

# AN EVALUATION OF HOW WEB FRAMEWORKS SUPPORT DEVELOPERS TO BUILD SECURE APPLICATIONS

Kim Leppänen

Master's thesis in Software Engineering  
Instructor: Ivan Porres Paltor  
Faculty of Science and Engineering  
Åbo Akademi University  
2024

## Abstract

An increasing number of applications are being built for the web. For this task, developers typically use a number of different frameworks to ease and speed up the development. Frameworks can make complex problems easy by providing tools, patterns and abstraction layers, but can frameworks help developers in one often forgotten area: the application's security?

Vulnerabilities in web applications can originate from many different sources. A vulnerability might exist due to improper implementation, but also due to poor design. A feature that has been designed in an insecure manner, cannot necessarily be made secure even with a perfect implementation.

The purpose of this thesis is to evaluate how modern web frameworks can help developers build more secure applications. What aspects of security is something a framework can independently manage, what kind of tools can a framework provide the developer to guide them build secure software and what parts of the security is such that a framework cannot manage and is left solely as the responsibility of the developer.

An example application using Vaadin Flow and Spring Boot frameworks, both modern Java based tools, was written for this thesis. The example application was then security tested for vulnerabilities described in the OWASP Top Ten list. The purpose of the evaluation was to understand, which vulnerabilities were directly mitigated by the frameworks and which aspects of the application security is something the developers must understand and mitigate themselves.

This thesis found that only a few explicit technical vulnerabilities were mitigated by the frameworks, while some of the vulnerabilities were such that frameworks could *guide* the developers by providing tools, but could not ensure full mitigation of the vulnerabilities. To properly secure an application, collaboration is needed between software, network, system, and security engineers, and good DevSecOps practices need to be implemented.

## Table of Contents

1 INTRODUCTION .....	4
1.1 Open Web Application Security Project - OWASP .....	4
1.1.1 OWASP Top Ten .....	5
1.1.2 OWASP Testing Guide .....	5
1.1.3 OWASP's Application Security Verification Standard (ASVS) .....	5
1.2 Scope of the thesis .....	6
2 EXAMPLE APPLICATION .....	7
2.1 Functional requirements .....	8
2.2 Non-functional requirements .....	9
3 APPLICATION DESIGN .....	9
3.1 High-level architecture .....	9
3.2 Data model .....	10
3.2.1 Users and roles .....	11
3.2.2 Course information .....	11
4 IMPLEMENTATION .....	12
4.1 Vaadin Flow .....	12
4.2 Spring Boot .....	13
4.3 Presentation layer and the user interface .....	13
4.4 Business logic layer .....	14
4.5 Data access layer .....	14
5 VULNERABILITY REVIEW .....	15
5.1 A01:2021-Broken Access Control .....	15
5.1.1 Insecure Direct Object References .....	20
5.1.2 Cross-Site Request Forgery (CSRF) .....	26
5.2 A02:2021-Cryptographic Failures .....	32
5.2.1 Framework level mitigation .....	33
5.3 A03:2021-Injection .....	33
5.3.1 SQL injections .....	35
5.3.2 Cross-site scripting - XSS .....	40
5.4 A04:2021-Insecure Design .....	46

5.4.1 Framework level mitigation .....	46
5.5 A05:2021-Security Misconfiguration .....	47
5.5.1 Analysis of frameworks' default configuration.....	48
5.6 A06:2021-Vulnerable and Outdated Components .....	50
5.7 A07:2021-Identification and Authentication Failures .....	51
5.7.1 Framework level mitigation.....	51
5.8 A08:2021-Software and Data Integrity Failures .....	54
5.8.1 Software integrity.....	54
5.8.2 Data integrity .....	55
5.9 A09:2021-Security Logging and Monitoring Failures .....	57
5.10 A10:2021-Server-Side Request Forgery .....	58
5.10.1 Mitigation strategies.....	60
5.10.2 Framework level mitigation strategies .....	61
6 DISCUSSION AND CONCLUSIONS .....	62
7 REFERENCES .....	67
8 SVENSKT SAMMANDRAG.....	80
9 APPENDIX A.....	83

# 1 INTRODUCTION

In today's world, we see an increasing number of applications being written for the web. Even such applications as have traditionally been considered desktop software have been transformed into web applications. Google has released several web versions of applications previously only existing as desktop software. A good example is Google's project, which brings traditional office tools to a web environment, allowing the creation and editing of text documents, spreadsheets, and presentation documents with your browser. Microsoft responded to this by creating Office 365, their web equivalent for the traditional version of their office suite.

We see web applications used for many aspects of our day-to-day lives, from hosting sensitive information such as personal, financial, or even healthcare data to applications we use for controlling devices such as smart doorbells (ring.com) or other IoT devices. Even though these applications are made to make our lives more comfortable and convenient, they also come with a dark side: they have become lucrative targets for hackers.

Applications are becoming more complex, while development teams are expected to deliver features at an increasing pace. This combination makes it difficult to ensure that the applications we develop are built securely. Unfortunately, the priorities of software development are often in the functionality, the appearance, and the usability, values that directly appeal to the end user. Security, however, often comes as a secondary priority. Sometimes security is considered a software feature that could be applied to the application at the last stages of development. In fact, security should be taken into account early in the design phase before any code has been written. If security has not been taken into account early enough in the development, correcting security issues to an almost completed software can be next to impossible, or at least expensive and time-consuming.

## 1.1 Open Web Application Security Project - OWASP

Open Web Application Security Project, commonly known as OWASP, is the world's largest nonprofit organization aiming to improve software security. It is a global organization that provides industry-leading educational and training conferences around secure software development (OWASP, n.d.-a). OWASP maintains and develops several community-led open-source projects that provide the community with valuable resources used by developers worldwide to secure their applications. For the structure and foundation, this thesis will rely on three of OWASP's projects.

### **1.1.1 OWASP Top Ten**

The OWASP Top Ten project was created to raise awareness of the most critical security risks found in modern web applications. The project's purpose is not to release a comprehensive list of all vulnerabilities but to release a document listing vulnerabilities the security community has a consensus of being the ten most critical vulnerability categories found in web applications. The published document functions as a starting point for developers and organizations that want to improve the security of their applications (OWASP, n.d.-b). The project started in 2003 and, over the years, has achieved a pseudo-standard status, being used as a baseline for application security (OWASP, 2021a). The newest version of the document was released in 2021.

The OWASP Top Ten list will be used as a baseline for the security assessment done in this thesis.

### **1.1.2 OWASP Testing Guide**

When building software, results are typically tested for defects. The functionality goes through acceptance testing, verifying that the application's business logic works in the way intended and that the application accurately performs the designed functions. Security is often not a visible feature, yet the software we build is expected to be secure. Secure software does not happen by accident; it results from careful design, thoughtful implementation, and rigorous and systematic security testing.

In order to have a more systematic approach based on the principles of engineering and science, OWASP has created a program called the Web Security Testing Guide (WSTG). The Web Security Testing Guide provides a practical guideline on *how* to test an application for common web vulnerabilities (OWASP, n.d.-e). This thesis will use OWASP's Web Security Testing Guide as a foundation for testing an example application against the described vulnerabilities.

### **1.1.3 OWASP's Application Security Verification Standard (ASVS)**

While OWASP's Web Security Testing Guide gives guidance on *how* to test an application, it does not dictate *what* should be tested. The Application Security Verification Standard is a project released by OWASP, a standard with a list of requirements applications should fulfill to be considered secure.

Not all applications need to be equally secure. Implementing rigorous security controls comes with a high monetary cost, which might not be sensible for all applications. Security is a matter of risk assessment: which risks are we

willing to accept, and which risks need to be mitigated? ASVS acknowledges this principle by defining three different security levels, each increasing in depth. Level 1 is meant for low-risk applications that do not handle any sensitive data. Level 2 is intended for applications that contain sensitive data, while level 3 should be used for critical applications (OWASP, 2021d, p. 11).

This thesis will use ASVS version 4.0.3, level 1 when considering what security aspects to consider in the example application.

## 1.2 Scope of the thesis

This thesis aims to review how modern software development frameworks help developers create secure software. While frameworks can do only so much, the responsibility of securing an application cannot be left solely to the frameworks being used. Ultimately, it is the developer's responsibility to ensure design and implementation take security into account from the get-go and is not an afterthought.

The thesis will rely on OWASP's Top Ten, WSTG, and ASVS projects. These projects were chosen due to the status OWASP has gained in the security community. The Top Ten and ASVS projects are considered de facto baselines or standards in many contexts, even though competing projects exist.

As a basis for the thesis, a simple web application has been built using modern frameworks commonly used by web developers. The target is to review the implementation against common web application vulnerabilities outlined in OWASP Top Ten. The evaluated questions are

- Which vulnerabilities are mitigated by the used frameworks and require minimal knowledge from the developers to use securely?
- For proper mitigation, which vulnerabilities need more active design or implementation-specific decisions from the developers?
- How do the default settings of the chosen frameworks correlate to security standards?

The review for vulnerability mitigation provided by frameworks will be grouped into four categories.

1. *Mitigated out-of-the-box by the framework.* The framework mitigates these vulnerabilities in such a way that it requires no active interaction from the software developer.
2. *Mitigated by the framework through configurations.* Vulnerabilities falling into this category can be mitigated by the frameworks used but

not through default configuration. Using the default configuration would make the application vulnerable.

3. *Helpers are provided by the framework.* The frameworks do not directly mitigate the vulnerabilities, but they give the developers helpers that guide or ease the implementation of secure solutions.
4. *No mitigation support is provided.* These vulnerabilities need to be mitigated by the developers through proper design and implementation.

The application's source code is only one aspect of secure development; the overall security is also impacted by the infrastructure of the development and deployment environments. While equally important, this thesis focuses on the frameworks used, and thus, this thesis will not cover, for example, attacks against servers with improper configurations or attacks made through vulnerabilities in protocols or cryptographic algorithms. These vulnerabilities are not application-level vulnerabilities and, hence, are not affected by framework choices and thus will be left out of the scope of this thesis. What is covered could be considered as the aspects the developer has to consider when doing the actual implementation of the software, not what the server administrator does when deploying the application or when setting up the deployment environment. The focus is on weaknesses in an application due to improper or insufficient technical implementations. For example, SQL injections (discussed in chapter 5.3.1) are discussed because an attack is often possible due to improper ways of constructing and executing queries. At the same time, vulnerabilities in third-party libraries are not evaluated in-depth, as those are often mitigated through processes and tools in the CI/CD pipeline.

## 2 EXAMPLE APPLICATION

For this thesis, an example web application has been built using a modern technology stack. The domain for the example application was chosen in such a way that it will demonstrate requirements commonly found in web applications, such as restricting content from non-authenticated users or users who otherwise lack appropriate privileges.

The example application is a course management software that can be used by service providers (for example, universities or companies providing training) to publish course information and maintain course registrations and evaluations. The application will have typical features, such as authentication and authorization, create, read, update, and delete operations, and the need to maintain both data confidentiality and integrity.



## 2.1 Functional requirements

The example application will need to fulfill the following functional requirements:

1. Users and access rights
  - 1.1. The application must support four access rights levels: *non-authenticated users*, *course attendees*, *course instructors*, and *administrators*.
  - 1.2. Anyone should be able to register as a user with *course attendee* access rights, providing a name and email address.
  - 1.3. A *course attendee* should have access to everything a non-authenticated user has
  - 1.4. A *course instructor* should have access to everything a *course attendee* has in addition to views and actions limited explicitly to this role.
  - 1.5. An *administrator* should have access to everything a *course instructor* has in addition to views and actions limited explicitly to this role.
2. Viewing courses
  - 2.1. The application should list all available published courses that have yet to end.
  - 2.2. Regardless of access rights, any user has to be able to review course basic information for any published course (even those that have ended).
  - 2.3. A user should be able to use keywords to search for courses.
  - 2.4. Different views and variations of those (e.g., different course information) should be accessible through direct URLs and thus be bookmarkable.
3. Registering for a course
  - 3.1. A *course attendee* should be able to register for any course within its registration period, but not beyond it, if and only if the maximal number of attendees has yet to be reached.
4. Course management
  - 4.1. A course should contain the following information: name, description, course schedule (start and end date), maximal number of attendees, registration validity period, and status (published/unpublished).
  - 4.2. A *course instructor* should be able to add a new course.
  - 4.3. A *course instructor* should be able to modify all course information except course registrations if the course has not been completed.

- 4.4. A *course instructor* should be able to publish or unpublish courses.
  - 4.5. A *course instructor* should be able to grade the attendees for a course.
- 5. User profile
  - 5.1. A *course attendee* should be able to see in their user profile for which courses they have signed up for in their user profile.
  - 5.2. A *course attendee* should be able to cancel their signup during the registration period.
  - 5.3. A *course attendee* should be able to see their course grade for completed courses.
- 6. User administration
  - 6.1. An *administrator* should be able to define the access level for any user.

## 2.2 Non-functional requirements

The example application will need to fulfill the following non-functional requirements:

- 1. Data integrity of the course evaluations needs to be ensured. The application needs to provide means for auditing who has made the course evaluations and provide technical means for guaranteeing that the data has not been tampered with.
- 2. A set of password policies needs to be applied, even if hard-coded.

## 3 APPLICATION DESIGN

### 3.1 High-level architecture

The example application follows a simple three-layer architecture. As the name indicates, the application code is separated into three different logical layers: a presentation layer, a business logic layer, and a data access layer.

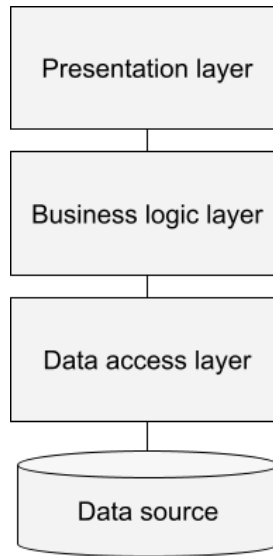


Figure 1. Illustration of the high-level architecture of the example application.

The presentation layer is responsible for the user interface and handling the interactions with the end user. The presentation layer solely manages and forwards the user interactions and is responsible for the user interface logic but does not know anything about how to access the underlying data layer or how to process user inputs.

The business logic layer is responsible for processing data and handling user interactions. The business logic layer decides *what* to do with the data and *how* it should be processed. While the business logic layer interacts with the data, it does not know anything about the underlying data sources or how they are accessed.

The data access layer acts as a glue between the application logic and the underlying data source. The data access layer determines how the data sources are accessed, potentially combining data from multiple data sources. This abstraction layer gives the flexibility to change the underlying data sources (for example, switch from one database provider to another or change the database level data model) without making any changes to the business logic or presentation layer.

### 3.2 Data model

The application is implemented with a single relational database. Below is a description of the database table structure used in the application.

### 3.2.1 Users and roles

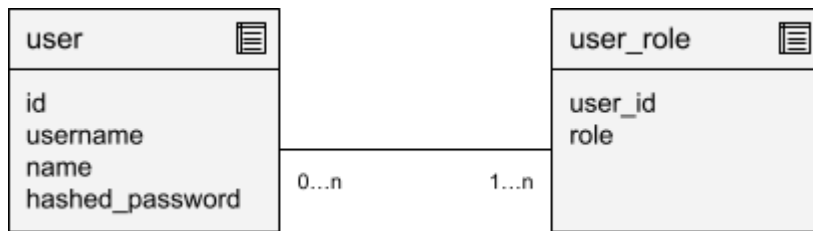


Figure 2. The data model for storing users and the user roles used for access control.

Users and roles are represented by two simple database tables with a many-to-many relationship. The `user` table contains details about the user, including a hashed version of their password with a salt value. The `user_role` table is a mapping table linking an individual user to the user roles it possesses.

### 3.2.2 Course information

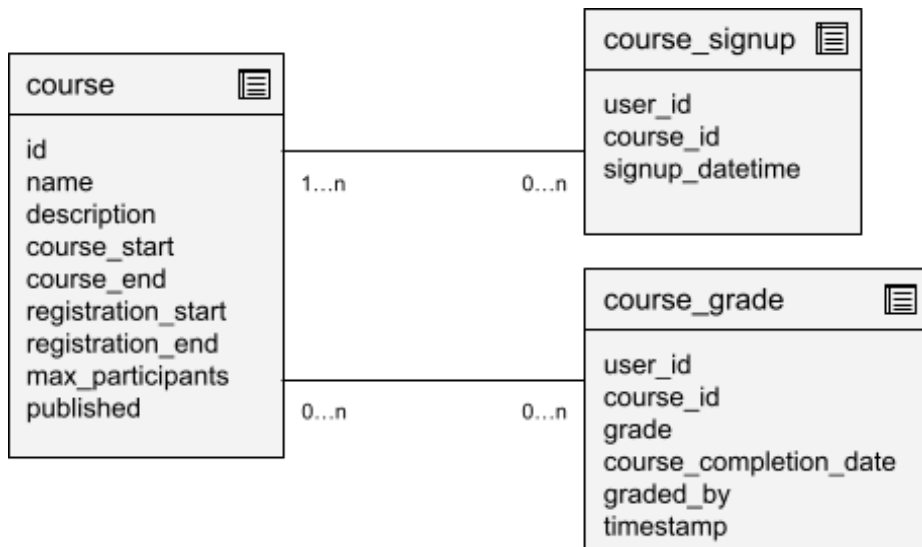


Figure 3. The data model for course-related information.

Managing the course information and registration data is implemented using three database tables. The `course` table hosts all the information related directly to the course, such as descriptions and validity periods. The `course_signup` table acts as a mapping table of which users have signed up for any particular course.

The `course_grade` table contains the information of a student's grade for a particular course. In principle, this information has been included in the `signup` table. However, as the concept of `signup` and `grade` differs on a logical level and in functionality, it makes sense to separate these into two different tables. A `signup` is a boolean value; either a user is signed up for a course or not. If the user is not signed up for a course, then the mapping table should not contain an entry for the user, while a `grade` is something more persistent data. A `grade` can be updated (for example, if the wrong grade was entered or the user has raised their grade), but once a `grade` has been given, it should never be deleted.

If the `grade` needs to be updated, it will not be managed as an update query; rather, a new table entry will be made with a new timestamp. When fetching a user's `grade`, we simply fetch the row with the latest timestamp for any given `user_id ↔ course_id` combination. This effectively means that the `course_grade` table's entries can be considered immutable, as they are never changed or deleted.

This logical difference allows us to implement an extra layer of security already on the database level. Instead of allowing the database user full access to the `course_grade` table, we can grant it only access to `SELECT` and `INSERT` queries, limiting the possibility of malicious users falsely modifying or deleting grades. However, this is not a sufficient measure to guarantee the integrity of the grades, as discussed later in the thesis.

## 4 IMPLEMENTATION

The example application is implemented using two popular Java-based frameworks. The user interface is implemented using the open-source Vaadin Flow framework, while the business logic and the data access layers are implemented using Spring Boot.

### 4.1 Vaadin Flow

Vaadin Flow is a full-stack web application framework that allows developers to build web applications purely in Java without writing any HTML or JavaScript code (Vaadin, n.d.-a). Unlike many other web frameworks, Vaadin Flow is a server-side framework, meaning the user interface logic and application state are handled and maintained on the server (Vaadin, n.d.-b).

Vaadin's architecture is discussed more in-depth in the chapter describing broken access control.

## 4.2 Spring Boot

The Spring Framework is a Java-based tool that provides a comprehensive programming and configuration model for modern Java-based enterprise applications. Spring provides infrastructural support on the application level, such as dependency injection, authentication and authorization, simple APIs for data access over JDBC or JPA, and many more core functionalities needed in any modern web application. Spring provides these functionalities without unnecessary ties to deployment environments, allowing developers to focus on application-level business logic (Spring, n.d.-c).

Spring Boot is a version of Spring Framework that takes an opinionated stance on the framework's configuration and third-party libraries, allowing the developers to create a standalone application with minimal upfront configuration (Spring, n.d.-d).

## 4.3 Presentation layer and the user interface

The application is built as a single-page application using Vaadin Flow as the frontend technology. The application consists of five views providing the main functionality of the application:

- The main view, accessible by all users, lists all available courses.
- The course detail view is used to view course details and sign up for a course.
- Course management view where teachers can add, modify, and delete courses and manage attendees and their grades.
- The user profile view is where an individual user can manage their account details and see their course signups and grades.
- The administration view is where the application administrator can, among others, manage the users of the application.

In addition to these main functionality views, there are a small number of helper views:

- Login view
- Registration view
- Search results view

The user interface is implemented following the principles outlined in Vaadin Flow's documentation. The application only uses out-of-the-box user interface components, and no custom client-side code was written. Application and user interface logic was implemented on the server side using Java.

## 4.4 Business logic layer

The business logic layer is lightweight and consists of only two classes: `CourseService` and `UserService`. These two classes contain the simple business logic needed for the application and interact as an intermediary layer between the user interface and the data access layer.

## 4.5 Data access layer

The application connects to a relational database, using MySQL as the relational database management system. The application uses a data access layer as an abstraction layer between the application logic and the part accessing the database and executing the queries.

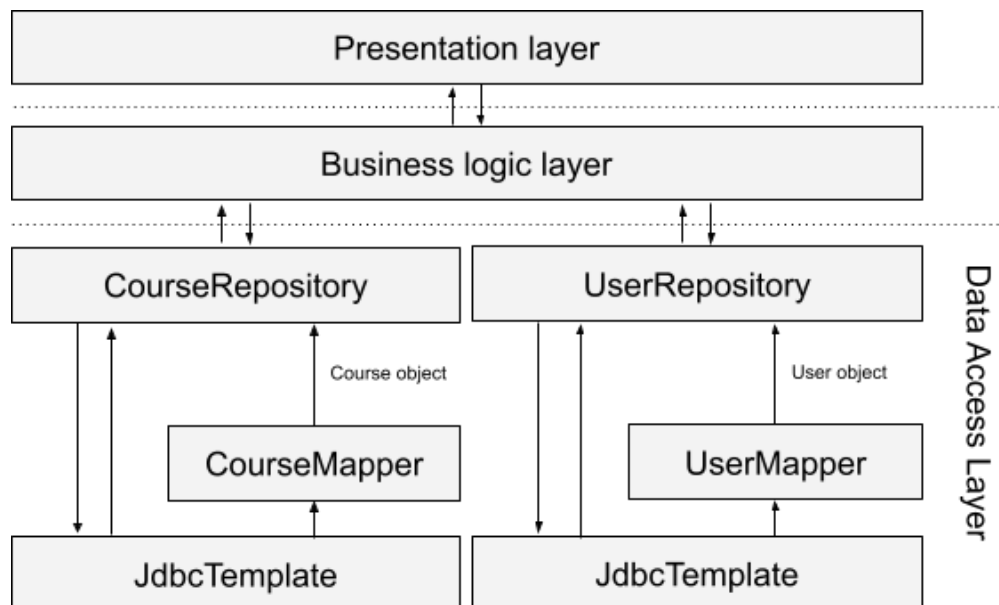


Figure 4. Illustration of the data access layer.

The data access layer consists of three types of classes: repositories, mappers, and models. Repositories act as a facade towards the business logic layer. It publishes APIs through which create, read, update, and delete operations are performed. The business logic layer never interacts directly with the underlying mechanisms, such as mappers or JDBC.

`JdbcTemplate` is Spring's core to the JDBC package, providing simplified usage of JDBC (Spring, n.d.-b). The class executes SQL queries and maps the responses into resultsets, which then can be mapped into Java objects. In the example application, for example, the `CourseMapper` is a

`RowMapper`, which translates a query result into `Course` objects that the business logic and presentation layers can directly use.

When the business logic layer makes a request to the repository, for example, asking for all available courses, it makes the requests through the `CourseRepository` and receives as a result a set of `Course` objects. The underlying `JdbcTemplate` and Mappers are entirely invisible to the business logic layer.

## 5 VULNERABILITY REVIEW

This chapter will review the vulnerability categories listed in the OWASP Top Ten list. Each category is presented with a high-level description, after which the category is analyzed against vulnerabilities on the application level versus vulnerabilities in the server configuration, certificates, or the development process. The vulnerabilities that are affected (or caused) by the implementation (including choice of frameworks) are discussed in more detail. The cause of a vulnerability is explained along with the recommended mitigation strategy; then, the example application is tested against the vulnerability. Finally, an analysis is made as to whether the vulnerability was mitigated by the frameworks used or by the developer's implementation and whether the framework could have done the mitigation.

### 5.1 A01:2021-Broken Access Control

Broken access control refers to a set of vulnerabilities that give unauthorized access to information or functionality that was not intended for the user (OWASP, 2021b). Let us consider a trivial example where an application provides partial access to news articles, but reading the full article requires a paid subscription. The application wants the user to be able to read the beginning of the article to get them interested but hides most of the content for non-subscribers. The feature might be implemented so that the beginning of the content is available for all users, and the remainder of the article is accessible through a "Subscribe to read the full article" button. What technical means does a developer have to limit the content to subscribed users? Is the full article added to the DOM tree and just visually limited? Is the full article accessible through another URL, and is the access limited to it in any way? Or maybe the application uses an AJAX call to a REST API that serves the browser with the article's content, but are there any checks in the REST API's implementation to make sure that the user is authorized to access the content?



To properly secure an application against broken access control, appropriate authorization mechanisms must be implemented in all relevant places, including the user interface and any available APIs. It includes limiting access to functionality, parts of the user interface, to resources (such as files), and APIs.

The OWASP Testing Guide instructs to test the access control both horizontally and vertically (OWASP, n.d.-c). Horizontal access control testing means we test if a user can gain access to functions or resources that should be accessible to a user who holds a different identity but has the roles or level of privileges. Vertical testing, on the other hand, tests if we can gain access to functions or resources that should only be accessible to a user who holds a higher role. Another approach that needs to be tested is privilege escalation. In other words, are there ways we can interfere with the application's behavior in such a way that we can change the role and access rights of a user to gain a higher role?

The testing starts by identifying the potential places where these kinds of privilege escalations or circumventions could happen. In the example application, it could be accessing views one does not have the rights to access (user and course management views) or performing actions that are not allowed to the given user (changing a user's role, modifying course data, course evaluations, or course registration for unauthenticated users).

Vaadin provides a built-in, view-based access control mechanism to manage the authorization of users' access to a view (Vaadin, 2023). To define the access rights for a view, a developer can simply annotate a view `@AnonymousAllowed`, `@PermitAll`, `@RolesAllowed`, or `@DenyAll` annotations. When a user wants to access a particular view in the application, an HTTP request is sent to the server, triggering a navigation event. Vaadin's navigation mechanism will automatically check the view class for access control annotations and based on the defined access controls and the current user's authentication status, either grant or deny access to the view. For example, the user administration view definition looks as follows:

```
@PageTitle("Users")
@Route(value = "users/:courseUserID?/:action?(edit)", layout
= MainLayout.class)
@RolesAllowed("ADMIN")
public class UsersView extends Div implements
BeforeEnterObserver {
```

Whether the access is granted or not, Vaadin will give the user access to the view or redirect the user to the login page. While Vaadin does provide the developer with an easy-to-use mechanism for controlling access to individual views, it does not, however, provide a means for performing access control on individual actions, such as adding a new course.

Vaadin's server-side architecture does, however, make it more challenging to attack individual actions. With a more traditional choice of frameworks, a developer might develop the user interface with one technology (such as React<sup>1</sup>), which communicates with a backend containing the business logic over REST APIs. With this architecture, the developer is responsible for implementing the server-client communication. That communication sequence flow might look something like this:

1. The end user fills in a form with course details and clicks on the “Add new course” button.
2. The presentation layer code (running as JavaScript in the browser) aggregates the information and makes an HTTP request to the correct API on the server. That API might live in a URL such as <https://example.com/api/v1/addCourse>.
3. The server processes the request and response with an appropriate HTTP response message.

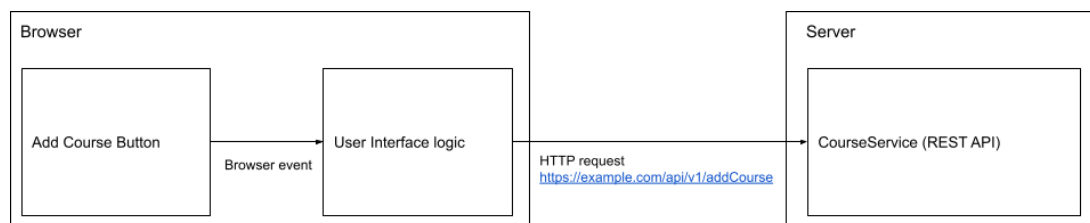


Figure 5. Illustration of how the “add course” action could be implemented in a more thick-client architecture.

With this communication architecture, the developer would need to secure the REST API as well to make sure that a user lacking the appropriate authorization cannot trigger the action that creates a new course. Even though the application's user interface logic performs the HTTP request, nothing is stopping a malicious user from circumventing the client-side logic and making the HTTP request to the REST API with arbitrary values.

Vaadin's component-based architecture differs from this by only exposing an RPC (Remote Procedure Call) interface to an individual component's actions, such as *the button was clicked*. The button click event is

---

<sup>1</sup> <https://reactjs.org/>

entirely business logic agnostic; only the information about a particular button in the user interface being clicked is sent to the server. The user interface logic resides on the server side, which then processes the button click event, which in turn calls the `addCourse` method in the business logic. Note that the `addCourse` method is never exposed to the Internet and thus would never be accessible directly by a (malicious) end user. It does not mean that when developing an application with Vaadin, the developer would not need to implement access control mechanisms for actions; it only means the architecture makes it more difficult for malicious users to exploit vulnerabilities in broken access control.

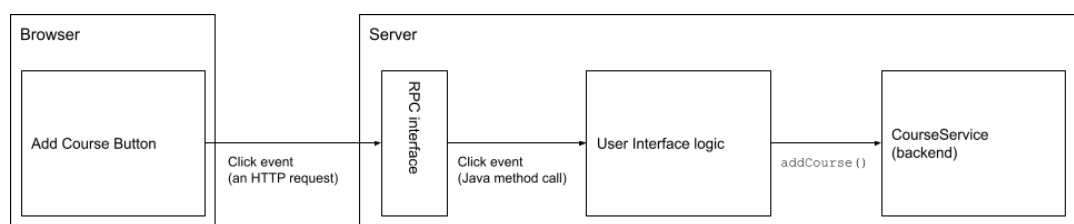


Figure 6. Vaadin’s architecture relies on remote procedure calls, which delegate only information on user interface events, such as button clicks, while the application logic resides completely on the server side.

Vaadin maintains its application state on the server side, meaning the server is, at any given moment, aware of what is visible on the user’s screen and in which state the different user interface elements are. In practice, this means that if we disable a button, the server will know that the button is in the *disabled* state. If the button is in the disabled state, then the server also knows that it should not be possible to click on the button. Thus, if an attacker tries to send a false request that emulates a click event, the server would know not to expect such an event and refuse to process the event, blocking any attempts to circumvent the application state.

While Vaadin’s inbuilt components use a user interface and business logic agnostic implementation, that is not necessarily true for components built by third parties or the development team. With Vaadin, it is possible to create your own components with their own client-side implementation, allowing the developers themselves to decide what kind of functionality is exposed directly to the internet as RPC calls. From a purely technical perspective, it is possible to create a more client-heavy view which would implement some of the user interface logic on the client side and expose the `addCourse` method through the RPC interface. For this reason, it is good to abide by the *defense in depth* principle. The defense in depth principle is based on layering security controls,

in other words, implementing security controls in multiple places and not relying just on one control (Conklin & Shoemaker, 2022, p183). The principle is not meant to be applied only at the source code level but also for, for example, the network and server infrastructure. However, we could implement the defense in depth principle in the example application by applying method-level authorization on the service layer. It has the added benefit of having the security controls already in place if we later want to expose those same services for another application, such as a native mobile client of the same application.

Spring Framework security module (Spring Security) has functionality for implementing method-level authorization (Spring, n.d.). As Vaadin's authentication implementation is based on Spring Security, the method-level authorization will work mostly out-of-the-box with a Vaadin application. The authorization mechanism works simply by adding an annotation to the methods we want to secure, defining which roles can access the method. When the method is being called, Spring Security will verify whether or not the given user has the appropriate roles to execute the method. If the user lacks the needed rights, then an `AccessDeniedException` is thrown.

In the example application, the appropriate place for adding method-level security would be in the service layer. The below example illustrates how the `CourseService` class secures the updating of course information to only valid user roles.

```
@Service
public class CourseService {
    ....
    @PreAuthorize("hasAnyRole('INSTRUCTOR', 'ADMIN')")
    public Course update(Course entity) {
        return repository.save(entity);
    }
    ....
}
```

Listing 1. Example of how method level protection is applied in the business logic layer, restricting user access based on roles to the function updating course data.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that *helpers are provided by the frameworks* for managing access control.

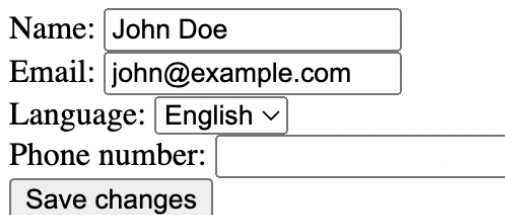
In addition to trying to circumvent access control directly, exploiting other vulnerabilities can achieve the same results. The OWASP Testing Guide instructs to test the application against Insecure Direct Object References. The

OWASP Top Ten category description also includes Cross-Site Request Forgery attacks in this category, which are attacks against improper session management. These two vulnerabilities are discussed in detail in the following sections.

### 5.1.1 Insecure Direct Object References

An insecure direct object reference vulnerability occurs when an application exposes details about internal implementation objects, such as files, database records, or keys, as URL parameters or as hidden fields in forms. Exposing the internal values used for referencing an object can be dangerous, as it exposes the application to a multitude of different kinds of attacks and potentially gives an attacker access to read, update, or delete content that they otherwise would not have access to.

Consider a typical web application requiring its users to register to the site. The application will most likely have some sort of profile page the user can use to manage their details. The form for editing one's details might look as shown in Figure 7.



Name:

Email:

Language:

Phone number:

Figure 7. Example of a form used for modifying a user's details.

Traditionally, these forms have been built using HTML's form elements, with the visible parts as input fields and metadata added as hidden fields. The example form's HTML is shown below.

```
<form action="/updateProfile" method="POST">
<!-- Meta data -->
<input type="hidden" name="user_id" value="4183" />
<!-- Input fields -->
Name: <input type="text" name="name" value="John Doe" /><br>
Email: <input type="text" name="email"
value="john@example.com" /><br>
...
<input type="submit" value="Save changes">
```

```
</form>
```

Listing 2. Example of how a traditional HTML form for updating a user profile might be implemented.

The example form exposes internal object details in the metadata, namely, the user ID field. The user ID field is added so the application knows which user record needs to be updated. From the field's value, we can guess that the value is most likely the primary key of the database record containing the user details. If the application lacks authorization checks to ensure the current user whose profile is being modified is the same, then the application would make itself vulnerable to a malicious user modifying a database record to which they should not have access. Even though the user ID field is not visible on the rendered page, it is still trivial for a malicious user to modify the hidden field's value to any other value.

For example, this kind of vulnerability can be used to take over an account. Let us assume that a malicious user could change the email address of any given user by just changing the `user_id` field's value. If the malicious user can figure out the administrator's user ID (it might be simply the first database record, having an ID value of '1'), then they could potentially use this vulnerability for an account takeover. Changing the email address means that any emails sent by the system will be redirected to an address of the attacker's choice - including any password reset emails. Password resets are often implemented by submitting a reset link to the email address provided in the user profile. If multifactor authentication is not enabled, then this attack path might be successful and compromise the whole system.

The vulnerability can also be leveraged against other objects than database records. Consider an application with three roles: normal users, power users, and administrators. Maybe a power user can manage other user accounts, including elevating their roles up to also being power users. The input field for selecting the appropriate user role might be a simple select field, as shown below. What would happen if a malicious user used an HTTP proxy or the browser's developer tools to change the value of one of the options to "administrator"?

```
<select name="role">
  <option value="user">Normal user</option>
  <option value="power_user">Power user</option>
</select>
```

Listing 3. A selection drop-down menu uses the internal role names as the option values, thus exposing the application's internal implementation details.

It is worth noting that this vulnerability is not limited to just hidden form fields. However, it can be any reference to an object that exists on the client side, for example, cookies, URL parameters, JSON objects in the local storage, form field values, or any other form of data that is exposed to the browser.

#### **5.1.1.1 Mitigation strategies**

There are two main mitigation strategies against indirect object reference vulnerabilities (CWE, n.d.). The first method to mitigate the vulnerability is to implement record-level access control, meaning that the application will verify that the user requesting access to an item has permission to access the record. In the profile update example above, the application should verify that the `user_id` value in the hidden form field matches the user ID of the logged-in user. If there is a mismatch between the IDs, then the action should be denied.

The second mitigation strategy is not to expose the IDs to the browser in the first place. It can be implemented by encrypting the IDs or by replacing and mapping the keys to internal values. For example, instead of using numbers, the application could use a randomly generated string or some other representation of the value that does not expose the underlying key.

#### **5.1.1.2 Framework level mitigation**

Vaadin provides protection against indirect object reference vulnerabilities out-of-the-box without the developer needing to even be aware of it - to some extent. In a Vaadin application, the client side is relatively thin, meaning it does not have any application logic and contains only a limited amount of application data. In a Vaadin application, the server is in complete control of the application state and what is shown on the screen and, thus, in full control of what data needs to be exposed to the browser. This architecture allows Vaadin to limit the exposed data to contain only relevant data for rendering the view in the desired way, meaning only the texts that need to be shown. It allows Vaadin to hide all underlying objects and never expose any identifiers, such as primary keys, to the browser.

The following example will illustrate how Vaadin hides internal object details from the browser and only exposes relevant data. The example implements a similar role selection as described in section 5.1.1. For this example, we have implemented a Java class called `Role` with two properties, `name` and `UUID`. The `UUID` is a randomly generated string that uniquely identifies an object. This property represents an internal value that should not be directly exposed to the client side.

```
private class Role {
    private String uuid;
    private String name;

    public Role(String uuid, String name) {
        this.setUuid(uuid);
        this.setName(name);
    }
    // Getters and setters are omitted for brevity
}
```

Listing 4. Example of a class that uses a UUID as the unique identifier instead of an integer.

Next, an ArrayList is populated with three Roles, each with its unique identifier. The ArrayList is then linked with Vaadin's Select component (Vaadin's implementation of a dropdown menu).

```
List<Role> roles = new ArrayList<>();
roles.add(new Role(UUID.randomUUID().toString(),
    "Administrator"));
roles.add(new Role(UUID.randomUUID().toString(), "Power
    User"));
roles.add(new Role(UUID.randomUUID().toString(), "User"));

Select<Role> roleSelect = new Select<>();
roleSelect.setItemLabelGenerator(Role::getName);
roleSelect.setItems(roles);
```

Listing 5. Three roles are added to a dropdown selection menu in a Vaadin application. Note how the role UUIDs are passed to the select component.

When rendering the view, the response to the HTTP request contains a JSON object describing what should be rendered to the screen. By examining this JSON object, we can see that the role names are sent to the browser (as they are used as the labels in the dropdown) but not the UUIDs. In listing 6 we can see that for each item, we have a text value for the label, but the item's value property is simply an integer between one and three, which in this case is also the order value of the items in the dropdown selection.

```
[
```



```

{
  ...
  "changes": [
    ...
    {
      "node": 48,
      "type": "put",
      "key": "label",
      "feat": 3,
      "value": "User"
    },
    {
      "node": 48,
      "type": "put",
      "key": "value",
      "feat": 3,
      "value": "3"
    },
    ...
    {
      "node": 50,
      "type": "put",
      "key": "label",
      "feat": 3,
      "value": "Power User"
    },
    {
      "node": 50,
      "type": "put",
      "key": "value",
      "feat": 3,
      "value": "2"
    },
    ...
    {
      "node": 52,
      "type": "put",
      "key": "label",
      "feat": 3,
      "value": "Administrator"
    },
  ],

```

```

    {
        "node": 52,
        "type": "put",
        "key": "value",
        "feat": 3,
        "value": "1"
    },
    ...
}
]

```

Listing 6. Part of the JSON in the HTTP response instructs the presentation layer to render a dropdown menu. We can see that the labels for the options are in the JSON, but the UUID is never exposed to the client side.

Even though Vaadin does provide an out-of-the-box solution for protecting the application against indirect object references, it is still possible for a developer to introduce such a vulnerability. The example application's requirement 2.4 states that a user should be able to bookmark the individual views and their content variations. In practice, this means that, for example, for the course details view, we want to be able to access any individual course's details directly through a unique URL. The URL could look like this:

<https://example.com/course/33183/>  


  
domain
view
parameter

Figure 8. The URL structure for accessing a specific course's details.

In this example, the URL would consist of three parts: the domain, view name, and view parameters. The third part, the parameter, is in the example application used for defining which course information should be shown in the view. The example shows it as an integer value, but in practice, it could be any value that can be used in a URL. When encountering an integer value in the URL, an educated guess would be that it is mostly the primary key for the database record containing the view's details. It would expose the application to be vulnerable to an indirect object reference exploitation. If the application's implementation lacks record-level access control, then an attacker could enumerate all courses, even those that should be hidden from the user. Viewing hidden course details is not probably the end of the world, but what if

a similar vulnerability would exist in a healthcare application showing patient data?

A framework, such as Vaadin, cannot know what a developer's intent is with the view parameters. They may or may not be used for identifying objects. Because the view parameters can be used in any arbitrary way, a framework cannot impose a mechanism to obfuscate the values, as that might break the intent of the parameters themselves. Hence, securing this part of the application's functionality will reside with the developer, not the framework.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that indirect object reference vulnerabilities are *mitigated out-of-the-box by the framework*, to the extent feasible, considering limitations to knowing the developers' intentions.

### 5.1.2 Cross-Site Request Forgery (CSRF)

Cross-site request forgery, also known as CSRF or XSRF, is a vulnerability in web applications where a user is tricked into performing an action in the vulnerable application without the consent or knowledge of the user. This vulnerability becomes possible if the server treats an HTTP request as an authentic request regardless of its origin (CWE, n.d.-b).

What makes this vulnerability devious is that it is executed through the victim's browser using the victim's HTTP session. From the server's perspective, the HTTP request is made by a valid, authenticated user and thus any security controls implemented to restrict access to functions or data will not be sufficient to protect the user and the application against this vulnerability.

The vulnerability can best be explained through an example. Consider an online banking application where users can pay bills and transfer money to other accounts. The application is built as a single-page web application that uses its backend functionality through REST API calls. When the user Alice wants to make a money transfer, a GET request is made to the URL `http://bank.site/api/transfer` with two parameters, `account` and `amount`, where the first parameter specifies the receiver's account number and the latter specifies the sum to be transferred. When transferring money to the account number 1234, the request could look as follows:

```
http://bank.site/api/transfer?account=1234&sum=1000
```

This request contains two types of information: the GET parameters dictating how much money should be sent and the account number dictating to which account the money will be sent. The HTTP header will contain the user's normal session token, authenticating the user as Alice.

A malicious user, let us call them Eve, could now use this information to create a cross-site request forgery attack and trick Alice into transferring money to Eve's account. Eve can construct a malicious URL and send it as a link to Alice.

```
<a href="http://bank.site/api/transfer?account=5678
&sum=1000">View my pictures</a>
```

Alice will see a link with the text “View my pictures,” if she is not paying attention, she might not notice that the link leads to somewhere other than what the text might indicate. When Alice clicks on the link, a request is sent to the server using Alice’s session requesting money to be transferred to Eve’s account. As the request came from Alice’s browser, using Alice’s session, the server will not be able to distinguish this from any other valid request. This particular attack would require Alice to click on the link to trigger the request. Eve could increase her chances of having the malicious URL requested by embedding the URL in an image tag and using social engineering tactics to trick Alice into visiting a website containing the image tag.

```

```

Visiting a page containing the above image tag would trigger a similar HTTP request as Alice clicking on a link directly, with the exception that the only indication that something is wrong is that the image would not load. Alice would only see a broken image while the banking application receives a valid-looking request for a money transfer. The broken image can be circumvented by visually hiding the image, for example, by making the image’s size 1px by 1px. If Alice is logged into her banking account when visiting the malicious site, money will get transferred from her account to Eve’s account without Alice noticing it (OWASP, n.d.-d). Although the example scenario is unlikely, it demonstrates the principle behind the attack.

The application level trust problem, which is the principle behind cross-site request forgery attacks, was first described by Norm Hardy (1988, pp. 36-38) in what he called “confused deputy”.

In January 2008, Symantec reported a cross-site request forgery attack that took place in Mexico, an attack with devastating consequences (Symantec, 2008). The attacker embedded malicious code inside an e-mail and sent it to unsuspecting users. The e-mail was disguised as a notification to the victim, telling him they had received an e-card at a popular website. The e-mail also contained an <img> tag, triggering an HTTP GET request to the user’s router control panel. The attack targeted a specific vulnerable route model popular in Mexico. The router’s security vulnerability allowed the attacker to redefine DNS settings with a cross-site request forgery attack

without requiring the user to log in to the router's control panel. Back to the malicious email containing the attack payload, anyone who loaded the HTML in the e-mail and was the owner of this specific router model became a victim of the attack.

The cross-site request forgery attack changed the victim's router DNS settings so that any requests to a popular Mexico-based banking site would be automatically redirected to a domain controlled by the attacker. The domain controlled by the attacker had a rogue version of the banking site, and any user who used this site had their credentials stolen.

### 5.1.2.1 Countermeasures

A study by Likaj et al. (2021) outlined the various methods used in web applications to counter cross-site request forgery vulnerabilities. They identified 16 distinct defense mechanisms against CSRF, categorized into four groups of vulnerable behaviors that, when removed, the CSRF attack is no longer successful. These four groups introduced by Likaj et al. (2021) are described below with examples of concrete mitigation strategies.

**Origin checks.** As the name of the vulnerability indicates, the source of a CSRF vulnerability is when the server accepts a request originating from a third-party source as a valid, user-intended request. It may be a JavaScript XHR or submitting a form on a website controlled by the attacker. An effective countermeasure, when applicable, is verifying that the request's Origin header matches the application's domain address (Barth et al., 2008, pp. 82-83). Tricking a user into clicking on a link in an e-mail or embedding the malicious URL to an image tag and posting it on a forum would mean that the attack originates from another site than the site where the target application lies. It means the HTTP header containing the origin site would be something other than the target application's URL. As the origin header is defined by the browser and not by the requesting application, we can ensure the request does not originate from a third-party site. However, it is worth noting that the Origin header is not included in all requests, for example, in GET requests (Mozilla, n.d.). Thus, all state-changing requests should use, for example, POST requests.

Another approach to ensuring the requests are coming from the same domain as where the application lies is using custom headers (OWASP, n.d.-f). JavaScript allows developers to include custom headers in XMLHttpRequest (XHR) calls. Custom headers can, however, only be used within the same domain and will not be included in the HTTP request if the call is cross-domain. The server can thus check the presence of the header - if present, the request must have come from within the same domain. If absent,

the request may have originated from another source. The downside of this approach is that it is only applicable for XmlHttpRequests.

**Request Unguessability.** An application is vulnerable to CSRF if the attacker can consistently replicate a request to seem valid. An effective countermeasure is to modify the content of an HTTP request so that an attacker cannot know all the needed content of a request, which would be considered valid for processing.

Stateful applications can use a shared secret (token), which is included in all HTTP requests and validated by the server before processing the request. A cryptographically secure token is generated on the server side and stored in the server-side session. The same token is provided to the client side for the browser to include in every HTTP request as either a URL parameter (for GET requests) or as a parameter in the request body (Schreiber, 2004).

Consider the use case described earlier, where a user wants to transfer money from their account to another account. When the user opens the form containing the information for money transfer, a token is created, which is unique to this specific action for the active user session. This token is then added as a parameter to the HTTP request by embedding the token as a hidden field in the form. When the application receives the request, it validates that the token is present and is actually linked to the active user session. If a valid token is not found, the application should not perform the requested action but instead log the incident in the security logs as a possible case of a cross-site request forgery attack.

```
<form action="/transfer">
<input type="text" name="account" />
<input type="text" name="sum" />
<input type="hidden" name="csrf_token"
    value="so3rrZtS0khWkwh9h6Fbh2tUqFsoXYd4BMzfo3rXAL8LcyLj
    PQcV1oqeizWb4R9" />
...
```

Listing 7. Example of a HTML form containing a unique CSRF token.

Even though using secret tokens provides an application with a high level of protection against cross-site request forgery attacks, it does not secure an application completely against these attacks. The application is still vulnerable if the token is leaked to an attacker. There are multiple ways this could happen. For example, it could be exposed via GET requests where the token could leak to HTTP log files, browser history, or the referrer header if the application links to a third-party site (Likaj et al., 2021, p. 375). Hence, it is recommended that all sensitive actions are performed as POST requests and GET requests are

only used for data retrieval; this way, security tokens do not need to be embedded in the GET requests (OWASP, n.d.-f).

A similar approach can be applied to applications with a stateless backend. This pattern is called “double submit,” which relies on submitting a token in two different ways so that an attacker cannot forge it. Instead of storing the token in the server-side session, the token can be stored as a variable in a cookie. The application thus needs to send the token both as a cookie value and as a parameter in the request. The server then validates that the parameter in the body matches the parameter in the cookie. This mechanism is effective, as cookies’ same origin policy disallows third-party sites from reading cookie values.

**Same-Origin Policy for Cookies.** An application can add a browser cookie, which is required to be passed back to the server for any state-changing action to be performed. If the cookie is missing, the server should reject the action. Setting the cookie’s `SameSite` attribute to `strict` will ensure that the browser will pass the cookie to the server only if the request originates within the same domain.

**User Intention.** Cross-site request forgery attacks can also be hindered by verifying the user’s intent. After performing an action, such as transferring money from an account, the user can be requested to perform a simple task to verify that the action was intentional. Such a task could be solving a CAPTCHA or entering a one-time token delivered by, for example, an SMS or an email.

It is worth noting that any two-step action is not sufficient to protect against cross-site request forgeries. For example, a simple confirmation popup asking, “are you sure you want to transfer the money?” is insufficient if the HTTP request sequence is predictable (Schreiber, 2004). Let us consider the above example of money transfer. The initial request might look like this:

(1) `http://bank.site/api/transfer?account=5678&sum=1000`

After this, the user is presented with a summary page of the transfer with a button for confirming the transaction. This transaction triggers another HTTP request, such as the one below.

(2) `http://bank.site/api/transfer?confirm=1`

A successful transfer would now require the user to confirm their intention. However, since the two HTTP request contents are predictable, an attacker can craft a simple script that first makes an HTTP call to (1), after which a slight delay is performed, after which an HTTP request is triggered to (2) finalizing the transaction, without the user’s knowledge or true consent.

### 5.1.2.2 Framework level mitigation

We can examine the example application's HTTP requests to determine if any CSRF protections are in place. A simple action in the application triggered the following request.

```
POST /?v-r=uidl&v-uiId=0 HTTP/1.1
Host: localhost:9090
Content-Length: 210
// ..some headers excluded for clarity
Origin: http://localhost:9090
Referer: http://localhost:9090/
Cookie: JSESSIONID=9B7BE1749278E15205B884481C6F4C44
Connection: close

{
  "csrfToken": "710307d4-fa87-4fc3-967a-9e5dd01f438f",
  "rpc": [
    { "type": "publishedEventHandler",
      "node": 5,
      "templateEventMethodName": "confirmUpdate",
      "templateEventMethodArgs": [1],
      "Promise": 1 }
  ],
  "syncId": 3, "clientId": 3
}
```

Listing 8. An HTTP request made by a Vaadin application. We can see the presence of a CSRF token in the request body.

We can immediately see that the request body has JSON containing a `csrfToken` attribute. This token is added by the Vaadin framework automatically out-of-the-box without any configuration required by the developers. The same token is repeated in subsequent HTTP requests. Vaadin uses only one endpoint through which all remote procedure calls (RPCs) go through without providing any additional endpoints, limiting the potential attack surface. Thus, we can conclude that Vaadin applications are sufficiently secured against CSRF attacks by making the requests unguessable using a CSRF token.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that cross-site request forgery vulnerabilities are *mitigated out-of-the-box by the framework*.



## 5.2 A02:2021-Cryptographic Failures

The next category in OWASP's top ten list is "Cryptographic Failures". In previous versions of the top ten list, the category was called "Sensitive Data Exposure", which describes the symptoms often caused by cryptographic failures. This category deals primarily with vulnerabilities related to data and how sensitive data might inadvertently be exposed in the application (OWASP, 2021c).

Sensitive data exposure is something that needs to be subjectively considered for each application - one type of data might be sensitive in one application while in another application, it would not be considered sensitive. While a lot of the data sensitivity categorization is subjective, there are, however, a number of types of data that always need to be protected in a secure manner, such as passwords, health information, or credit card numbers.

Once the sensitive data in the application has been identified, we need to consider how the data is protected both when at rest (for example, how it is stored in the database or disk) and when in transit (for example, how the data is protected between the browser and the server, between the load balancer and the backend). Protecting the data is done using cryptographic measures, for example, by encrypting or hashing the data. When using cryptographic methods, we need to ensure that the used algorithms are not deprecated as old and weak.

The example application does not have a lot of data that would be considered sensitive, with the exception of users' passwords. Nor does the application integrate into any third-party services, which would need to be secured. Let us review how Vaadin Flow and Spring Framework handle the passwords both at transit and at rest.

The example application is built to be deployed and run on one application server, meaning, the web server serving the browser and the backend all reside on the same application server. Hence, there is no data in transit on the server side, only between the browser and the web server and potentially, depending on the configuration, between the application server and the database server. Encryption of the data between the browser and server is done using SSL, which is configured in the application server and not in the application code. Hence, we can conclude that considering data at transit is out-of-scope in this thesis.

Let us consider how the frameworks help us store passwords in a secure manner. Passwords should never be stored as plaintext or encoded (CWE, 2006) as a vulnerability, such as an SQL injection, might expose all of the users' passwords to the attacker. A password should be stored so that

even if the raw data of the password (typically, a cryptographic hash of the plaintext password) is compromised, it would be impractical for an attacker to guess or otherwise discover the correct password. There are a number of best practices that dictate how passwords (or other memorized secrets, such as PIN codes) should be stored; for example, one is released by the US government's National Institute of Standards and Technology (Grassi et al., 2017, pp. 13-15). When storing the password, the password shall be salted (CWE, 2009) and hashed using a suitable (and strong enough) one-way key derivation function (CWE, 2006b). The salt used should be such that it is not easily predictable, such as using the user's username as the salt value (CWE, 2009b).

### 5.2.1 Framework level mitigation

Vaadin Flow is agnostic to how and where any application data is stored, as it is purely a user interface framework. Vaadin Flow does integrate with Spring Security, allowing developers to leverage Spring's authentication features. Vaadin promotes using their Vaadin Start service to create an application stub upon which a developer can continue building their Vaadin application (Vaadin, 2023b). The application stub provides a simple login feature using an in-memory database integrated with Spring Security. In the stub application, the passwords are stored as hashes using bcrypt, which is among the recommended algorithms for this purpose (OWASP, 2021d, p 26). It is worth noting that nothing forces the developers to use this practice but rather leaves password management as the developer's concern.

On a more general level, storing sensitive data, not just passwords, needs to be protected accordingly. Upon inspection, Spring Framework does not seem to provide any helpers for the developer to handle storing sensitive data.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that the chosen frameworks *do not provide mitigation support* against cryptographic failures, even though there is some help for one use case, but a generic solution is lacking.

## 5.3 A03:2021-Injection

An injection attack is possible when a web application does not validate user input data and uses the unvalidated data in the application logic. It might allow a malicious user to *inject* their code into the application logic and perform activities the application was not designed to do. Such an activity could be, for

example, reading database content that is not meant to be exposed to the end user.

Most of today's websites and applications are, to some degree, interactive or dynamic. Even a simple company website might today be built using content management systems (CMS), have a contact form, or maybe ask the user to sign up for their newsletter. It means that even the simplest websites are rarely completely statically implemented using plain HTML files with no scripting or backend functionality and thus interact with the end user in some way. For a simple website built using a content management system, it means, at minimum, that the CMS needs to know which pages to serve to the end user. The navigation information is often passed to the CMS using URL parameters or dynamic URLs. This information is considered user-provided data; if the data is not validated on the server-side, it might make a web application vulnerable to injection attacks.

Injection attacks are based on a problem where user input data is poorly (or not at all) validated before being used in the application logic. Failure to properly validate the user's input data can lead to a situation where the attacker can manipulate the application's commands by entering unexpected data and thus changing the commands executed by the application. Consider an example where the user is asked for his birth year. We know that the input data the user provides should be a four-digit number; however, nothing guarantees that the user will enter a valid value. The question is how an application will behave with unexpected data. What if the year is something other than numbers? If the input data has not been validated, we cannot be sure how the application will behave. In a best-case scenario, the application will discard the value and continue functioning as expected. A worse scenario would be that the application goes into a state from which it cannot recover automatically; in other words, the application could break, a situation that would be bad for the user experience. The worst possible scenario would be that invalid data validation would open up unexpected doors for an attacker, allowing him to perform various attacks against the server, the application, and the users.

User input should always be validated before using any application logic, such as SQL queries. A common mistake made by inexperienced programmers is the failure to properly validate data from an HTML form. An HTML form can contain hidden fields which are not visible in the user's browser. Often, these fields are filled by the application logic and never require any interaction from the user. An inexperienced programmer may think that validating these fields is not necessary, since it was filled by the application logic and therefore we know what it contains. Hidden fields can be misleading because even if the user cannot see the field in his browser, it does not mean that it cannot be manipulated. Therefore, it is imperative always to validate all

data sent by the client (in this case, the browser) and not only the visible fields of a form and trust the hidden content to be valid.

In this chapter, we will discuss various injection attacks. Some of these injection attacks target the server, trying to extract data, elevate privileges, or even compromise the server itself. Some injection attacks, such as cross-site scripting attacks, target the end user, for example, to steal their credentials.

### **5.3.1 SQL injections**

Structured Query Language, or SQL, is a standardized computer language designed to retrieve, manipulate, and manage data in relational databases. The most common SQL commands have to do with data retrieval and manipulation, but SQL is not limited to that. With SQL, one can also manipulate the actual database, for instance, by creating and deleting database tables, users, or even entire databases.

SQL injections were first reported in 1998 when a security researcher noticed that by manipulating normal user inputs like “name”, he was able to extract sensitive data from a Microsoft SQL server (Horner & Hyslip, 2017, p. 99). A SQL injection vulnerability occurs when unvalidated user input data is passed directly into a query, modifying the query’s original purpose. The reason to perform an SQL injection varies a lot. However, the attack intent can be grouped into the following categories: identifying injectable parameters (in other words, probing which parameters are vulnerable to SQL injection attacks), performing database finger-printing, determining database schema, extracting data, adding or modifying data, performing denial of service, evading detection (for example, remove auditing logs), bypassing authentication, executing remote commands and performing privilege escalation (Halfond et al., 2006).

There have been numerous instances of data systems falling victim to SQL injections. Albert Gonzalez, an American hacker, was sentenced to 20 years in prison for his involvement in a credit card processor breach. Collaborating with two other hackers, Gonzalez employed SQL injection techniques to infiltrate 7-Eleven’s network in August 2007. This breach resulted in the unauthorized access and theft of an unspecified amount of credit card data. In the same year, the hacker group utilized SQL injection to compromise Hannaford Brothers, which led to the theft of 4.2 million debit and credit card numbers. (Zetter, 2010).

#### **5.3.1.1 Example**

SQL injections can best be described through an example. Consider the example application, which requires the user to log in before he gets access to certain parts of the application. Logging in is done by providing a username and a corresponding password. Our example application has a database table called 'user' containing the login credentials of all users. The table contains four fields: a user ID, the username, the user's name, and the password. Let us view an example of how the table would be populated with the three fields typically used for authenticating the user (excluding the name of the user).

id	username	hashed_password
1	George	#####
2	Susan	#####
3	William	#####

Table 1. Example data in the 'user' table. The hashed passwords are masked for simplicity.

When a user logs in, he provides a username and a password in a login form. The user-given password is hashed, and then the database is queried for the user with the following SQL query.

```
"SELECT id, username, hashed_password FROM user
WHERE username=' ' + username +
' ' AND hashed_password=' ' + password + ' '"
```

If the input value for the username is "George" and the value for the password is "foo", then the resulting query would be

```
"SELECT id, username, hashed_password FROM user
WHERE username='George' AND hashed_password = '<hashed value of
foo>'"
```

If both the username and the password match the ones in the database, the query will return the first row in the table. However, the query was constructed by concatenating the query's content and the user-provided input. Suppose the user's input data is not validated. In that case, an attacker can leverage a tautology-based attack where code is injected with a conditional statement that is always evaluated to be true (Halfond et al., 2006). Consider what would happen if the string "' OR 1=1 --" would be entered as the username. The resulting query would look like this:

```
"SELECT id, username, hashed_password FROM user
WHERE username=' ' OR 1=1 -- ' AND hashed_password = ''"
```

The username provided by the attacker begins with a single quote, which closes the username constraint in the query. The username field cannot be empty, which would thus yield zero results, but then the attacker added the conditional (`OR 1=1`), which is always evaluated as true. The double dash represents the beginning of a comment, and the query interpreter will ignore everything after the dashes. This query would thus return all users in the database.

This application might still not work correctly, as it might expect the query to return just one row, not multiple rows. The injection could be further modified by adding a clause that limits the query from returning more than one row. Depending on the database engine used, this can be achieved, for example, with MySQL, by adding "`LIMIT 1`" to the end of the query (MySQL, n.d.). With this simple injection, a malicious user could gain access to the system without knowing a single username or password.

A more harmful SQL injection attack could be done using a piggy-backed query-based attack. In this type of an attack, the attacker tries to inject additional queries to be executed together with the original query (Halfond et al., 2006). A malicious user could try to drop database tables to perform a denial-of-service attack. It can be achieved by entering the following username: `"'; DROP TABLE user; --"`. The resulting query would be

```
"SELECT id, username, hashed_password FROM user
WHERE username=''; DROP TABLE
user; -- ' AND hashed_password = ''"
```

In SQL, the semicolon represents a separator between two different queries. The resulting query would actually be two different and independent queries. The first query would try to fetch a user, while the second query would delete the user table altogether. To, among other, limit the impact of these kinds of potential vulnerabilities, it is recommended that the database user used for accessing the data is only granted access to specific databases and granted limited permission for performing actions, for example, allowing only `SELECT`, `UPDATE` and `DELETE` queries (OWASP, n.d.-j).

### 5.3.1.2 Countermeasures

SQL injection countermeasures can be broadly classified into three different categories: defensive coding, SQL injection vulnerability detection, and SQL injection attack runtime prevention. The SQL injection vulnerability detection methods rely on methods and tools mainly used in the testing and debugging phase, such as using static analysis, injection tools, or manual testing. The runtime prevention of SQL injection attacks is based on tools deployed together with the software that might, for example, try to recognize and stop harmful strings (Shar & Tan, 2013, p70-75). In this thesis, we will focus more on the defensive coding, as it is more relevant to what help frameworks can provide to the developers.

The means to defend against SQL injection vulnerabilities depends to some extent on the nature of the SQL query being executed. For most cases, such as the example used earlier in this section, the best approach would be to use prepared statements using parameterized queries (OWASP, n.d.-k).

```
public void login(String username, String hashedPassword) {
    String query = "SELECT id, username,
hashed_password FROM user WHERE username=? AND
hashed_password=?";
    PreparedStatement statement =
connection.prepareStatement( query );
    statement.setString( 1, username);
    statement.setString( 2, hashedPassword);
    ResultSet results = pstmt.executeQuery( );

    // The rest of the method is removed for
    // simplicity
}
```

Listing 9. An example of a SQL query being structured used parameters and executed as a prepared statement.

The above example illustrates the usage of prepared statements with parameters. The developer first defines the query and then binds the parameter values to the query. This way, the database can distinguish between what the executable code for the query is and which parts are variables. This approach makes it impossible for an attacker to escape the variable context to modify the actual query's structure.

A similar alternative for prepared statements would be to use stored procedures. Stored procedures are queries that are "stored" and thus known by the database server. When the application wants to execute a query, it makes a *call* to the stored procedure with the wanted parameters. In this

method, the user's input values only affect the variables in the query but cannot modify the structure of the query itself.

Stored procedures and prepared statements work well in cases where the structure of the SQL query is known and not constructed dynamically. OWASP (n.d-k) encourages avoiding dynamic queries and refactoring the code to only use known queries when possible. However, there are some valid use cases where dynamic queries might be necessary. One such example could be dynamic reporting - in order to allow the end user completely free hands in combining and fetching data for reports, it might be required to create the queries dynamically, meaning, for example, changing the `FROM` part of the query or adding `JOINS` based on the end user's input.

When dynamic queries are needed, a combination of escaping, data type validation (e.g., not allowing strings where integers are expected), and whitelisting (only accepting predefined values) should be used. If possible, one should also consider using an Object Relational Mapping (ORM) framework that builds the queries for you.

#### 5.3.1.3 Framework level mitigation

Spring Frameworks comes with modules providing the developer with a variety of ways to connect to an underlying data source. Spring Data JPA makes it easy to implement repositories using the Java Persistence API (JPA). JPA provides developers with an object-relational mapping facility, which allows the developers to interact with the database through objects and method calls. The underlying SQL queries are mostly hidden from the developer.

For those developers who want to avoid using JPA, Spring Data JDBC provides an alternative that provides similar functionalities with Object Mapping and repositories (Spring, n.d.-g). Developers can define their queries using annotations, which by nature are completely static and cannot be modified in runtime. Any variables needed in the query are thus forced to be parameterized.

The developer can interact with the `JdbcTemplate` class, which simplifies the use of JDBC to avoid common errors. It executes core JDBC workflow, while the application code is responsible for providing the SQL query and extracting the results. The following example illustrates how we can update the user details of a given user using the `JdbcTemplate`.

```
String updateQuery = "UPDATE users SET name = ? WHERE id = ?";
jdbcTemplate.update(updateQuery, user.getName(),
user.getId());
```



Listing 10. Example of the usage of Spring's `JdbcTemplate`.

When diving deeper into Spring's internals, we can see that the `update` method uses prepared statements to set up the query execution.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that Spring *provides helpers* for mitigating SQL injection vulnerabilities. We can deduce that Spring provides the developer with good tools to protect against SQL injection vulnerabilities, given that the tools are used correctly. As with any tool, they can leave us vulnerable if used incorrectly. From a technical perspective, nothing is stopping a developer from implementing the above method in an insecure manner and introducing a SQL injection vulnerability.

```
String updateQuery = "UPDATE users SET name = '" +  
user.getName() + "' WHERE id = " + user.getId();  
jdbcTemplate.update(updateQuery);
```

Listing 11: An example of the same functionality used incorrectly introducing a vulnerability.

### 5.3.2 Cross-site scripting - XSS

Web applications and pages often leverage user-generated content or input to make the application more tailored for the individual user. A trivial example of this is to greet a user by their name when they log into a website. However, from a browser's perspective, the browser does not know what content is user-provided and what is part of the application - the browser simply renders all of the HTML content it receives. Thus, what happens if the user-provided content is not so innocent after all?

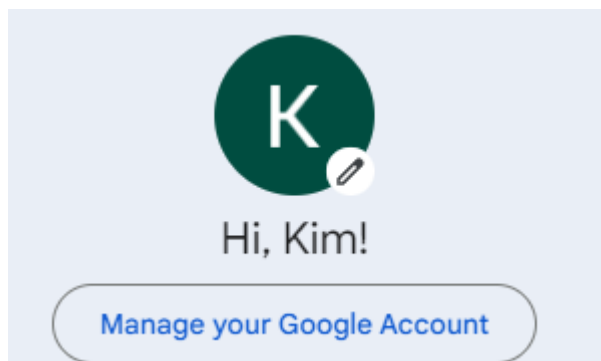


Figure 9. Google greets its logged-in users by the user's first name. The name is user-provided data, and thus, could contain malicious content.

Cross-site scripting (XSS) attacks are a type of injection attack where an attacker is able to inject malicious code into a vulnerable website. The attack occurs when the malicious code is executed on another end user's browser (OWASP, n.d.-I). The term cross-site scripting was coined in 2000 when CERT released its advisory detailing the vulnerability (CERT, 2000).

One of the most known XSS attacks occurred in 2005, a case so famous that it received a name of its own: the Samy Worm. A teen named Samy Kamkar was using a then-popular social networking site called MySpace. MySpace contained a feature allowing its users to customize their user profile pages. Samy noticed that the customization feature allowed him to use HTML and custom JavaScript. He worked on a script that would automatically add him as a friend by anyone visiting his profile page. Samy quickly noticed this was inefficient, as few were visiting his profile page. He then modified his script to copy itself on the visitor's profile page, becoming a self-propagating worm. The script spread like wildfire, and Samy received more than a million friend requests within the first 24 hours. This incident forced MySpace to shut down its site for them to understand what was happening and purge the site of the worm (Franceschi-Bicchierai, 2015).

A cross-site scripting vulnerability also played a role in the US presidential election campaigns in 2008. During the Pennsylvania Democratic primary election in April 2008, a hacker found an XSS vulnerability in Barack Obama's election campaign website. By crafting a simple redirect script, the attacker could forward any user who visited the community blog section of Obama's site to Hillary Clinton's website instead (Dignan, 2008).

Another example is from March 2010, when the Conservatives in the United Kingdom launched a website containing a feature that directly embed Tweets with the hashtag #cashgordon on their website. The site did, however, not validate the content of the Twitter posts before embedding them. The failure to validate data from an untrusted source made it possible to launch a cross-site scripting attack simply by including the attack code in a Twitter post. This vulnerability was used, among others, to redirect unsuspecting visitors to other pages on the Internet, such as the Labour Party's website and pornographic pages (Beckford, 2010).

The above examples might feel relatively harmless as to the consequences of an XSS vulnerability, but in reality, the impact might be much more severe. Suppose no validation on user input is made and no content filtering is applied. In that case, it is up to the attacker's imagination in which ways they could leverage the vulnerability. After all, the attack could be any JavaScript the attacker wants to execute on the victim's browser. The script could be, for example, a keylogger to steal user credentials, stealing user cookies for account hijacking or scanning the victim's intranet for sensitive information.

Cross-site scripting attacks can generally be divided into three categories: stored attacks, reflected attacks, and DOM-based attacks, based on how they are performed. The three categories are discussed in more detail below.

#### **5.3.2.1 Stored attacks**

Stored attacks are attacks where a malicious user manages to get the malicious code persisted onto the target server (OWASP, n.d.-I). The attack is possible in cases where the applications allow users to enter data, which will then be stored in the server's database. Examples of such applications could be message forums, comment fields, or, for example, in the attack against Barack Obama's site, a blog engine open to community members. Consider a social networking website that allows users to post their own content. An attacker could post a message containing the malicious code. The code would be persisted on the server and executed by any subsequent users to whom the malicious user's post is shown.

#### **5.3.2.2 Reflected attacks**

Reflected cross-site scripting attacks are performed in a way where the malicious code is sent to the server and rendered on the page without the malicious code being stored on the target server (OWASP, n.d.-I). An example of a typical reflected attack is where the malicious code is transferred along with the HTTP request in the form of a URL parameter. Consider a search engine that takes the search keywords as a URL parameter. A valid request searching for sites with information about cross-site scripting could look like this:

```
http://search.engine/?keywords=xss
```

The results page would contain the text "Sites found with the keywords 'xss':" along with a list of links to sites matching the keywords. Consider what would happen if the keywords were changed as follows.

```
http://search.engine/?keywords=<script>alert('xss');</script>
```

With the URL above, an attacker would try to inject a piece of JavaScript into the page. If the site includes the keywords unvalidated on the results page, the script tag and its content would be embedded into the page source code and executed as a part of the page. A script that alerts some text might irritate the user, but it does not harm the user from a security perspective. The same

vulnerability can be used to steal the user's cookies from the vulnerable site. For example, the attacker could use the cookie to launch an automated attack to hijack the victim's user account on the target. The following URL would create an image tag pointing to a domain controlled by the attacker. As a URL parameter to the image, we add the user's cookie contents, thus allowing the attacker to steal the cookies (Stuttard & Pinto, 2011, p. 610)

```
http://search.engine/?keywords=<script>var+i=new0Image;+i.src=
"http://evil.site/"%2bdocument.cookie;</script>
```

This attack is more difficult to execute because the exploiting script is not stored on the target server but is transferred as a part of the URL. Only users who open the URL containing the malicious script can become attack victims. To make people open the malicious URL will require some social engineering. The attacker could, for example, try to disguise the URL so that it is not immediately recognized as malicious by the potential victim. For example, the attacker could disguise the URL by using a URL shortener service.

#### 5.3.2.3 DOM-based attacks

The third type of XSS vulnerability was described in 2005 by security researcher Amit Klein. The vulnerability is quite similar to a reflected XSS vulnerability. However, the mechanism is slightly different. Stored and reflected attacks rely on malicious code being sent to the server, which then embeds the code along with the rest of the page content in the HTTP response. DOM-based attacks do not require the malicious code to be sent to the server where countermeasures usually are applied. Instead, it relies on the vulnerable site to have client-side code (JavaScript) that uses data from the document object (or other objects the attacker can influence) in an insecure manner (Klein, 2005).

Klein demonstrated the vulnerability through a simple web page that shows the visitor's name on the page.

```
<HTML>
<TITLE>Welcome!</TITLE>
Welcome,
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
</HTML>
```

Listing 12. Example of a web page that reads a URL parameter using JavaScript and shows the name on the page.

The page's code is static, in the sense that the generated code returned at the end of the HTTP request is always the same. However, the page contains a small JavaScript portion that reads the URL parameters and parses the user-entered name. The name is then dynamically added to the page to show the individualized welcome message. Typically, this page could be called with the following URL:

```
http://some.site/vulnerable_page.html?name=Joe
```

The page would print out the text "Welcome, Joe". By changing the name parameter, the script can be used for printing out arbitrary text, for example, JavaScript. The following example would show the user's cookie contents in an alert box.

```
http://some.site/vulnerable_page.html?name=
<script>alert(document.cookie);</script>
```

The difference between a DOM-based attack and a reflected attack lies in whether the untrusted user-supplied data is incorporated into an HTTP response generated by the server or if it is processed on the client side, subsequently updating the Document Object Model (DOM) with an unsafe JavaScript call. OWASP has also used this differentiation to simplify the XSS vulnerability types into two categories: Server XSS and Client XSS (OWASP, n.d.-m).

#### 5.3.2.4 Countermeasures

Countering cross-site scripting vulnerabilities can be challenging, as there is no specific single action the developer should take, as the appropriate countermeasures are context-dependent. User data might be added into several contexts, not just as text to the page. For example, the developers might want to use user-supplied data to be inserted into the CSS code to allow end users to modify the look and feel for some parts of the application, or when providing a URL to another site, that URL is used in the `<a>` tag's `href` attribute to make the URL a clickable link.

To avoid XSS vulnerabilities, the developers have to make sure that user-provided data cannot escape their intended context or allow the execution of unintended code. OWASP provides developers with a cheatsheet on how to prevent XSS vulnerabilities. On a high level, these

mechanisms can be summarized into the following three actions (OWASP, n.d.-n).

**Encoding.** User-provided data should be encoded in the appropriate context-dependent way before using the data. For example, for HTML contexts (such as `<div>$userContent</div>`), HTML entity encoding should be performed on the user-provided data. HTML entity encoding transforms characters such as `<` or quotation marks to `&lt;` and `&quot;`, respectively (OWASP, n.d.-n).

**HTML Sanitization.** In some use cases, the developers need to allow the end user to enter custom HTML code. For example, a content management system (CSM) might want to allow the administrator to change the structure of the page by using a What You See Is What You Get (WYSIWYG) editor. Applying HTML entity encoding on the resulting HTML would prevent XSS vulnerabilities but also break the functionality. In these situations, the correct approach is to sanitize the HTML to remove any unsafe content. For this, the developer should use an existing library, such as DOMPurify, and not try to implement the sanitization themselves (OWASP, n.d.-n).

**Safe Sinks.** Where possible, the developers should prefer using so-called safe sinks. These are variables that the browser will interpret as text and not as code, effectively leaving the encoding up to the browser. An example would be using the `elem.textContent` or `elem.value` instead of `elem.innerHTML` (OWASP, n.d.-n).

### 5.3.2.5 Framework level mitigation

According to Vaadin's documentation, the framework has built-in protection against cross-site scripting vulnerabilities (Vaadin, n.d.-e). The framework relies on the *safe sinks* approach, using browser APIs, such as `innerText` instead of `innerHTML`) that interpret the content as text instead of HTML. To allow some valid use cases for using custom HTML and JavaScript, Vaadin provides APIs allowing HTML but also warns the developers to sanitize the content before passing it to the insecure methods.

```
Div div = new Div();
div.getElement().setProperty("innerHTML", "<b>This IS
bolded.</b>");
div.add(new Html("<b>This IS bolded.</b>"));
new Checkbox().setLabelAsHtml("<b>This is bolded too.</b>");
```

Listing 13. Example of insecure methods that allow raw HTML content. The developers using these APIs are responsible for sanitizing the content before using the methods.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that *Vaadin provides an out-of-the-box mitigation* for XSS vulnerabilities when the developers do not explicitly go outside the defined boundaries.

## 5.4 A04:2021-Insecure Design

Insecure design refers to vulnerabilities stemming from design decisions that have not taken security into account. This category separates between insecure design and insecure implementation. A flaw in the design cannot be remedied by any implementation that adheres to the design, as the vulnerability might lie, for example, in the business logic and not in the implementation code (OWASP, 2021f).

An example of an insecure design is how users recover accounts when they forget their passwords. A commonly seen variant is to ask users for known secrets during the account registration phase, such as, “*What was the name of your first pet?*”, and use these questions to validate identity when recovering an account in case of a forgotten password. This approach is inherently insecure, as the answers to these “secret” questions might be known to people close to the victim or might be exposed through open-source intelligence or social engineering. Hence, NIST 800-63b prohibits using a memorized secret to obtain a new list of look-up secrets (Grassi et al., 2017, p. 47).

To counter security-related design flaws, OWASP recommends, among others, to collaborate with application security experts to implement a robust and secure development lifecycle, and to help evaluate and design security and privacy-related controls. It is recommended to use threat modeling for critical authentication, access control, business logic, and key flows to identify potential security problems already in the design (OWASP, 2021f).

### 5.4.1 Framework level mitigation

While this category is extremely broad in its content, highly context-dependent, and implementation agnostic and, thus, mostly out-of-scope for this thesis, it is worth noting that the used frameworks' architecture does provide protection against some limited aspects of this category. Applications built with Vaadin are thin-client applications, meaning the code executed in the client (browser) is extremely limited and does not contain any application or business logic. It means some security controls, such as input validation, are automatically implemented on the server side.

The example application built for this thesis has a non-functional requirement stating that data integrity needs to be ensured. The purpose was

to have means in place that ensured that an attacker could not modify the course evaluations without leaving a trace. The implementation for this is a design decision, and neither Vaadin nor Spring provides any functionality to help the developers ensure the data's integrity. Proper data integrity is a combination of implementing proper access control, implementing audit logs that, for example, use cryptographic signing of the changes, and generally adopting security best practices.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that this category is *Not Applicable*.

## 5.5 A05:2021-Security Misconfiguration

The applications themselves and the frameworks used for creating the application are becoming increasingly configurable. Configurability serves multiple purposes, providing flexibility for the vendors and developers using the tools. For example, feature flags (the ability to enable or disable certain features using configurations) allow the developers to cover a larger amount of varying use cases without having to release different software versions. However, a framework vendor cannot know which features a user wants to use; hence, by default, all features might be enabled, or maybe the framework vendor has made an educated guess on which features are *typically* used and enable those by default while lesser used functionality is disabled by default.

The discussion on whether or not to enable a feature by default boils down to usability versus security. Having a feature enabled by default makes it work out-of-the-box, making it *easy* for the user to start using the feature. This does come with the caveat that most users do not use all features, meaning there might be unused features that stay enabled even in production, allowing attackers a larger attack surface.

Another perspective is that the same software has different configuration needs depending on the development lifecycle. While developing an application, it is important for the developers to get as much information as possible about what is happening in the application, and hence, it is essential to have maximum transparency into the application's inner workings - what errors are occurring, stack traces of the errors or what software components are being used. While this is beneficial for the developers, a production version should not have the same level of transparency, as all the same information would disclose details to a potential threat actor, making it easier to perform a successful attack against the application.

The fifth category in OWASP's top ten list is about how the application and the server it is running on are configured - is the whole stack configured so that no unnecessary features are enabled, default passwords have been



changed, and no unnecessary details about the inner workings leak out (OWASP, 2021e).

### 5.5.1 Analysis of frameworks' default configuration

In the scope of this thesis, server-level configuration will be considered out-of-scope, and the focus will be purely on the frameworks used. Testing the application for misconfiguration focuses on improper input validation, which might cause stack traces to leak to the client (OWASP, n.d.-g), making sure debug modes are disabled and HTTP headers are handled appropriately (OWASP, 2021d, 62-63).

The default build of a Vaadin application is meant for development purposes and is in so-called debug mode. This mode is meant to provide the developers with as much information as possible using transparency. For example, it reveals information about the different software versions used.

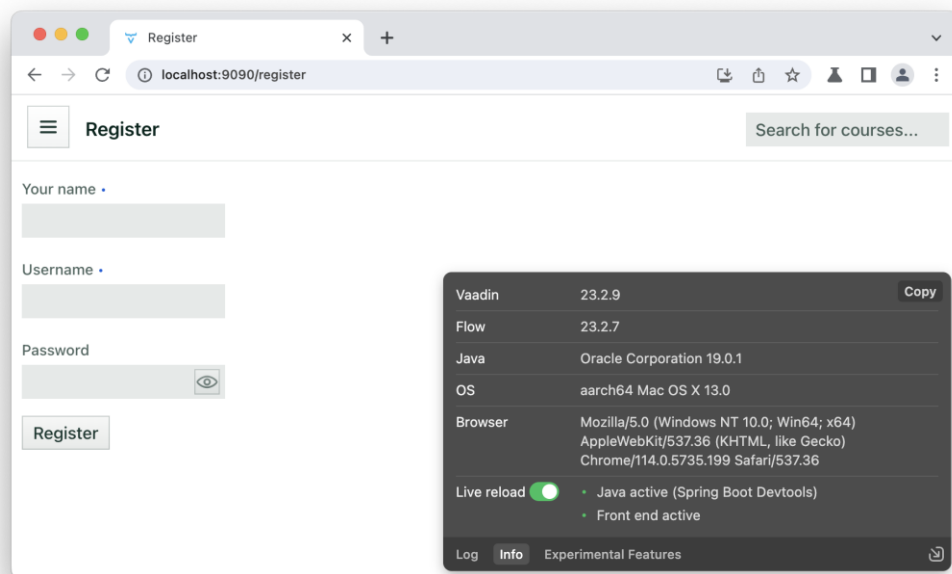


Figure 10. The default build of a Vaadin application is meant for development time, enabling an information window showing the application's internal details.

An attempt to break the input validation, send improper requests, or cause a runtime exception to occur on the server does not reveal any stack traces to the client side, even in development mode. Accessing unauthorized

files or folders on the server is prevented even in development mode, but it does reveal all accessible paths in the application.

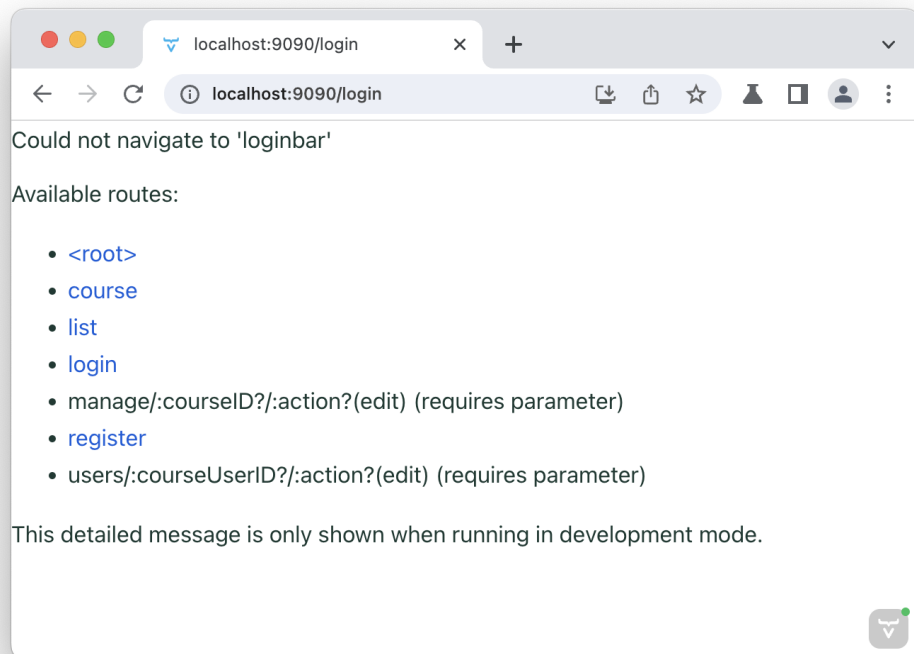


Figure 11: The development time build reveals all available paths in the application, information that should not be disclosed in a production build.

Running Vaadin in a production mode is done by running a build using a production profile (Vaadin, n.d.-c). When testing the application's behavior with a production build, none of the sensitive information was disclosed as it was in a development mode build.

While most security headers were present, a Content Security Header policy was seen to be missing. However, this is typically configured in the application server and not in the application (although possible in the application as well).

As Vaadin is a programming framework, the framework package does not contain any example applications, administrative pages, or default passwords that should be disabled or removed.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that this category is *mitigated by the framework through configurations*.

## 5.6 A06:2021-Vulnerable and Outdated Components

When creating software, we rarely build it without using any third-party dependencies. It would be quite impractical and time-consuming to write software without using any frameworks or libraries, and for any application with even a minimal amount of complexity, it would be next to impossible. Using third-party libraries does come with a caveat - they might contain security vulnerabilities we may not be aware of. A vulnerability in a third-party library might thus expose one's application to the same vulnerability.

An example of such an incident is the vulnerability known as log4shell. It was a vulnerability publicly reported on December 10th, 2021, in the highly popular logging framework log4j (CVE, 2021). The vulnerability allowed an attacker to gain remote code execution (RCE) by injecting a prepared string into the logging. A remote code execution vulnerability allows an attacker to run arbitrary commands on the victim machine, and thus, NIST categorized this vulnerability as critical, the highest possible severity level (NIST, 2021). On December 11th, 2021, Microsoft's Threat Intelligence team reported that they had observed multiple tracked nation-state activity groups originating from China, Iran, North Korea, and Turkey using the vulnerability (Microsoft, 2021). Just days later, it was reported that the vulnerability was being used to infect machines by cryptominers and botnets (Kimayong, 2021).

The sixth category in OWASP's top ten list is about using third-party dependencies without maintaining the dependencies and updating them as vulnerabilities become known, thus exposing one's application to the vulnerability (OWASP, 2021g). This vulnerability category is not directly related to the frameworks used or their features, but rather, how an application is maintained and how dependencies are taken into account in the secure development lifecycle. For example, OWASP provides a free, open-source Software Composition Analysis (SCA) tool for scanning a project's dependencies and checking the used versions against known vulnerabilities (OWASP, n.d.-h). The tool can be easily integrated into the project's build process by adding a maven dependency and enabling the dependency checker plugin. The plugin can be configured to fail the build if a vulnerability of the given criticality level is found in one or more of the dependencies used in the project.

Below is a summary of a software composition analysis scan to the example project, with the default dependencies created by Vaadin Start.

### **fi.abo.kim.mycourses:mycourses:1.0-SNAPSHOT**

Scan Information (show less):

- *dependency-check version:* 8.3.1
- *Report Generated On:* Sat, 29 Jul 2023 14:00:03 +0300

- *Dependencies Scanned*: 2051 (777 unique)
- *Vulnerable Dependencies*: 26
- *Vulnerabilities Found*: 59
- *Vulnerabilities Suppressed*: 0
- *NVD CVE Checked*: 2023-07-29T13:59:29
- *NVD CVE Modified*: 2023-07-29T13:00:01
- *VersionCheckOn*: 2023-07-29T13:59:36
- *kev.checked*: 1690628378

As it can be noted, there is a high number of dependencies for a relatively trivial application, and those dependencies contain a non-trivial number of vulnerabilities, some of which are considered critical-level vulnerabilities (see Appendix A). Even though the dependencies contain critical-level vulnerabilities, it is not as straightforward to conclude that the example application also contains critical vulnerabilities that can be leveraged. Some of the vulnerabilities listed are only exploitable in specific deployment environments with specific configurations, which would not be applicable to the example application. This means there can be a high level of false positives, and any conclusions about the vulnerabilities' exploitability would need to be made only after a careful analysis.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that this category is *Not Applicable* as dependency management is more related to the development pipeline than the frameworks used.

## 5.7 A07:2021-Identification and Authentication Failures

Authentication often has a central role in an application. It identifies the user, based on which the application determines what data and features the user can access. The ability to gain unauthorized access critically impacts the system's reliability. The *Identification and Authentication Failures* category highlights vulnerabilities and design flaws related to user authentication. These cover technical flaws, for example, related to session management, allowing session fixation attacks, stealing of session identifiers, or mismanagement of the session lifecycle. Design flaws might include missing protection against brute force attacks, password spraying or credential stuffing, allowing weak passwords, or lacking multi-factor authentication (OWASP, 2021h).

### 5.7.1 Framework level mitigation

Vaadin's recommended way of handling authentication is through Spring Security, for which Vaadin provides an integration. Spring security provides a number of different ways to authenticate a user, ranging from a traditional username/password combination to OAuth2, SAML 2.0, CAS, JAAS, and X509 certificate-based authentication (Spring, n.d.-e).

ASVS chapters 2 and 3 dictate several requirements for both authentication and session management. The responsibility of implementing these requirements depends on the chosen authentication method. For example, suppose the application uses OAuth2 to implement single sign-on through social sites such as Google, Twitter, Facebook, or GitHub. In that case, requirements related to password management and recovery are the responsibility of those third-party sites, not the application. Let us first explore requirements related to session management, which are agnostic to the authentication method used.

#### 5.7.1.1 Session management

Managing the HTTP session is out-of-scope in this thesis, as the J2EE framework manages it. Thus, vulnerabilities such as session fixation or ensuring that the generation of session tokens is cryptographically sound would not be something that is in control of the developers.

Vaadin Flow applications store their application state as a session variable on the server side. Ensuring the session's lifecycle and cookies are managed correctly is essential for these applications. ASVS defines the appropriate requirements in sections 3.1-3.4. Reflecting against ASVS L1 requirements, we need to verify that a new session ID is generated upon the user's authentication and at logout and to ensure that using the browser's back button does not re-authenticate the user. Reviewing the HTTP responses shows that the previous sessions are invalidated, and new session IDs are generated upon both login and logout actions.

ASVS also recommends that session cookies use the secure attribute, ensuring that the cookie is transmitted only over HTTPS and that the `sameSite` attribute property is set to limit exposure to cross-site request forgery attacks. With default settings, the session cookies look as follows.

```
Set-Cookie: JSESSIONID=219A218A83FB6E3286314A72E645DC6A;  
Path=/; HttpOnly
```

As seen, neither the `secure` nor the `sameSite` attributes are defined by default. However, these can easily be configured in Spring's settings using the following properties:

```
server.servlet.session.cookie.same-site=Strict
server.servlet.session.cookie.secure=true
```

We can now see that the cookie is limited to HTTPS only and will only be submitted to the server when the request originates from the same domain.

```
Set-Cookie: JSESSIONID=1DD0D02B5A4C86BC32BBEAF25890B14D;
Path=/; Secure; HttpOnly; SameSite=Strict
```

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that session management vulnerabilities are partially *mitigated by the framework through configurations* (cookie settings) and partially *helpers are provided by the framework* (session invalidation on login/logout when the helpers are being used).

#### 5.7.1.2 Authentication when not using external authentication providers

If we do not use external authentication providers but rather decide to implement the authentication, user and password management ourselves, then ASVS sets a number of additional requirements. Vaadin Flow and Spring Security provide means for authenticating a user against a username/password combination but do not provide any built-in functionality for, for example, managing users, password recovery, multi-factor authentication, or protection against brute force attacks.

Vaadin Flow does provide some means to help fulfill some of ASVS's requirements, such as requirement 2.1.12 "*Verify that the user can choose to either temporarily view the entire masked password, or temporarily view the last typed character of the password on platforms that do not have this as built-in functionality.*". This requirement is built into Vaadin's PasswordField component.



Figure 12. Vaadin provides a PasswordField component that allows the unmasking of a password, thus filling the ASVS 2.1.12 requirement.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that *no mitigation support is provided* when it comes to

the secure implementation of authentication without using external authentication providers. Even though the PasswordField component does fulfill one of the ASVS criteria, it is not enough to conclude that, on a general level, the framework would provide helpers for managing this vulnerability category.

## **5.8 A08:2021-Software and Data Integrity Failures**

This category deals with integrity. In other words, the software we use and the data we receive and manage are what we expect them to be and are not maliciously modified in any way.

### **5.8.1 Software integrity**

Let us first discuss the integrity of software. Earlier, we discussed the risks related to using outdated third-party components containing known vulnerabilities, but that is not the only risk related to using third-party libraries. Suppose we do not manage our build pipeline and our dependencies securely. In that case, it can be possible for an attacker to intentionally introduce a vulnerability or a backdoor into our application through third-party libraries. Such a vulnerability might occur if an application loads third-party dependencies from a source outside the intended control sphere. Let us consider application A, which loads a third-party JavaScript library hosted on another server, for example, on a Content Delivery Network server (CDN). If an attacker gains access to the CDN server, they could upload a modified version of the JavaScript library application A uses, containing malicious code, such as a JavaScript-based key-logger. If application A does not sufficiently ensure the authenticity of the loaded code, it becomes vulnerable itself. These types of attacks are called supply chain attacks. Sonatype published their research results in their annual State of the Software Supply Chain report, where they estimated that supply chain attacks in open-source software have grown by an average of 742% annually in the past three years (Sonatype, 2023).

The mitigation strategies for supply chain attacks revolve around ensuring that non-vetted code does not end up in the application, whether through a dependency or not. In practice, this means that one should never use untrusted sources for fetching dependencies, use cryptographically secure hashes to verify the authenticity of the content downloaded and implement a secure CI/CD pipeline, including code review practices to make sure unintended code is not injected in the repository (OWASP, 2021i). For example, suppose an application uses a CDN server to serve its resources.

For external resources, it is recommended to use subresource integrity checks to make sure that the downloaded files have not been modified. The verification can be done by calculating a hash of a vetted resource and then including the hash in the script tag's integrity attribute (Mozilla, n.d.-b). The browser will verify that the hash of the file has not changed and thus ensure that the library has stayed unmodified from when it was vetted.

```
<script src="https://example.com/example-framework.js"
integrity="sha384-
oqVuAfXRRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4Jw
Y8wC" crossorigin="anonymous"></script>
```

Listing 14. An example of the integrity attribute, which helps ensure that libraries hosted on a third-party server have not been modified.

#### 5.8.1.1 Framework level mitigation

Supply chain attacks are mostly out of scope for this thesis, as they are more related to build pipelines and dependency management. Vaadin Start provides an application stub using Maven for dependency management. Default dependencies come either from Maven Central or Vaadin's repository for addons.

Reviewing run-time dependencies, we can see that a Vaadin Flow application provides all JavaScript resources locally and does not rely on Content Delivery Network servers; thus, subresource integrity checks are not used.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that the category is *Not Applicable*.

#### 5.8.2 Data integrity

The second part of this category is the integrity of data. An application should never trust data from an untrusted source to be valid and non-malicious. For web applications, this contains but is not limited to, any data originating from the browser.

A trivial example of this vulnerability is trusting a cookie's values in security-critical functions without ensuring they have not been modified (for example, by using a cryptographic hash to sign the values) (CWE, 2006c). For example, this could be storing the user details in a cookie and relying on those values for authentication or authorization. Simply modifying the cookie value could thus allow the attacker to bypass the authentication and authorization



controls easily (see section 5.1 Broken Access Control and 5.4 Insecure Design).

Another more traitorous vulnerability is the deserialization of untrusted data. In some situations, it is convenient to pass objects around by serializing the object at the sender's end and deserializing it by the receiving party. Java allows its objects to be serialized, for example, into a byte array, which can be encoded into a string. This string can be used as any other string; it can be stored in a database or passed to the browser. When we want to use the object again, the string can be decoded back into a byte array, which can then be deserialized into an ordinary Java object. Serialization is commonly used for caching, persistence, or for clustering.

The vulnerability comes into play if there is a possibility that a malicious user could modify the serialized object, and the server then deserializes the object without verifying its contents first. In the worst case, this would allow an attacker to execute arbitrary code on the server, leading to a so-called Remote Code Execution vulnerability, which is considered the most critical type of vulnerability. The popular continuous integration server, Jenkins, was found in 2015 to have a remote code execution vulnerability due to unsafe deserialization. By crafting a specialized serialized Java object, an attacker could execute any arbitrary code on the server (CVE, 2015).

#### **5.8.2.1 Framework level mitigation**

Vaadin's thin-client architecture might make it less prone to data integrity vulnerabilities. Vaadin stores the application's state on the server side, meaning unnecessary information is never sent to the client. For example, for authentication and authorization, user information, such as the logged-in user's ID or roles, is not transmitted to the client. This type of information is stored as a session variable, and the only cookie the browser uses is the session ID.

Vaadin communicates between the server and the client using remote procedure calls, which allows the browser to call on server-side methods and pass arguments to those. However, Vaadin is using a whitelisting approach to limit which methods can be called, and it limits the data types that can be used as arguments in a method call. To publish a server-side method for the client to use, it needs to be annotated using the `@ClientCallable` annotation and accepts only a few basic data types as arguments (Vaadin, n.d.-d). No complex data structures are passed between the client and the server, which would need a serialization/deserialization scheme.

While Vaadin limits the data types to a few basic ones and validates that the data it receives corresponds to these types, the framework cannot control the design decisions a developer might make. For example, Vaadin

does not (and should not) limit what type of information a developer decides to store in a browser cookie, which might expose the application to data integrity vulnerabilities.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that data integrity failure protection is categorized as *helpers are provided by the framework*. The argument for this categorization is that Vaadin’s architecture does provide some protection. However, in the end, it is the developer’s design decisions that impact what a framework can and cannot manage on behalf of the developer.

## 5.9 A09:2021-Security Logging and Monitoring Failures

Logging is rarely the source of a vulnerability, but it plays a vital role in detecting and investigating breaches. Properly designed logging and active monitoring of application logs can help us stop breaches to the application as they occur. On the other hand, the lack of appropriate and sufficient logging makes it impossible to detect ongoing breaches or even to investigate them afterward. Logs are the breadcrumbs the incident response teams use for tracking down what has happened in the application and who might be responsible for it.

What exactly should or should not be logged is context-dependent. The National Institute of Standards and Technology has released recommendations on how to perform computer security log management (Kent & Souppaya, 2006). On a high level, an application should log events related to user accounts (such as both failed and successful login attempts, password changes, or use of privileges), usage information (the number and size of various transactions), and significant operational actions (such as application failures or configuration changes). By actively monitoring the logs, we are able to detect, for example, brute force attacks or significant data transfers and perform countermeasures as they are happening (for example, locking of accounts). An appropriate log message should contain relevant (but not sensitive) information, such as which action was performed, which user did the action, and from which IP address the action was performed.

The log messages must be sufficiently encoded to avoid log forging and log injection attacks. An example of this is the log4shell vulnerability discussed in section 5.6.

Security logging is not strictly a technical feature but rather a design question, and hence, it cannot be “mitigated” on a framework level. That said, for the security logs to be useful, they need to be monitored (which is out-of-scope in this thesis), and for the monitoring to work, the logs need to be in a format that is easily understood by the monitoring tools. Spring Boot uses Apache Commons Logging for all internal logging, allowing developers to

choose the underlying log framework implementation. Spring Boot provides default configurations for Java Util Logging, Log4J2, and Logback (Spring, n.d.-f).

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that the category is *Not Applicable*.

## 5.10 A10:2021-Server-Side Request Forgery

Modern web applications often implement features that are convenient to the end user. A commonly seen feature in social media applications is the previewing of links. The end user provides the application with a link, the server makes a request to the provided URL, interprets the response content, condenses it into a format that can give a glimpse into the URL's content, and then provides the end users a preview of the content of the URL. This feature is convenient for other users to have an idea of what content is behind a link without having to visit the site.

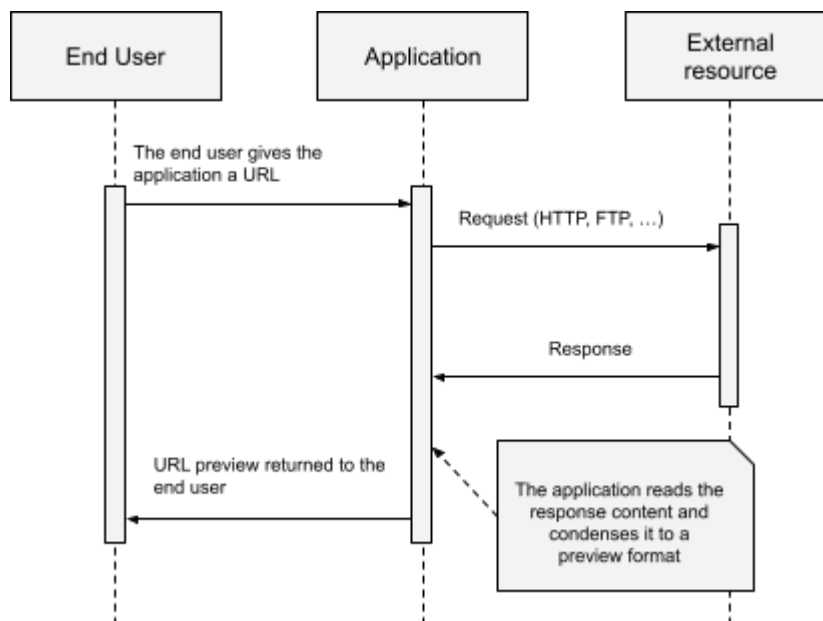


Figure 12. Example sequence flow of an application showing a preview of a user-provided URL.

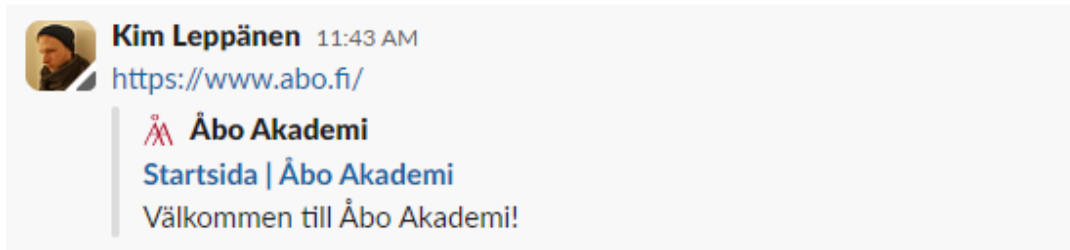


Figure 13. The popular instant messaging application Slack provides a preview of the content of links posted by users.

Another typical use case is integrating an application into an external system using webhooks. A webhook is simply a callback function to an external API, which is called when specific conditions are met. An example of this could be an instant messaging application that scans for specific keywords. When a specific keyword, such as “#help” is noticed, the application makes an API call to an external system, providing the system with messages containing the keyword. The feature can be used, for example, to automate the creation of support tickets when people request help in a chat. The sequence flow is relatively similar to the first example, except the call is now made to a *pre-configured* URL. The process might even be almost invisible to the end user, who might only receive an email verification for creating a support ticket.

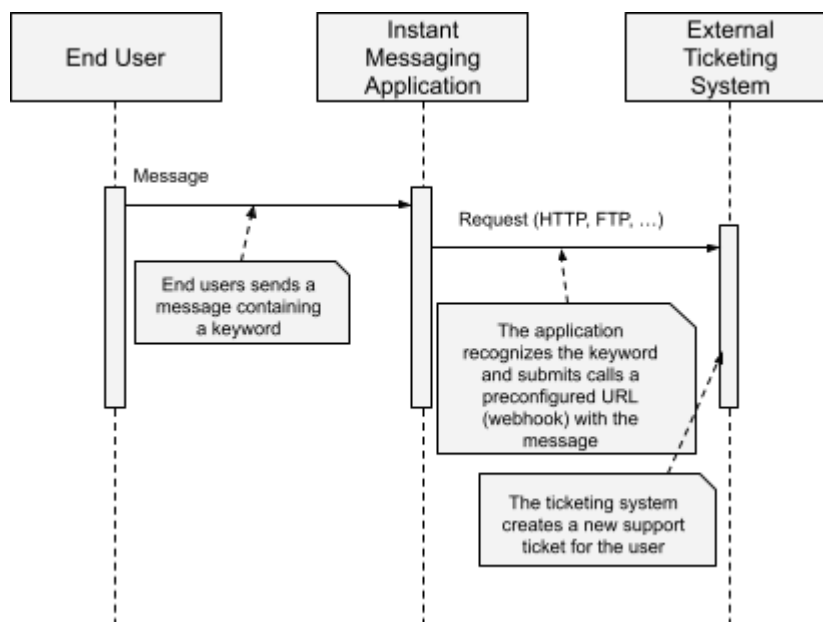


Figure 14. An example of the flow for an application to notice a keyword in a user message and use a webhook to create a support ticket in an external system.

As with many other vulnerabilities, the problems occur when user-provided input is not properly validated, and a malicious user behaves in an unexpected way. In its simplicity, a Server-Side Request Forgery (SSRF) attack is when an application allows an attacker to cause server-side requests to unintended locations (OWASP, 2021j).

There are multiple ways an attacker can leverage the vulnerability. A typical way to exploit the vulnerability is to gain access to resources that otherwise would be out of reach for the attacker. At its simplest, instead of providing a web URL, an attacker could try to use a different protocol and access local files, for example, `file:///etc/passwd`. Another example is to access an internal server that is not accessible from the public internet. The attack is performed by providing an IP-based URL pointing to an IP in the internal network, such as `https://192.168.0.68/admin` (PortSwigger, n.d.).

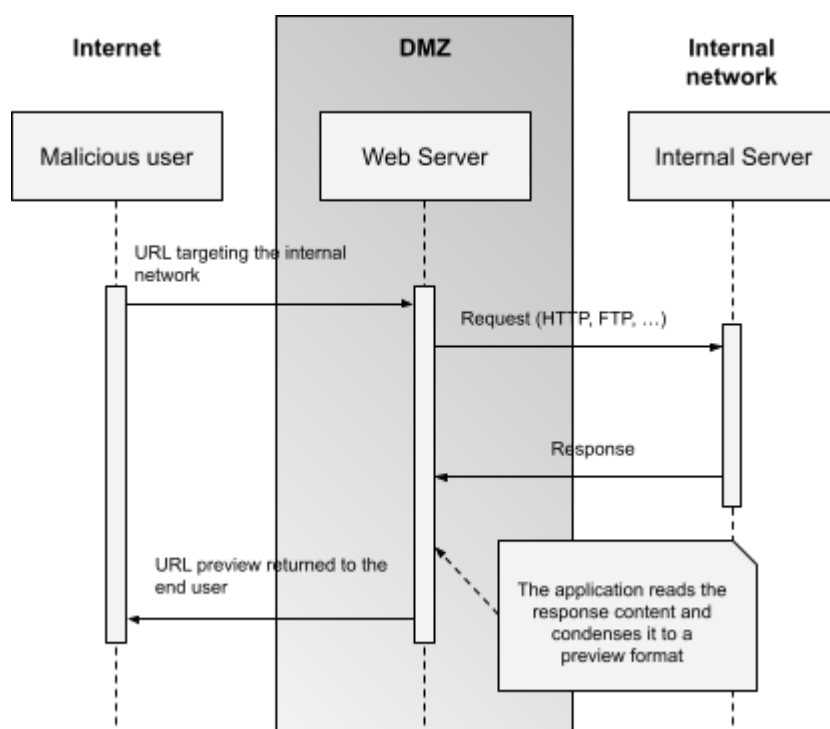


Figure 15. Example of how an SSRF vulnerability could be used to access resources in the internal network outside of the DMZ.

### 5.10.1 Mitigation strategies

At first glance, mitigating server-side request forgeries might seem like a relatively simple task of proper input validation to ensure that only external resources are requested. Although proper input validation and enforcement of

specific ports and protocols make it more difficult to leverage the vulnerability, it is not enough to mitigate the problem. Some techniques to circumvent SSRF protections include obfuscation of URLs, registering a domain but registering the DNS to point to, for example, 127.0.0.1, or using redirects (PortSwigger, n.d.). Especially the two latter approaches cannot be detected as malicious purely by programmatically examining the URL.

An effective strategy for the mitigation of SSRF vulnerabilities requires a combination of application and network-level strategies. Input validation and firewall policies denying the application access to internal resources are recommended (OWASP, n.d.-i).

Jabiyev et al. (2021, pp. 1629-1631) propose a generic solution using a separate server isolated from the internal network with the sole purpose of fetching external resources. As this helper server is isolated from the internal network, any DNS pointers to internal servers or redirects would not be effective. In front of the web application, the research group placed a reverse proxy to detect all external URLs and modify them to point to the helper server. Thus, when the application makes a request to a URL, it is directed to the helper server, which performs the actual request to the external resource. This way, the web application and the web server would never make requests to other sources than the helper server, and thus all the internal resources would be protected.

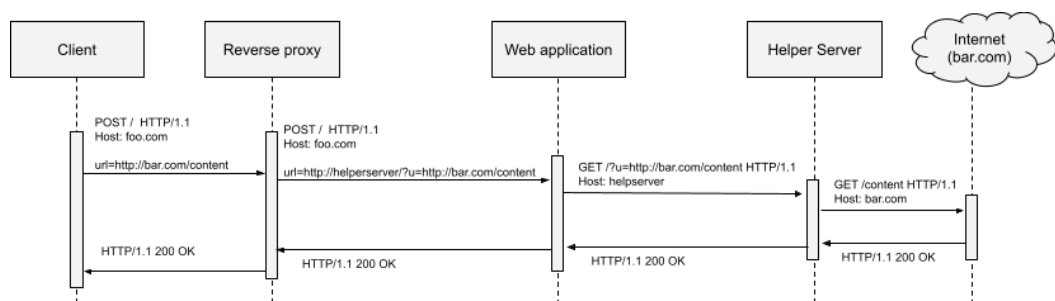


Figure 15.: Overall architecture of the defense solution proposed by Jabiyev et al.

### 5.10.2 Framework level mitigation strategies

At the time of the writing of this thesis, neither Vaadin nor Spring Framework had any functionality that, by default, performed requests that could be vulnerable to server-side request forgery. Spring offers means for making requests to external sources, but the framework is entirely agnostic to how developers use that feature and for which purposes.

There are many different valid use cases for making server-side requests. However, it is extremely hard to generalize on a high level what should and should not be allowed, as the needs might vary significantly depending on the use case. Hence, it would be impractical for the Spring framework to implement limitations that may or may not meet the developers' needs. Thus, it is more up to the individual application developers to ensure they implement the appropriate mitigations. If, for example, Vaadin decides to implement a built-in “preview content” component into their framework, then it would be meaningful for Vaadin to implement restrictions already into the framework, if nothing else, at least with default configurations that limit the end users possibilities to abuse the feature. However, as stated in the previous section, a successful mitigation strategy needs both an application and network-level mitigation.

**Evaluation.** Reviewing against the evaluation criteria defined in section 1.2, we can conclude that Server-Side Request Forgery is categorized as *no mitigation support is provided*. A proper mitigation implementation requires both application and network-level protection, and neither Vaadin nor Spring provides the developers with any additional help for implementing such a protection.

## 6 DISCUSSION AND CONCLUSIONS

This thesis has reviewed the security aspect of developing an application using a modern Java development stack against vulnerabilities defined in OWASP's Top Ten list. We have explored the root causes for the vulnerabilities the recommended mitigation strategies, and reviewed how Vaadin and Spring frameworks either mitigate or help the developers to mitigate the vulnerabilities when developing an application.

As we can see from the OWASP Top Ten category descriptions, the categories are not necessarily just one vulnerability with a specific technical mitigation strategy. A category might be more conceptual and raise awareness of security-oriented thinking (such as the “Secure Design” category) or be a collection of multiple specific vulnerabilities, such as the “Broken Access Control” category. The latter included broader concepts and specific vulnerabilities such as indirect object references and cross-site request forgeries.

This thesis aimed to review the vulnerability categories and then map how frameworks help developers mitigate the vulnerabilities. The mapping was done against four different groups: *Mitigated out-of-the-box by the framework*, *Mitigated by the framework through configurations*, *Helpers are provided by the framework*, and *No mitigation support is provided*. The table below summarizes the mapping done in this thesis.

	Mitigated out-of-the-box by the framework	Mitigated by the framework through configurations	Helpers are provided by the framework	No mitigation support is provided
A01:2021-Broken Access Control			●	
Insecure Direct Object References	●			
Cross-Site Request Forgery (CSRF)	●			
A02:2021-Cryptographic Failures				●
A03:2021-Injection				
SQL injections			●	
Cross-site scripting - XSS	●			
A04:2021-Insecure Design	<i>Not Applicable</i>			
A05:2021-Security Misconfiguration		●		
A06:2021-Vulnerable and Outdated Components	<i>Not Applicable</i>			
A07:2021-Identification and Authentication Failures				
Session management		●	●	
Authentication when not using external authentication providers				●
A08:2021-Software and Data				



	Mitigated out-of-the-box by the framework	Mitigated by the framework through configurations	Helpers are provided by the framework	No mitigation support is provided
Integrity Failures				
Software integrity	<i>Not Applicable</i>			
Data integrity			●	
A09:2021-Security Logging and Monitoring Failures	<i>Not Applicable</i>			
A10:2021-Server-Side Request Forgery				●
<b>Summary, count</b>	<b>3</b>	<b>2</b>	<b>4</b>	<b>3</b>

The first thing we can notice from the summary table is that multiple rows are marked as “Not Applicable”. It indicates that to create an application that is also deployed securely, the development team needs to consider multiple aspects beyond just the application itself. To achieve a high level of security, one needs to consider how the server, the application server, the database, and all other related infrastructure are configured. How is the network segmented, in which way are user accounts isolated from unnecessary privileges that might be abused, or how is the CI/CD pipeline secured?

Out of the evaluated categories and explicit vulnerabilities, only three were mitigated out-of-the-box by the framework (and even two were subpoints to Broken Access Control). Could this number be increased? Surely, but not without a cost. Let us discuss what practical consequences there would be if a framework took a bigger responsibility for the application’s security.

**Default configurations.** Vaadin has chosen the default build to be a development build, not a production build. There is no technical limitation as to why the default build could not be a production build, but rather, it is a conscious decision on Vaadin’s part. Like many other tool vendors, the default settings tend to be geared towards ease of use rather than production use. The reason for this is to help developers get started with the tools with minimum hassle, thus increasing the adoption of the tools. Changing the default settings to suit production use would most likely mean that it is more

difficult to take the tool into use or to get started with it. In Vaadin's case, it would mean that the debug information would not be as easily available, something quite vital for a beginner when getting started.

**Helpers are provided by the framework.** Some vulnerabilities in this category cannot be handled by the framework, no matter what. For example, access control is a good example where the difference between "right" and "wrong" solely depends on the application's intent. A framework cannot know which users should have access to a view and who should be denied access, not without someone configuring this information. Then again, a framework could mitigate SQL injections completely by not allowing the developer to write their own queries. Instead, a framework could force users to use an ORM or allow query building only through typesafe APIs. Some frameworks do exactly this, but the drawback is that the developer loses flexibility and maybe even the ability to define complex (but efficient) queries.

**No mitigation was provided.** Similar to the category above, not everything can be handled by a framework. For example, secure design, by definition, is something that is context-dependent and needs to be *designed* to fit the purpose. Some other more implementation-level categories, such as the implementation of authentication and secure user management, are something a framework could handle (emphasis on the words *a framework*). Frameworks typically specialize in solving one problem (or at least, problems within one domain). For example, Vaadin is built for creating modern web-based user interfaces using Java. However, it is not an authentication framework nor an ORM, even though both are functionalities typically used in a Vaadin application. There are *platforms* that provide developers with everything end-to-end. However, these are often no-code or low-code platforms that reduce the flexibility in what and how things are built or even where the application can be deployed.

Even if a framework could mitigate most vulnerabilities, the recommended approach is to apply the defense-in-depth principle, meaning that we do not have a single point of failure in the application but apply security on several layers. If one layer fails, there is another layer to catch the problem. One approach that has gained popularity is using Web Application Firewalls (WAF). A WAF is a piece of software that is deployed in front of the application itself, and all network traffic goes through the WAF.

The WAF inspects user inputs, and in many instances, it can detect and stop malicious payloads, such as XSS or SQL injection attack attempts. While Web Application Firewalls can be, in many cases, effective, they might give developers a false sense of security. An accidental misconfiguration routing the traffic directly to the application, bypassing the WAF, will expose the application to vulnerabilities. If the vulnerabilities are not mitigated at the root cause, then a WAF is simply a bandaid that might one day accidentally come off, and one might end up losing one's data or worse.

There is no silver bullet to security. To build secure software, the developers need to understand how the tools they are using work, what the responsibility of the framework is, and what the developers' responsibilities are. Developers need to understand what kind of vulnerabilities their applications might be exposed to and what risks those potential vulnerabilities impose. Collaboration is needed with network, system, and security engineers; good DevSecOps practices should be implemented, and proper risk tolerance should be agreed upon with the business stakeholders. After all, the level of security we implement is directly tied to the risks we are willing to take - a hobby project will not need the same level of security as a healthcare system responsible for literally vital functions.

## 7 REFERENCES

Barth, A., Jackson, C., & Mitchell, J. C. (2008). Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 2008, Conference on Computer and Communications Security, 2008, Alexandria, Virginia, USA, October 27-31, 2008* (pp. 75-88). Association for Computing Machinery.

<https://doi.org/10.1145/1455770.1455782>

Beckford, M. (2010, March 23). Conservatives embarrassed as hackers exploit loophole on anti-union website. *Telegraph*.

<https://www.telegraph.co.uk/technology/twitter/7499228/Conservatives-embarrassed-as-hackers-exploit-loophole-on-anti-union-website.html>

CERT. (2000, February 2). *CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests*. 2000 CERT Advisories. Retrieved December 7, 2023, from [https://insights.sei.cmu.edu/documents/507/2000\\_019\\_001\\_496188.pdf](https://insights.sei.cmu.edu/documents/507/2000_019_001_496188.pdf)

Conklin, W. A., & Shoemaker, D. P. (2022). *CSSLP Certification All-in-One Exam Guide, Third Edition*. McGraw-Hill Education.

CVE. (2015, November 25). *CVE-2015-8103*. Retrieved October 22, 2023, from <https://www.cve.org/CVERecord?id=CVE-2015-8103>

CVE. (2021, December 10). *CVE Record*. CVE Record | CVE. Retrieved July 29, 2023, from <https://www.cve.org/CVERecord?id=CVE-2021-44228>

CWE. (2006, July 19). *CWE - CWE-261: Weak Encoding for Password (4.12)*. Common Weakness Enumeration. Retrieved July 24, 2023, from <https://cwe.mitre.org/data/definitions/261.html>

CWE. (2006b, July 19). *CWE - CWE-328: Use of Weak Hash (4.12)*. Common Weakness Enumeration. Retrieved July 24, 2023, from <https://cwe.mitre.org/data/definitions/328.html>

CWE. (2006c, July 19). *CWE - CWE-565: Reliance on Cookies without Validation and Integrity Checking (4.12)*. Common Weakness Enumeration. Retrieved August 20, 2023, from <https://cwe.mitre.org/data/definitions/565.html>

CWE. (2009, March 03). *CWE - CWE-759: Use of a One-Way Hash without a Salt (4.12)*. Common Weakness Enumeration. Retrieved July 24, 2023, from <https://cwe.mitre.org/data/definitions/759.html>

CWE. (2009b, March 03). *CWE - CWE-760: Use of a One-Way Hash with a Predictable Salt (4.12)*. Common Weakness Enumeration. Retrieved July 24, 2023, from <https://cwe.mitre.org/data/definitions/760.html>

CWE. (n.d.-b). *CWE - CWE-352: Cross-Site Request Forgery (CSRF) (4.11)*. Common Weakness Enumeration. Retrieved January 30, 2023, from <https://cwe.mitre.org/data/definitions/352.html>

CWE. (n.d.-a). *CWE - CWE-639: Authorization Bypass Through User-Controlled Key (4.11)*. Common Weakness Enumeration. Retrieved January 12, 2023, from <https://cwe.mitre.org/data/definitions/639.html>

Dignan, L. (2008, April 21). Obama site hacked; Redirected to Hillary Clinton. *ZDNET*. <https://www.zdnet.com/article/obama-site-hacked-redirected-to-hillary-clinton/>

Franceschi-Bicchierai, L. (2015, October 4). The MySpace Worm that Changed the Internet Forever. *VICE*.  
<https://www.vice.com/en/article/wnjwb4/the-myspace-worm-that-changed-the-internet-forever>

Grassi, P. A., Fenton, J. L., Newton, E. M., Perlner, R. A., Regenscheid, A. R., Burr, W. E., & Richer, J. P. (2017, June). Digital Identity Guidelines: Authentication and Lifecycle Management.  
<https://doi.org/10.6028/NIST.SP.800-63b>

Halfond, W. G.J., Viegas, J., & Orso, A. (2006). *A Classification of SQL Injection Attacks and Countermeasures*. Georgia Institute of Technology. Retrieved November 19, 2023, from  
<http://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf>

Hardy, N. (1988, October). The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), 36-38. <https://doi.org/10.1145/54289.871709>

Horner, M., & Hyslip, T. (2017, June 30). SQL Injection: The Longest Running Sequel in Programming History. *Journal of Digital Forensics, Security and Law*, 12(2), 97-108. <https://doi.org/10.15394/jdfsl.2017.1475>

IETF. (1999, June). *RFC 2616 HTTP/1.1*. Hypertext Transfer Protocol -- HTTP/1.1. Retrieved May 6, 2023, from <https://www.ietf.org/rfc/rfc2616.txt>

Jabiyev, B., Mirzaei, O., Kharraz, A., & Kirda, E. (2021). Preventing Server-Side Request Forgery Attacks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (pp. 1626–1635). Association for Computing Machinery. <https://doi.org/10.1145/3412841.3442036>

Kent, K., & Souppaya, M. (2006, September). Retrieved August 3, 2023, from <https://doi.org/10.6028/NIST.SP.800-92>

Kimayong, P. (2021, December 17). *Log4j Attack Payloads In The Wild | Official Juniper Networks Blogs*. Juniper Blogs. Retrieved July 29, 2023, from <https://blogs.juniper.net/en-us/security/in-the-wild-log4j-attack-payloads>

Klein, A. (2005, July 4). *[DOM Based Cross Site Scripting or XSS of the Third Kind] Web Security Articles*. Web Application Security Consortium. Retrieved

December 21, 2023, from

<http://www.webappsec.org/projects/articles/071105.shtml>

Likaj, X., Khodayari, S., & Pellegrino, G. (2021). Where We Stand (or Fall): An Analysis of CSRF Defenses in Web. In *Proceedings of 2021 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2021): October 6-8, 2021, Donostia-San Sebastián, Spain* (pp. 370-385). Association for Computing Machinery.

<https://doi.org/10.1145/3471621.3471846>

Microsoft. (2021, December 11). *Guidance for preventing, detecting, and hunting for exploitation of the Log4j 2 vulnerability*. Microsoft. Retrieved July 29, 2023, from <https://www.microsoft.com/en-us/security/blog/2021/12/11/guidance-for-preventing-detecting-and-hunting-for-cve-2021-44228-log4j-2-exploitation/>

Mozilla. (n.d.). *Origin - HTTP | MDN*. MDN Web Docs. Retrieved July 11, 2023, from <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Origin>

Mozilla. (n.d.-b). *Subresource Integrity - Security on the web | MDN*. MDN Web Docs. Retrieved August 6, 2023, from [https://developer.mozilla.org/en-US/docs/Web/Security/Subresource\\_Integrity](https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity)



MySQL. (n.d.). *MySQL :: MySQL 8.0 Reference Manual :: 8.2.1.19 LIMIT Query Optimization*. MySQL :: Developer Zone. Retrieved November 20, 2023, from <https://dev.mysql.com/doc/refman/8.0/en/limit-optimization.html>

NIST. (2021, December 10). *NVD - CVE-2021-44228*. NVD. Retrieved July 29, 2023, from <https://nvd.nist.gov/vuln/detail/CVE-2021-44228>

OWASP. (2021c, September 21). *A02 Cryptographic Failures - OWASP Top 10:2021*. Retrieved July 24, 2023, from [https://owasp.org/Top10/A02\\_2021-Cryptographic\\_Failures/](https://owasp.org/Top10/A02_2021-Cryptographic_Failures/)

OWASP. (2021b, September 24). *A01 Broken Access Control - OWASP Top 10:2021*. Retrieved November 19, 2022, from [https://owasp.org/Top10/A01\\_2021-Broken\\_Access\\_Control/](https://owasp.org/Top10/A01_2021-Broken_Access_Control/)

OWASP. (2021e, September 24). *A05 Security Misconfiguration - OWASP Top 10:2021*. Retrieved July 26, 2023, from [https://owasp.org/Top10/A05\\_2021-Security\\_Misconfiguration/](https://owasp.org/Top10/A05_2021-Security_Misconfiguration/)

OWASP. (2021f, September 24). *A04 Insecure Design - OWASP Top 10:2021*. Retrieved July 27, 2023, from [https://owasp.org/Top10/A04\\_2021-Insecure\\_Design/](https://owasp.org/Top10/A04_2021-Insecure_Design/)

OWASP. (2021g, September 24). A06 Vulnerable and Outdated Components - OWASP Top 10:2021. Retrieved July 29, 2023, from [https://owasp.org/Top10/A06\\_2021-Vulnerable\\_and\\_Outdated\\_Components/](https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/)

OWASP. (2021h, September 24). A07 Identification and Authentication Failures - OWASP Top 10:2021. Retrieved July 30, 2023, from [https://owasp.org/Top10/A07\\_2021-Identification\\_and\\_Authentication\\_Failures/](https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/)

OWASP. (2021i, September 24). A08 Software and Data Integrity Failures - OWASP Top 10:2021. Retrieved August 4, 2023, from [https://owasp.org/Top10/A08\\_2021-Software\\_and\\_Data\\_Integrity\\_Failures/](https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/)

OWASP. (2021j, September 24). A10 Server Side Request Forgery (SSRF) - OWASP Top 10:2021. Retrieved November 5, 2023, from [https://owasp.org/Top10/A10\\_2021-Server-Side\\_Request\\_Forgery\\_%28SSRF%29/](https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/)

OWASP. (2021a, September 24). *The OWASP Top Ten*. The OWASP Top Ten. Retrieved November 19, 2022, from <https://www.owasptopten.org/>

OWASP. (2021d, October). *The OWASP Application Security Verification Standard (ASVS) 4.0.3*. GitHub. Retrieved July 25, 2023, from <https://github.com/OWASP/ASVS/raw/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf>

OWASP. (n.d.-a). *About the OWASP Foundation*. OWASP Foundation.

Retrieved November 18, 2022, from <https://owasp.org/about/>

OWASP. (n.d.-d). *Cross Site Request Forgery (CSRF)*. OWASP Foundation.

Retrieved February 9, 2023, from <https://owasp.org/www-community/attacks/csrf>

OWASP. (n.d.-f). *Cross-Site Request Forgery Prevention*. OWASP Cheat Sheet Series. Retrieved July 11, 2023, from

[https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html)

OWASP. (n.d.-n). *Cross Site Scripting Prevention*. OWASP Cheat Sheet Series. Retrieved December 22, 2023, from

[https://cheatsheetseries.owasp.org/cheatsheets/Cross\\_Site\\_Scripting\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html)

OWASP. (n.d.-l). *Cross Site Scripting (XSS)*. OWASP Foundation. Retrieved December 7, 2023, from <https://owasp.org/www-community/attacks/xss/>

OWASP. (n.d.-j). *Database Security*. OWASP Cheat Sheet Series. Retrieved December 5, 2023, from

[https://cheatsheetseries.owasp.org/cheatsheets/Database\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Database_Security_Cheat_Sheet.html)

OWASP. (n.d.-h). *OWASP Dependency-Check*. OWASP Foundation.

Retrieved July 29, 2023, from <https://owasp.org/www-project-dependency-check/>

OWASP. (n.d.-b). *OWASP Top Ten*. OWASP Foundation. Retrieved

November 19, 2022, from <https://owasp.org/www-project-top-ten/>

OWASP. (n.d.-i). *Server Side Request Forgery Prevention*. OWASP Cheat Sheet Series. Retrieved November 9, 2023, from

[https://cheatsheetseries.owasp.org/cheatsheets/Server\\_Side\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html)

OWASP. (n.d.-k). *SQL Injection Prevention*. OWASP Cheat Sheet Series.

Retrieved December 6, 2023, from

[https://cheatsheetseries.owasp.org/cheatsheets/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html)

OWASP. (n.d.-m). *Types of XSS*. OWASP Foundation. Retrieved December

22, 2023, from [https://owasp.org/www-community/Types\\_of\\_Cross-Site\\_Scripting](https://owasp.org/www-community/Types_of_Cross-Site_Scripting)

OWASP. (n.d.-e). *WSTG - Stable*. WSTG - Stable | OWASP Foundation.

Retrieved July 4, 2023, from <https://owasp.org/www-project-web-security-testing-guide/stable/0-Foreword/README>

OWASP. (n.d.-g). *WSTG - Stable*. WSTG - Stable | OWASP Foundation. Retrieved July 26, 2023, from [https://owasp.org/www-project-web-security-testing-guide/stable/4-Web\\_Application\\_Security\\_Testing/08-Testing\\_for\\_Error\\_Handling/01-Testing\\_For\\_Improper\\_Error\\_Handling](https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/08-Testing_for_Error_Handling/01-Testing_For_Improper_Error_Handling)

OWASP. (n.d.-c). *WSTG - v4.2*. WSTG - v4.2 | OWASP Foundation. Retrieved November 29, 2022, from [https://owasp.org/www-project-web-security-testing-guide/v42/4-Web\\_Application\\_Security\\_Testing/05-Authorization\\_Testing/02-Testing\\_for\\_Bypassing\\_Authorization\\_Schema](https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/05-Authorization_Testing/02-Testing_for_Bypassing_Authorization_Schema)

PortSwigger. (n.d.). *What is SSRF (Server-side request forgery)? Tutorial & Examples | Web Security Academy*. PortSwigger. Retrieved November 5, 2023, from <https://portswigger.net/web-security/ssrf>

Schreiber, T. (2004, December). *Session Riding - A Widespread Vulnerability in Today's Web Applications*. SecureNet GmbH. Retrieved July 11, 2023, from [https://crypto.stanford.edu/cs155old/cs155-spring08/papers/Session\\_Riding.pdf](https://crypto.stanford.edu/cs155old/cs155-spring08/papers/Session_Riding.pdf)

Shar, L. K., & Tan, H. B. K. (2013, March). Defeating SQL Injection. *Computer*, 46(2013.13), 69-77. 10.1109/MC.2012.283

Sonatype. (2023). Sonatype | State of the Software Supply Chain - 8th annual report. Retrieved August 4, 2023, from <https://www.sonatype.com/hubfs/8th%20Annual%20SSCR%20-%202023.pdf>

Spring. (n.d.). *Method Security :: Spring Security*. Spring Boot. Retrieved December 28, 2022, from <https://docs.spring.io/spring-security/reference/servlet/authorization/method-security.html>

Spring. (n.d.-b). *JdbcTemplate (Spring Framework 6.0.10 API)*. Retrieved July 3, 2023, from <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.jdbc.core/JdbcTemplate.html>

Spring. (n.d.-f). *Spring Boot | Logging*. Retrieved August 3, 2023, from <https://docs.spring.io/spring-boot/docs/3.1.2/reference/html/features.html#features.logging>

Spring. (n.d.-e). *Authentication*. Spring. Retrieved July 30, 2023, from <https://docs.spring.io/spring-security/reference/servlet/authentication/index.html#servlet-authentication>

Spring. (n.d.-d). *Spring Boot*. Spring Boot. Retrieved July 4, 2023, from <https://spring.io/projects/spring-boot>

Spring. (n.d.-g). *Spring Data JDBC and R2DBC*. Spring. Retrieved December 6, 2023, from <https://docs.spring.io/spring-data/relational/reference/index.html>

Spring. (n.d.-c). *Spring Framework*. Spring Boot. Retrieved July 4, 2023, from <https://spring.io/projects/spring-framework>

Stuttard, D., & Pinto, M. (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley.

Symantec. (2008, January 22). *Drive-by Pharming in the Wild*. Retrieved February 8, 2019, from <http://www.symantec.com/connect/blogs/drive-pharming-wild>

Vaadin. (2023, March 28). *Enabling Security*. Vaadin. Retrieved May 1, 2023, from <https://vaadin.com/docs/latest/security/enabling-security>

Vaadin. (2023b, April 19). *Starting a Project | Get Started*. Vaadin. Retrieved July 25, 2023, from <https://vaadin.com/docs/latest/guide/start>

Vaadin. (n.d.-e). *Common Vulnerabilities | Advanced Security Topics | Security*. Vaadin. Retrieved December 22, 2023, from <https://vaadin.com/docs/latest/security/advanced-topics/vulnerabilities>

Vaadin. (n.d.-c). *Production Build | Deploying to Production*. Vaadin. Retrieved July 26, 2023, from <https://vaadin.com/docs/latest/production/production-build>

Vaadin. (n.d.-d). *Remote Procedure Calls | Element API | Creating UI*.

Vaadin. Retrieved October 22, 2023, from

<https://vaadin.com/docs/latest/create-ui/element-api/client-server-rpc>

Vaadin. (n.d.-b). *Security Architecture | Advances Security Topics*. Vaadin.

Retrieved July 3, 2023, from

<https://vaadin.com/docs/latest/security/advanced-topics/architecture>

Vaadin. (n.d.-a). *Vaadin Flow | The Full-Stack Java Web Framework*. Vaadin.

Retrieved July 3, 2023, from <https://vaadin.com/flow>

Zetter, K. (2010, March 26). Hacker Sentenced to 20 Years for Breach of Credit Card Processor. *WIRED*. <https://www.wired.com/2010/03/heartland-sentencing/>



## 8 SVENSKT SAMMANDRAG

### En evaluering om hur webbramverk kan hjälpa utvecklare att bygga mer säkra webbapplikationer

Allt fler applikationer utvecklas i dagens läge för webben. Även sådana applikationer som traditionellt har betraktats som desktop-programvara har omvandlats till webbapplikationer. Webbapplikationer används för många aspekter av vårt dagliga liv, vi betalar fakturor på nätbanken, beställer mat från butiken eller diskuterar med läkaren över en chat. Även om dessa applikationer är gjorda för att underlätta våra liv, har de också en mörk sida: de har blivit lukrativa mål för hackare.

Open Web Application Security Project (OWASP) är världens största icke-vinstdrivande organisation som jobbar med it-säkerhet av webbapplikationer. Deras kanske mest kända projekt är OWASP Top Ten, som sammansätter till ett dokument vilka sårbarheter som anses vara de tio mest kritiska kategorierna av sårbarheter som finns i webbapplikationer. Två andra välkända projekt är Web Security Testing Guide (WSTG) och Application Security Verification Standard (ASVS). WSTG är ett dokument som beskriver *hur* man skall säkerhetstesta en webbapplikation, medan ASVS beskriver *vad* man skall testa.

Avsiktet med detta diplomarbete är att utvärdera på vilka sätt webbramverk kan hjälpa utvecklare att bygga mer säkra webbapplikationer. För att evaluera de valda ramverken används en kursregistreringsapplikation vars funktionella och ickefunktionella krav från ett säkerhetsperspektiv motsvarar en typisk modern webbapplikation. Applikationen säkerhetstestas för de sårbarheter enumererat i OWASP Top Ten med hjälp av principerna och metoderna som beskrivs i WSTG och ASVS. Sårbarheterna kategoriseras i fyra olika grupper på basen av hur ramverken stöder utvecklarna att mitigera sårbarheterna. Kategorierna är:

1. **Åtgärdas direkt av ramverket.** Dessa sårbarheter mitigeras av ramverket på ett sådant sätt att det inte kräver någon aktiv interaktion från programutvecklaren.
2. **Åtgärdas av ramverket genom konfigurationer.** Sårbarheter i denna kategori kan mitigeras av de ramverk som används, men inte genom standardkonfigurationer. Att använda standardkonfigurationen skulle göra applikationen sårbar.
3. **Ramverket innehåller hjälpverktyg.** Sårbarheterna mitigeras inte direkt av ramverken, men ramverken förser utvecklarna med hjälpmedel som vägleder eller underlättar utvecklarna att implementera säkra lösningar.

4. **Inga hjälpmedel för att mitigera sårbarheten.** Sårbarheterna måste åtgärdas av utvecklarna genom korrekta beslut om design och implementering.

Exempelapplikationen är byggd användande av en trenivå-arkitektur där presentationslagret (användargränssnittet), logiklagret samt lagret med tillgång till databasen är separerade från varandra. Användargränssnittet implementerades med hjälp av Vaadin Flow-ramverket. Med hjälp av Vaadin Flow, kan utvecklarna bygga webbapplikationer användande endast av java-programmeringsspråket, vilket betyder att utvecklarna inte har behov att lära sig HTML eller JavaScript, något som annars är typiskt för webbutveckling. Logik- samt datalagret implementerades med hjälp av Spring Boot.

En OWASP Top Ten kategori kan vara mer konceptuell med avsikt att öka medvetenheten om säkerhetsorienterat tänkande (t.ex. kategorin "Säker design") medan en annan kategori kan vara en samling av flera specifika sårbarheter, t.ex. kategorin "Broken Access Control". Den senare omfattar inte bara bredare begrepp, utan även specifika sårbarheter som indirekta objektreferenser och cross-site request forgeries. En OWASP Top Ten kategori kunde således kategoriseras i evaluering i fler än en av de valda grupperna, beroende på om kategorin innehöll mer specifika sårbarheter eller ifall de handlade om bredare begrepp.

Analysen hanterade sammanlagt 15 olika specifika sårbarheter eller sårbarhetsgrupper. Det första man kunde se av analysen var att fyra av sårbarhetskategorierna var evaluerade som "ej tillämpligt". Detta betyder att dessa sårbarheter ligger inte i koden av programmet, utan, till exempel i själva tekniska designet av applikationen - en osäker design kan inte göras säkert genom en bra implementation. På grund av att grundproblemet av sårbarheterna ligger utanför koden, kan de inte därför heller hanteras av ramverken. Detta betyder att utvecklingsteamet måste ta hänsyn till flera aspekter utöver själva programmerandet, för att kunna skapa en applikation som kan konstateras vara säker.

Endast tre av de evaluerade sårbarheterna kunde mitigeras direkt av ramverken: "Insecure Direct Object References", "Cross-Site Request Forgery" samt "Cross-Site Scripting". Dessa är tekniska sårbarheter med specifika tekniska lösningar och därför kan de hanteras direkt av ramverken.

Fyra sårbarheter eller sårbarhetskategorier grupperades under "ramverket innehåller hjälpverktyg": bristfällig åtkomstkontroll, SQL-inketioner, sessionhantering samt integritet av data. Vissa av dessa sårbarheter kan inte av tekniska skäl hanteras av ramverket. Som ett bra exempel är bristfällig åtkomstkontroll. Sårbarheten handlar om att försäkra, att användare inte kan komma åt funktionalitet, data eller andra resurser som de inte har rättigheter till. Själva begränsandet av rättigheterna kan hanteras av ramverken, men problemet ligger i att ett ramverk inte kan veta vilka resurser en användare bör

ha rättigheter till, eftersom det alltid är beroende på ett icke-tekniskt kontext. SQL-injektioner är ett annat bra exempel. Spring Boot erbjuder bra verktyg för att mitigera sårbarheten, men ramverket kan inte *tvinga* användaren att använda verktygen på ett rätt sätt. Om utvecklaren inte förstår hur SQL-injektionssårbarheterna uppstår är det möjligt att hen använder ramverkets verktyg på fel sätt åt på det sättet introducerar en sårbarhet i själva applikationen.

Två sårbarheter kunde mitigeras med hjälp av konfigurationer. Standardkonfigurationerna är ofta avsedda för att göra det enkelt för utvecklare att ta i bruk nya verktyg och utveckla med dem, men dessa konfigurationer är inte lämpliga för produktionsanvändning.

Inga mitigationsstrategier eller verktyg erbjöds av *de valda ramverken* till tre sårbarhetskategorier. Server-Side Request Forgery är ett tekniskt problem som kräver en gemensam lösning mellan programvaran och nätverket, och därför kan inte lösas direkt av ett ramverk. Kryptografiska fel och autentisering av användare handlar mycket om vad man anser att är ett ramverks ansvar. Vaadin har valt att lösa ett problem omkring utveckling av användargränssnitt, medan Spring Boot löser problem i logiklagret. Varkendera av dessa ramverk har valt att ha autentisering inom det problemdomän de vill erbjuda lösningar till - till sist och slut handlar det om att ramverksutvecklarna måste bestämma hur de spendera de begränsade resurser de har till förfogande. Det finns ramverk som löser problemet kring autentisering av användare, men ett sådant ramverk var inte med i evalueringen.

Det finns ingen patentlösning för säkerhet. För att bygga säker programvara måste utvecklarna förstå hur de verktyg de använder fungerar, vad som är ramverkets ansvar och vad som är utvecklarens ansvar. Utvecklarna måste förstå vilken typ av sårbarheter som deras applikationer kan utsättas för och vilka risker dessa potentiella sårbarheter medför. För att bygga en applikation som kan anses vara säker, krävs det samarbete med nätverks-, system- och säkerhetsingenjörer, god DevSecOps-praxis och en lämplig risktolerans bör överenskommas med produktägaren. När allt kommer omkring är den säkerhetsnivå vi implementerar direkt kopplad till de risker vi är villiga att ta - ett hobbyprojekt behöver inte samma säkerhetsnivå som ett sjukvårdssystem som ansvarar för livsviktiga funktioner.

## 9 APPENDIX A

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
bcprov-jdk18on-1.71.jar	cpe:2.3:a:bouncycastle:bouncy-castle-crypto-package:1.71:****:* cpe:2.3:a:bouncycastle:bouncy_castle_crypto_package:1.71:****:* cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle:1.71:****:* cpe:2.3:a:bouncycastle:legion-of-the-bouncy-castle-java-cryptography-api:1.71:****:* cpe:2.3:a:bouncycastle:the_bouncy_castle_crypto_package_for_java:1.71:****:* *** ..*	<a href="#">pkg:maven/org.bouncycastle/bcprov-jdk18on@1.71</a>	MEDIUM	1	Low	60
commons-fileupload-1.4.jar	<a href="#">cpe:2.3:a:apache:commons_fileupload:1.4:****:*</a>	<a href="#">pkg:maven/commons-fileupload/commons-fileupload@1.4</a>	HIGH	1	Highest	115
ejs:3.1.8	<a href="#">cpe:2.3:a:ejs:ejs:3.1.8:****:*</a>	<a href="#">pkg:npm/ejs@3.1.8</a>	CRITICAL	1	Highest	8
h2-2.1.214.jar	<a href="#">cpe:2.3:a:h2database:h2:2.1.214:****:*</a> ..*	<a href="#">pkg:maven/com.h2database/h2@2.1.214</a>	HIGH	2	Highest	44
jackson-databind-2.13.4.2.jar	<a href="#">cpe:2.3:a:fasterxml:jackson-databind:2.13.4.2:****:*</a> cpe:2.3:a:fasterxml:jackson-modules-	<a href="#">pkg:maven/com.fasterxml.jackson.core/jackson-databind@2.13.4.2</a>	MEDIUM	1	Highest	44

```
java8:2.13.4.2:*.~*.~*.~*.~*.~*
```

json5:2.2.1	<a href="#">cpe:2.3:a:json5:json5:2.2.1:*:*:*:*:*</a>	<a href="#">pkg:npm/json5@2.2.1</a>	HIGH	2	Highest	8
maven-core-3.0.jar	<a href="#">cpe:2.3:a:apache:maven:3.0:*:*:*:*:*</a>	<a href="#">pkg:maven/org.apache.maven/maven-core@3.0</a>	CRITICAL	1	Highest	23
maven-settings-3.0.jar		<a href="#">pkg:maven/org.apache.maven/maven-settings@3.0</a>	CRITICAL	1		25
maven-shared-utils-3.1.0.jar	<a href="#">cpe:2.3:a:apache:maven_shared_utils:3.1.0:*:*:*:*:*</a> <a href="#">cpe:2.3:a:utils_project:utils:3.1.0:*:*:*:*:*</a>	<a href="#">pkg:maven/org.apache.maven.shared/maven-shared-utils@3.1.0</a>	CRITICAL	1	Highest	30
nimbus-jose-jwt-9.23.jar (shaded: net.minidev:json-smart:2.4.8)	<a href="#">cpe:2.3:a:json-smart_project:json-smart:2.4.8:*:*:*:*:*</a> cpe:2.3:a:json-smart_project:json-smart-v2:2.4.8:*:*:*:*:*	<a href="#">pkg:maven/net.minidev/json-smart@2.4.8</a>	HIGH	1	High	31
semver:6.3.0		<a href="#">pkg:npm/semver@6.3.0</a>	HIGH	2		5
semver:7.0.0		<a href="#">pkg:npm/semver@7.0.0</a>	HIGH	1		5
semver:7.3.7		<a href="#">pkg:npm/semver@7.3.7</a>	HIGH	2		6
snakeyaml-1.30.jar	<a href="#">cpe:2.3:a:snakeyaml_project:snakeyaml:1.30:*:*:*:*:*</a>	<a href="#">pkg:maven/org.yaml/snakeyaml@1.30</a>	CRITICAL	7	Highest	44

spring-boot-2.7.5.jar	<a href="#">cpe:2.3:a:vmware:spring_boot:2.7.5:*:*:*:*:*:*</a>	<a href="#">pkg:maven/org.springframework.boot/spring-boot@2.7.5</a>	CRITICAL	2	Highest	38
spring-boot-devtools-2.7.5.jar	<a href="#">cpe:2.3:a:vmware:spring_boot:2.7.5:*:*:*:*:*:</a> <a href="#">cpe:2.3:a:vmware:spring_boot_tools:2.7.5:*:*:*:*:*:</a> <a href="#">cpe:2.3:a:vmware:spring_tools:2.7.5:*:*:*:*:*:</a>	<a href="#">pkg:maven/org.springframework.boot/spring-boot-devtools@2.7.5</a>	CRITICAL	2	Highest	40
spring-boot-starter-web-2.7.5.jar	<a href="#">cpe:2.3:a:vmware:spring_boot:2.7.5:*:*:*:*:*:</a> <a href="#">cpe:2.3:a:web_project:web:2.7.5:*:*:*:*:*:</a>	<a href="#">pkg:maven/org.springframework.boot/spring-boot-starter-web@2.7.5</a>	CRITICAL	2	Highest	36
spring-core-5.3.23.jar	<a href="#">cpe:2.3:a:pivotal_software:spring_framework:5.3.23:*:*:*:*:*:</a> <a href="#">cpe:2.3:a:springsource:spring_framework:5.3.23:*:*:*:*:*:</a> <a href="#">cpe:2.3:a:vmware:spring_framework:5.3.23:*:*:*:*:*:</a>	<a href="#">pkg:maven/org.springframework/spring-core@5.3.23</a>	HIGH	3	Highest	37
spring-security-core-5.7.4.jar	<a href="#">cpe:2.3:a:pivotal_software:spring_security:5.7.4:*:*:*:*:*:</a> <a href="#">cpe:2.3:a:vmware:spring_security:5.7.4:*:*:*:*:*:</a>	<a href="#">pkg:maven/org.springframework.security/spring-security-core@5.7.4</a>	CRITICAL	4	Highest	38
spring-security-crypto-5.7.4.jar	<a href="#">cpe:2.3:a:pivotal_software:spring_security:5.7.4:*:*:*:*:*:</a> <a href="#">cpe:2.3:a:vmware:spring_security:5.7.4:*:*:*:*:*:</a>	<a href="#">pkg:maven/org.springframework.security/spring-security-crypto@5.7.4</a>	CRITICAL	5	Highest	38



vite:3.1.0	<a href="#">cpe:2.3:a:vitejs:vite:3.1.0:*:*:*:*:*</a>	<a href="#">pkg:npm/vite@3.1.0</a>	HIGH	2	Highest	8
------------	---	------------------------------------	------	---	---------	---