to find the desired library to load by searching world-readable directories. These are unsafe directories because an attacker could manipulate.
Example: the Windows API provides a dynamic library search algorithm that also searches the directory from which the application was loaded and the working directory of the parent process. These are potentially unsafe locations because they are not read/write protected. Thunderbird was once vulnerable because it attempted to load the "wsock32.dll" using this algorithm provided by the Windows API (CVE-2012-1943).
Impact: Attackers can execute arbitrary code by placing malicious libraries in unsafe locations.
Recommendations: This problem was fixed using two strategies. The first one was to implement a library-loading algorithm that uses absolute file paths to access the desired library. The second strategy searches only from the system directories (which are read-/write protected by default).

- **Wrong Path to Library:** It is caused by when the application hardcoded the paths to the directories from which libraries/code are loaded/executed, but the hardcoded path is incorrect.
Example: During its installation on Windows machines, Thunderbird executed the "program.exe" located at "C:\" instead of the executable placed in its installation directory.
Impact: Attackers can execute arbitrary code by placing their malicious library/Trojan horse executable file in the hardcoded directory.
Recommendations: The fix involved hardcoding paths per operating system and their versions.

- **CWE-266, CWE-269, CWE-274, and CWE-280 Privileges/Per-**

missions Management Issues:

- **Incorrect Transfer of Privilege Information**: It occurs when the permissions and privileges of an actor is unavailable when the authorization check takes place. We observed cases where permissions/privileges information were not propagated to child process/objects before the authorization took place.
  Example: Chromium lets users specify what Websites can load plug-ins into the Web browser. However, Chromium did not transmit this list of authorized website to the component that performs the authorization check. It resulted in untrusted websites loading plug-ins without user's consent (CVE-2010-2108).
  Impact: authorization bypass.
  Recommendations: The fix is to transmit the required information to the "authorizer" component.

- **Application-Level Enforcement of OS-level Permissions:** System files usually have a list of permissions that specify what users can access them. This enforcement is made by the underlying operating system. Doing this enforcement at the application level is inherently flawed. However, we noticed vulnerabilities in PHP caused by application-level enforcement of OS-level permission.
  Example: The `safe_mode` configuration parameter used to be available in PHP (5.4.0) to implement access control to files and directories on the Web server executing the PHP scripts. The design decision made at the time was to ensure that scripts running in the same server would not access files/directories from each other. This lead to multiple privilege elevation vulnerabilities. In addition, in multiple cases this application-level enforcement was missing (*i.e.*, developers would forget to check whether the *safe mode* was enabled and invoke the access control verification function).
  Impact: Privilege escalation.
  Recommendations: The remediation is to rely on the operating system's access control lists to enforce file/directory access.

- **Not Enforcing Resource Limits for a Sandboxed Process:** When running a sandboxing environment, thresholds are created to specify the maximum of resources a sandboxed process can use. We

found cases where the the threshold was not correctly enforced.
Example: Chrome's Native Client did not enforce limits for data usage. It allowed "row-hammer" attacks (CVE-2015-3335).
Impact: An incorrect threshold enforcement can result in resource exhaustion, where an attacker is able to degrade the system's performance.
Recommendations: The mitigation includes the enforcement of resource usage limits for both logical resources (*e.g.*, user data) to hardware ones (*e.g.*, CPU).

- **Escaping Authorization Check Through Hardlinks, Symbolic Links, or 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:** Realization weakness caused by granting to a lower privileged component more permissions than intended by the design.
Example: Worker processes were executed outside the sandboxed environment in Chromium (CVE-2010-4041).
Impact: Privilege escalation and arbitrary code execution.
Recommendations: The tasks performed by a component dictates the level of privilege they need. Thus, the mitigation involves executing the code in a sandboxed environment set up according to the component's privilege level.

- **CWE-284 Improper Access Control:**
  - **Incorrect Authorization of External APIs, Plug-ins, Extensions, or Libraries**: we observed scenarios where a plug-in based application did not correctly implement authorization for external programs (*i.e.*, APIs, plug-ins, extensions, and libraries).
  Example: Authorization checks were not performed by Thunderbird prior to granting access to local files to Java applets – a plug-in (CVE-2013-1717).
  Impact: data integrity compromises and arbitrary code execution.
  Recommendations: the remediation includes devising an intermediary layer between the application's code and plug-ins, extensions, or libraries. This layer is responsible for performing authorization checks.

– **No Warnings About Permissions Changes:** Permissions changes made at runtime requires user-mediation. We found instances where an application allowed extensions or plug-ins to modify their permissions at runtime.
Example: Malicious plug-ins were able to access the user's camera and Chromium did not show a warning to the user (CVE-2015-3334).
Impact: Attackers can collect user's data without their consent.
Recommendations: The mitigation involves implementing user mediation (via confirmation dialogs) whenever a privilege escalation request is made at runtime.

– **Incorrect Hostname Normalization:** Hostnames are a crucial component used to authorize a user. We noticed incorrect hostname normalization during the authorization check.
Example: A hostname that ended with an extra dot (`.`) mislead Chromium's authorization mechanism, resulting in an authorization bypass (CVE-2015-1269).
Impact: Attackers can bypass the authorization tactic.
Recommendations: The fix involves not accepting hostnames with invalid characters, as well as normalization of hostnames prior to authorization check.

– **Not Revoking Access:** After a resource was used, the tactic implementation did not revoke access to it.
Example: Attackers could access the user's camera by relying on an old session that remained active even after the user had navigated away from the webpage (Chromium's CVE-2014-1586).
Impact: data leakage.
Recommendations: The mitigation involves following the least privilege design principle [120] by dropping privileges when the resource is no longer being used.

A.5 "Validate Inputs" Tactic

From our empirical study, we found the following root causes:

- **CWE-59 Link Following:**
  - **Not Checking Whether Filepath is Symbolic Link:** caused by an implementation that gets a symbolic link as input, but that does not check whether the symlink leads to an unprotected file.
    Example: To store temporary installation files, PHP's setup script utilizes a predictable filename in `/tmp/`. A local attacker may use a symlink to replace the file and overwrite/delete user files (CVE-2014-3981).
    Impact: tampering with user files (*i.e.*, read/overwrite/delete files).
    Recommendations: Before conducting any file-related operation, the remedy entails checking the file's type. The implementation then verifies what file/directory the symlink resolves (if the file is a symlink).
- **CWE-89 SQL Injection:**
  - **Incorrect Escaping of Data:** We observed multiple cases of SQL injection caused by special characters (*e.g.*, quotes, backslashes, *etc.*) in a SQL querey that were not escaped or removed.
    Example: PHP did not escape characters from an external SQL query string before passing it to the *mysqli_fetch_assoc* function (CVE-2010-4700).
    Impact: integrity compromise of a relational databases' data.
    Recommendations: Fixing entails escaping SQL syntax characters (*e.g.*, back/forward slashes, double/single quotes, percentages, *etc.*).
- **CWE-94 Code Injection:** When harmful code segments are produced based on external inputs, this can lead to a tactical vulnerability. In this situation, attackers can provide inputs in the form of code syntax, infecting the software with harmful behavior. This can be used to collect data or disrupt the application execution. According to our findings, the following are the underlying causes of this tactical vulnerability type:
  - **Not Neutralizing Code Before Invoking a Dynamic Execution Function:** it occurs when the application accepts code provided as inputs and execute it without any sort of neutralization/validity check.
    Example: The *valueOf.call* and *valueOf.apply* methods in Thun-

derbird's built-in XML Binding Language (XBL) did not correctly validate input (CVE-2006-1733). It allowed attackers to execute arbitrary code.
Impact: Arbitrary code execution.
Recommendations: Interpreted programming languages may load and execute code from a string(*e.g.*, `eval()` in JavaScript). To enforce that only safe commands are executed, the application needs to parse the string and remove any unsafe commands from it.

- **Flawed Neutralization Routine:** The neutralization routine does not cover all the possible types of unsafe commands.
Example: In CVE-2012-3980, Thunderbird accepted JavaScript code as input and directly invoked the *eval* function without neutralizing any unsafe commands provided in the input.
Impact: Memory corruption and arbitrary code execution.
Recommendations: Neutralization of malicious commands and verification of the call's context (*i.e.*, from a lower / higher privileged actor) prior to invoking the code execution function (*e.g.*, `eval()`).

- **Performing Reflection Actions from Inputs:** Many programming languages allow reflection, a feature that allows you to load classes at runtime. Similar to "*Not Neutralizing Code Before Invoking a Dynamic Execution Function*", the application was not neutralizing the input before performing reflection operations.
Example: Firefox's JavaScript permitted that attackers could obtain a constructor from XBL compilation scope through leveraging a reflection call.
Impact: Arbitrary code execution, privilege escalation and denial of service.
Recommendations: Remediating this problem can be achieved via: neutralizing unsafe constructs in the input or hiding reflection calls from outside its trust area.

- **CWE-20 Improper Input Validation:** Software receives user inputs, which may have different requirements related to its type, size, boundary values, *etc.*. We found several scenarios where the input was not validated based on its requirements. It was caused by the following problems:

– **Validation Using Block Lists Rather than Allow Lists:** *Block lists* specify inputs that shall not be accepted by the system. They are inherently flawed, as it is difficult to create a comprehensive list of all possible malicious inputs. We found cases where the application relies on *block list* instead of an *allow list*, which dictates how a trustworthy input looks like.
Example: Chromium (CVE-2009-3931) had a block list of files to which they would deny their download. This list did not include potentially dangerous extensions (*e.g.*, `.mht`, and `.mhtml`, which are automatically executed by Internet Explorer 6).
Impact: arbitrary code execution.
Recommendations: It can be remediated by refactoring the input validation routine to rely on *allow lists*, which enumerate the accepted input types in the code. Although the problem can also be fixed by adding the missing malicious data into the block list, this approach is not the safest (as it is difficult to think upfront all the possible ways an attacker may try to tamper with the input).

– **Not Handling an Unexpected Data Type:** Inputs can have multiple types. We found cases where an application would not properly handle the input's type (*i.e.*, "Type Confusion").
Example: The input validation procedure in PHP incorrectly assumed that the input was an array (CVE-2015-4148).
Impact: Denial of service and application crashes.
Recommendations: The fix involves checking the actual data type from the input provided (rather than assuming it).

– **Broken Parser:** When the application receives complex data structures, it needs to parse it. We found cases where an incorrect parser implementation lead to vulnerabilities.
Example: Chromium's parser did not correctly handle URLs starting with "blob: " followed by another URL and a long username (CVE-2014-7899).
Impact: crashes and denial of service.
Recommendations: the remediation involves extensive testing of the parser to ensure it can correctly parse data structures received as input against a data schema.

– **Incorrect Escaping of Data:** We found scenarios where the application does not correctly neutralize "control" characters in an input.
Example: Chromium allowed that the value within an `href` attribute to be rendered as regular HTML entities (CVE-2015-6790). Attackers could inject malicious code in this attribute in order to steal data or CSRF tokens.
Impact: it can lead to arbitrary code execution and memory corruption.
Recommendations: The fix can be made by verifying the context in which the data will be used and apply escaping routines according to it. As an example, in an HTML rendering context, we need to escape HTML entities (*e.g.*, "" is escaped as "<span>") prior to rendering it to a Web page.

Appendix B

Extending DODO to other Languages

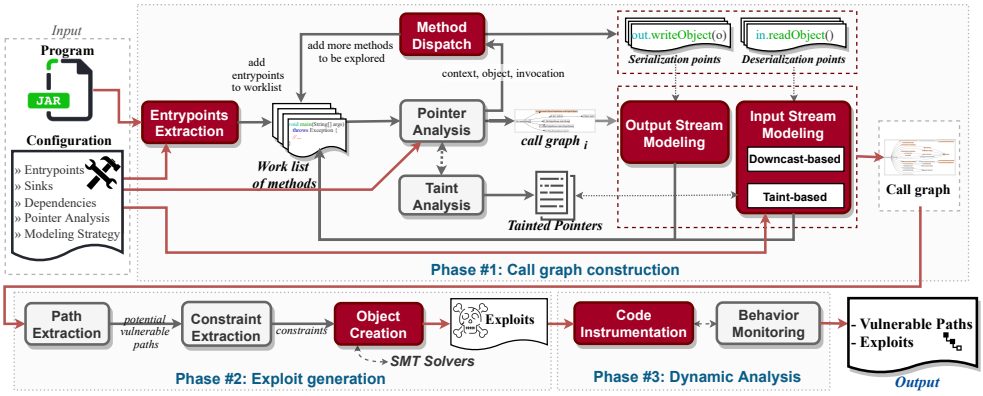


Figure B.1: Components in DODO that needs modifications to be extended to other programming languages (highlighted in red)

Untrusted object deserialization is a vulnerability that can affect multiple programming languages, such as Ruby and PHP [6]. Thus, this appendix discusses technical aspects for extending DODO to find untrusted object deserialization vulnerabilities in programming languages other than Java. These discussions do not intend to be *exhaustive* but rather *exploratory* by show-

ing the crucial components in our approach that require modifications. These components are highlighted in red in Figure B.1.

B.1 Java Bytecode as Input

DODO takes as input a compiled version of the program, *i.e.*, JVM bytecode. To support other languages, there are two approaches one can follow:

- **Alternative #1**: Certain programming languages (*e.g.*, Scala), which are JVM-hosted, can also be compiled to JVM bytecode [63]. Thus, DODO would be able to compute its call graph and subsequent analysis (*i.e.*, exploit generation and instrumentation). Similarly, Python and Ruby can also be compiled to JVM bytecode using Jython [2] and JRuby [4], respectively, which are a JVM-based implementation of these languages. However, prior research has shown that call graphs extracted from these JVM-based implementations are *unsound* [14]. The unsoundness is caused by the extensive use of `invokedynamic` instruction in the generated bytecode which, in turn, makes the analysis difficult to be performed statically.
- **Alternative #2**: DODO is implemented on top of WALA [76]. Currently, WALA (or other static analysis tools built on top of it) provides front-ends for a few programming languages, such as JavaScript [76], Python [50], and Swift [153]. Hence, a second alternative is to develop a front-end for their target programming language on top of WALA Common Abstract Syntax Tree (CAst) System [1]. This way, the front-end parses the source code to generate a method's IR that implements WALA's interfaces.

Although *alternative #1* would be the one that would least incur implementation efforts (*i.e.*, it does not require implementing new front-ends for WALA), they may render call graphs that are inherently unsound [14]. However, it is important to highlight that prior study of the soundness of call graphs for JVM-hosted languages/implementations used 10 programs from the Computer Language Benchmarks Game (CLBG) suite [5]. Hence, the study did not focus on the typical usages for object deserialization.

Therefore, one has to perform a trade-off analysis to find whether the `invokedynamic` instructions in the underlying bytecode affects the part of the

program of interest (*i.e.*, paths through callback methods). Then, finally decide whether to develop a WALA front-end.

B.2 Signature-Based Method Dispatch

As described in Line 23, DODO relies on class hierarchy information when computing the dispatch for tainted invocations. This is possible because Java is a statically typed language, *i.e.*, it requires that each variable are assigned a type and the type checking is enforced at compile time. However, dynamically typed languages, *e.g.*, Python and JavaScript, performs type checking at runtime (*i.e.*, variables can be declared without a type). Consequently, variables can be assigned an object of unrelated types [65,162]. For this reason, the dispatch for tainted instance invocations such as `x = o.g(`a_1`,`a_2`,...,`a_n`)` needs to compute the possible dispatches (call targets) based on the declared target g (*i.e.*, g's method signature).

B.3 Language-Specific API Modeling

Recall that DODO employs API modeling to abstract the inner workings of (de)serialization APIs (see Section 6.4.2). The current approach is tailored for modeling the classes `ObjectInputStream` and `ObjectOutputStream` from the `java.io` package according to the callbacks that are invoked (shown in Figures 2.1 and 2.2). Therefore, when adapting DODO to a different programming language, one has to change the API modeling component to match the specification for the target API.

For example, the default functions in PHP for writing/reading objects are `serialize()` and `unserialize()`, respectively [53]. The language has the following magic methods: `__sleep()`, `__wakeup()`, `__serialize()`, and `__unserialize()` [146]. During object serialization, the mechanism verifies whether the class has the `__sleep()` or `__serialize()` functions, in which case one of them gets invoked prior to the object serialization. Similarly, when reading an object, `__wakeup()` or `__unserialize()` functions are invoked if they are present on the object's class being deserialized. Therefore, our API modeling component has to abstract the behavior for these magic methods.

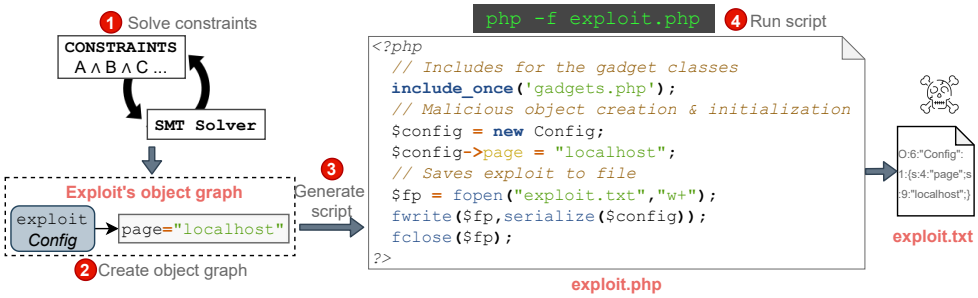


Figure B.2: An example of exploit generation for the PHP language

B.4 Object Generation

The exploit generation component is implemented in Java using reflection. It does so by first loading the project's JAR file using a `java.net.URLClassLoader` instance. Subsequently, it creates a malicious object whose class is in the newly loaded class(es) in the JAR file. However, for other programming languages, this implementation has to be modified to use the target language's serialization protocol.

One possible implementation is to modify this component to generate a *script file* in the target language that creates the malicious object. Figure B.2 shows an example of how this could be implemented for PHP. First, DODO would solve the extracted constraints using a solver (*e.g.*, Z3 [46]) to create an *object graph* in the memory. The object graph is simply an abstraction on how the object should look like in terms of fields and their values. From this object graph, then DODO generates a script (`exploit.php`) which instantiates the object represented in its graph. Subsequently, it runs this script by issuing the `php -f` command. This command executes the script which makes the exploit to be saved in a text file (`exploit.txt`).

B.5 Code Instrumentation

The code instrumentation component in DODO performs online instrumentation using Java Path Finder (JPF) [69, 110], a model checking tool. Thus, DODO implements a listener which directly monitors the instructions and data

being passed across instructions as the program is deserializing the exploit. Since Java Path Finder can only analyze Java programs, this component has to be re-implemented to use a different tool/approach for instrumentation.

In this context, one could develop an *online* or *offline instrumentation* approach [34]. For *offline instrumentation*, execution traces are captured and stored instead of directly monitoring the executed instructions. This can be achieved by performing AST-based instrumentation, in which the program's AST is transformed to a modified version with "probing instructions" to capture events of interest (such as methods invoked during execution). Subsequently, the captured execution trace is analyzed according to the heuristic described in Section 6.6. As an example, we can use PHP Parser [9] to perform code transformations to log method calls and then execute this transformed version of the program [112].

In the case of *online instrumentation*, the approach monitors the program as it is being executed and reports whether the system is vulnerable or not (*i.e.*, the sink is executed with exploit-provided values). While JPF [69] and Java agent [7] are tools/APIs that can be used for instrumenting Java bytecode, other languages have different APIs/tools available. For instance, Python programs can be instrumented at runtime by using the `inspect` and `sys` modules [3].