software

MDPI

Article

Automating Structured Query Language Injection and Cross-Site Scripting Vulnerability Remediation in Code

Kedar Sambhus † **and Yi Liu** *

Department of Computer and Information Science, University of Massachusetts Dartmouth, 285 Old Westport Road, Dartmouth, MA 02747, USA; kedarsambhus@outlook.com

* Correspondence: yliu11@umassd.edu

† Current address: Amazon, Bellevue, WA 98004, USA.

Abstract: Internet-based distributed systems dominate contemporary software applications. To enable these applications to operate securely, software developers must mitigate the threats posed by malicious actors. For instance, the developers must identify vulnerabilities in the software and eliminate them. However, to do so manually is a costly and time-consuming process. To reduce these costs, we designed and implemented Code Auto-Remediation for Enhanced Security (CARES), a web application that automatically identifies and remediates the two most common types of vulnerabilities in Java-based web applications: SQL injection (SQLi) and Cross-Site Scripting (XSS). As is shown by a case study presented in this paper, CARES mitigates these vulnerabilities by refactoring the Java code using the Intercepting Filter design pattern. The flexible, microservice-based CARES design can be readily extended to support other injection vulnerabilities, remediation design patterns, and programming languages.

Keywords: vulnerability mitigation; secure design pattern; automated code refactoring

Citation: Sambhus, K.; Liu, Y. Automating Structured Query Language Injection and Cross-Site Scripting Vulnerability Remediation in Code. *Software* **2024**, *3*, 28–46. https://doi.org/10.3390/software3010002

Academic Editor: Paolino Di Felice

Received: 19 November 2023
Revised: 1 January 2024
Accepted: 9 January 2024
Published: 12 January 2024

1. Introduction

A security vulnerability is a weakness in system security procedures, architectures, designs, implementations, or internal controls that attackers can exploit to serve their ill intentions. With rising advancement in software development technology, the risk of security threats is also rising rapidly. The National Vulnerability Database (https://nvd.nist.gov/ (accessed on 1 January 2024)) holds over 23,800 vulnerabilities published alone in 2022. This is a higher number than in previous years (21,822 in 2021 and 20,248 in 2020). Edgescan, a cybersecurity company that specializes in providing vulnerability management and application security services, publishes an annual vulnerability statistics report. According to its year 2023 report [1], about 33% of the internet-facing web application vulnerabilities are high or critical risk. Such a high number of risks requires a considerable amount of time to remediate. With the current techniques, as indicated by Edgescan, the "Mean Time to Remediation (MTTR) (the calendar days it takes to fix vulnerabilities across the Full Stack) for Critical Severity vulnerabilities on the web application/API layer is 73.9 days". Veracode, a cybersecurity company that specializes in providing automated application security testing and software security solutions, summarizes the results from scanning their customers' applications, and in its annual report of 2023, it found that over 74% of applications had at least one security vulnerability found in the last scan over the last 12 months. These include over 69% that have at least one OWASP Top 10 flaw, and over 56% have at least one CWE Top 25 flaw [2].

Many of the vulnerabilities found in Edgescan's 2021 report were more than three years old [3]. In 2020, 65% of the attacks utilized vulnerabilities that were at least 3 years old, and 32% of them were approximately five years old, dating back to 2015. The oldest vulnerability discovered is 21 years old and remains unpatched by the company [3,4]. This highlights the lack of investment in the time and effort required to secure applications.

Security is an essential nonfunctional requirement of a successful software system, which should be addressed during all phases of software development. Unfortunately, it is seen that security is implemented vastly during or after the development phase of the software development life cycle. The cost of fixing vulnerabilities after their first appearance is much higher than addressing them in earlier software development life cycle stages.

There are a few mature vulnerability scanners [5,6] that can scan web applications for security vulnerabilities. After vulnerabilities are identified, the development team needs to remediate them. However, removing vulnerabilities manually can be expensive and time-consuming [7], and managing vulnerabilities and deploying patches can be challenging [8].

This research aims to design and develop CARES (Code Auto-Remediation for Enhanced Security), an application that automatically identifies, allocates, and mitigates two of the most common web vulnerabilities, SQL injection (SQLi) and Cross-Site Scripting (XSS), in pre-existing web applications developed in Java.

Our research focuses primarily on SQLi and XSS vulnerabilities, given their substantial prevalence and potential for exploitation. The Edgescan 2023 report [1] reveals that injection vulnerabilities, including SQL injection, code injection, and more, continue to dominate the realm of application layer and API vulnerabilities. SQLi stands out as the most critical vulnerability, accounting for 23.4% of web application vulnerabilities. XSS, ranking third at 19.1%, also poses a significant risk. These statistics underscore the importance of addressing SQLi and XSS vulnerabilities as they persist as key attack vectors, despite the availability of effective mitigation measures and detection techniques.

For the first version of CARES, we focus on applications developed in the Java programming language. According to Veracode's annual report, based on the analysis of 760,000 applications [2], Java commands a significant share of applications scanned by Veracode clients (44%). However, it exhibits vulnerability remediation challenges when compared to .NET and JavaScript. Java applications tend to lag behind in reducing technical debt and improving security, with roughly 56% of them experiencing increased security debt. In addition, Java applications take notably longer to address vulnerabilities, requiring 243 days to close 50% of flaws, as opposed to the quicker response observed in .NET and JavaScript [2]. These data support our decision to focus on Java application in our research.

The main contributions of this study are as follows:

1. *Automated vulnerability mitigation:* This study develops CARES, a novel code refactoring application, to automate the mitigation of SQL injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities in Java web applications. It simplifies a traditionally manual and time-consuming security task, improving the efficiency of handling prevalent web vulnerabilities.
2. *Secure design integration:* This study emphasizes the integration of secure design patterns into CARES, going beyond simple code fixes to embed best practices directly into the application. This approach enhances the overall system security by addressing the security at the design level.
3. *Extensibility and flexibility:* Focusing on extensibility and flexibility, CARES is designed with a microservices architecture. This design allows CARES to adapt to changes, such as diverse programming languages, additional vulnerability types, and more vulnerability scanners.

The rest of this paper is organized as follows. Section 2 provides an overview of two common vulnerabilities, SQLi and XSS, and introduces secure design patterns and the microservices architecture. Section 3 details the system design that uses microservices and incorporates a chosen secure design pattern. Section 4 evaluates the effectiveness of the developed refactoring tool in mitigating SQLi and XSS vulnerabilities using the Tolven web application as a case study. Section 6 discusses the findings and potential areas of further investigation. Finally, Section 7 summarizes the work and outlines possible extensions to this study.

2. Background

This section provides a brief introduction to two software vulnerabilities: *SQL injection* and *Cross-Site Scripting*, overviews the secure design patterns and microservices architecture, and discusses related work.

2.1. SQL Injection (SQLi)

Identified in Common Weakness Enumeration as CWE-89, "Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')" [9], SQL injection (SQLi) is a web security vulnerability type that allows an attacker to interfere with the queries that an application makes to its database.

SQLi involves the insertion or "injection" of an SQL query via the input data from the client to the application. A successful SQLi attack enables the attacker to read sensitive data from the database, modify its data, or execute administration operations. The code in Listing 1 contains an SQLi vulnerability that reads in the request parameters collecting the credentials (*username* and *password*) from users' input fields (line 1–2) and uses them to directly construct a final query (line 3).

Listing 1. An SQLi vulnerability example.

```
1  String userName = RequestParameter("username");
2  String pwd = RequestParameter("password");
3  String sql = 'SELECT * FROM Users WHERE Name ="' + userName + '"
4      AND Pass ="' + pwd + '";
```

An attack can occur when a client enters `'1' OR '1' ='1'` in both the *username* and *password* input fields. The server will form the query as `SELECT * FROM Users WHERE Name ='1' OR '1' ='1' AND Pass= '1' OR '1' ='1'`, which is sent to the database for execution. Since the `'1' ='1'` condition always holds, the database returns a nonempty result, granting access to the attacker.

The code segment in Listing 2 demonstrates a real-world SQLi vulnerability found in class `SnapshotBean` in the package `org.tolven.analysis.ejb` of the Tolven Health Record application [10]. In this code snippet, an SQL query is dynamically created in the `createQuery()` method by appending a String variable `cohortType` to a prebuilt query (line 420–422).

Listing 2. An SQLi vulnerability in Tolven.

```
418  @override
419  public void deleteCohortPlaceholder(Account account, String cohortType){
420      Query query = em.createQuery("delete PlaceholderID pi where "
421          + "pi.account.id = :account and pi.extension LIKE '"
422          + cohortType + "%'");
423      query.setParameter("account", account.getId());
424      query.executeUpdate();
425  }
```

There is no validation in place to restrict the possible contents of the variable `cohortType`. Attackers can exploit this SQL query by sending a malicious request, for example, by setting the `cohortType` to `' OR 1=1 - -`. This manipulation leads to the construction of the following query.

```
delete PkaceholderID pi
Where pi.account.id = :account and pi.extension LIKE '' OR 1=1 --%'
```

`OR 1=1` makes the condition always true, and the double-dash sequence `- -` serves as a comment indicator in SQL, treating the remainder of the query as a comment. Thus, the condition in the `where` clause is evaluated as true, and all the records in the table `PlaceholderID` will be removed.

The cause of SQLi vulnerabilities is the lack of validation and sanitization for the data from outside sources. Prevention strategies include input validation, data sanitization, and

technology-specific or configuration solutions, such as prepared statements [11], stored procedures [12], and more. Blacklisting and whitelisting are two input validation strategies to prevent SQLi vulnerabilities. Blacklisting involves rejecting external input containing malicious keywords (e.g., `select`, `delete`, `insert`, `union`, `'`, `- -`, etc.). Whitelisting only allows requests with "good" inputs, such as alphanumeric characters, and reject others. Data sanitization involves escaping all external inputs to ensure that external data are treated as data rather than part of the SQL query construct by the underlying database. For example, we can escape the input " ' " to " \' " to neutralize the effect of the apostrophe (') during the SQL query formation.

2.2. Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS), identified as CWE-79, "Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')" [13], is one of the most commonly recurring security vulnerabilities in web applications. XSS allows an attacker to compromise the interactions that users have with a vulnerable application [14]. This is a type of injection in which an attacker uses a web application to send malicious code in the form of browser-side script to a different end user. The end user's browser has no way to know that it is a malicious script and will execute it because it thinks the script came from a trusted source. This opens up a vast amount of information to the attacker, such as cookies, session tokens, or other sensitive information retained by the browser. Liu et al. [14] surveyed and classified the XSS attacks into *DOM-based XSS attack*, *stored XSS attack*, and *reflected XSS attack*.

The code in Listing 3 shows an example of a typical DOM-based XSS attack. When a developer constructs a web page and uses the client input without sanitization to write to the DOM, attackers can exploit it to conduct an XSS attack using malicious input. In the example, the attacker sends a malicious script (`<script>alert(document.cookie)</script>`) embedded in the URL to a victim. Upon the victim's click, a request is sent to the server. Then, the victim's browser renders the response (with the JavaScript malicious code) from the server and executes it, generating an alert pop-up with a cookie. Attackers can use similar techniques to steal sensitive information.

Listing 3. Simple DOM-based XSS attack.

```
<script type="text/javascript">
    var val= '../test.php?cookie='+escape(document.cookie));
</script>
```

Listing 4 presents an XSS vulnerability found in the file `GenericServlet.java` under package `org.tolven.component.tolvenweb/src/org/tolven/ajax` in the Tolven application. In this case, the application directly obtains String data from the client request (line 165–167), which is then used as is (in lines 181 to 183) without proper validation, allowing malicious code to be injected and executed as JavaScript.

Listing 4. An XSS vulnerability in Tolven.

```
165  String path = req.getParameter( "element" );
166  String rolename = req.getParameter("role");
167  String source = req.getParameter("source");
168  // get menu structure
169  AccountMenuStructure ams = menuBean.findAccountMenuStructure(
170      activeAccountUser.getAccount().getId(),path);
171  MenuStructure ms = menuBean.findMenuStructure(activeAccountUser, ams);
172  String defPathSuffix = ms.getDefaultPathSuffix();
173  // get its preferred children
174  List<MenuStructure> children =
175      menuBean.findSortedChildren(activeAccountUser, ams);
176  TolvenResourceBundle tolvenResourceBundle =
177      TolvenRequest.getInstance().getResourceBundle();
178  String title = tolvenResourceBundle.getString("UserPreferencesTitle");
179  // prepare xml
```

```
180  writer.write("<ajax-response>");
181  writer.write("<response path=\"" + path + "\" role=\""
182          + rolename +"\" defpath=\"" + defPathSuffix + "\" "
183          + " title=\"" + title + "\">" );
```

An attacker can pass the following malicious JavaScript to one of the String variables, such as *rolename*, and the malicious code will be executed in the victim's browser.

```
<script>/* Bad stuff here... */</script>
```

Similar to the cause of SQLi vulnerabilities, XSS vulnerabilities are rooted in the lack of proper validation and sanitization of input data. Prevention strategies include input validation, data santization, and other technology-specific solutions such as implementing a *Content Security Policy* [15], using the *HTTPOnly cookie* flag [16], *SameSite cookie parameter* [17], and more. Input validation can be either blacklist- or whitelist-based. Blacklist validation disallows web requests from processing malicious content, such as `<SCRIPT>`, `HTML` tags, `JavaScript` tags, etc., while whitelist validation only allows requests with harmless characters. Data sanitization for preventing XSS is achieved by encoding or escaping the input data to ensure that inputs are treated as data rather than markup for the browser to process. For example, `<script>` is converted to `&lt;script&gt;` after sanitization using encoding. While displayed, the browser will output the original input string to the user but will not execute it.

2.3. Secure Design Patterns

A pattern is a general reusable solution to a commonly occurring design problem. Secure design patterns are meant to eliminate the accidental insertion of vulnerabilities into code [18]. A secure design pattern is a well-proven, reusable solution to a recurring security problem within specific contexts. They address security concerns at different levels of specificity, ranging from high-level architectural patterns that influence the overall system design to implementation-level patterns that provide guidance on implementing specific functionalities within the system [19].

Ratnaparkhi et al. [20] developed a methodology for selecting appropriate secure design patterns to mitigate software vulnerabilities. Their study proposed a collection of secure patterns across architecture, design, and implementation levels to address XSS and SQLi vulnerabilities. At the design level, they recommended patterns such as the *Intercepting Filter*, *Secure Chain of Responsibility*, and *Secure Strategy Factory*. For our study, we have chosen to apply the *Intercepting Filter* to mitigate XSS and SQLi vulnerabilities.

The *Intercepting Filter* pattern is a presentation-tier pattern for the Java 2 platform, Enterprise Edition (J2EE) [21]. This design pattern is used to facilitate preprocessing or postprocessing of requests or responses in an application. It is particularly useful when there is a need to filter and manipulate incoming requests before they reach the core functionality of an application or to process responses before they are sent back to the client. Figure 1 illustrates the structure of this pattern, and the participants in the pattern are described as follows.

- **Client**: The client initiates the request to the target component by sending it to FilterManager.
- **Filter**: This is an interface that defines the core execute method, which is implemented by Concrete Filters.
- **Concrete filters**: Concrete filters are classes that implement the filter interface. They are responsible for performing the actual filtering of requests and responses.
- **Filter chain**: A filter chain is an ordered collection of independent filters. Each filter in the chain is responsible for handling a specific aspect of request or response processing. The filters are executed sequentially.
- **Filter manager**: The filter manager class manages the filter processing flow. It creates the filter chain with the appropriate filters, ensuring they are in the correct order and and initiating the processing of requests or responses.

- **Target**: The target represents the resource or functionality that is requested by the client. It is the endpoint that the processed request is forwarded to after passing through the filters.

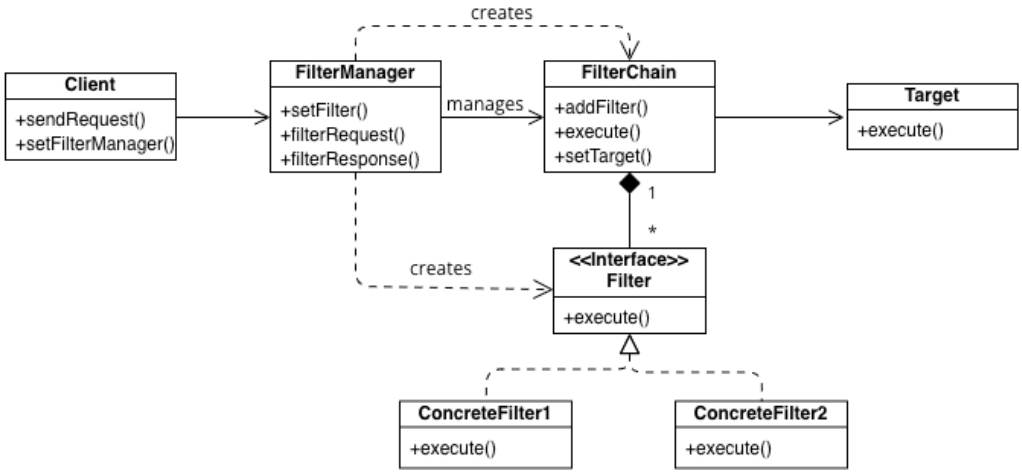

Figure 1. Intercepting Filter pattern.

The `Client` sends a web request to the server and the request invokes the `FilterManager`. The `FilterManager` has a `FilterChain` that is made up of individual filters. Once the request from the `Client` arrives, the `FilterManager` delegates the task of preprocessing or postprocessing the request to individual `Filters` defined by the `FilterChain` before sending the request to the real target. Adding or removing filters is made easier by modifying the `FilterChain`. The `Target` is the intended destination of the web request, and a `Filter` interface can be introduced for all individual filters to communicate with the `FilterChain` and `FilterManager`.

2.4. Microservice Architecture

Microservices architecture is a contemporary approach to structuring software applications. As described by Martin Fowler [22], in a microservices setup, a software application is built using small services, and each microservice has its own process, serving a specific purpose and communicating with other services through application programming interfaces (APIs). These services are designed to be easily deployed and scaled, which can lead to quicker development, improved scalability, and better resilience.

A microservices architecture exhibits many favorable characteristics, and the following two have the most significant impact on the design of our study.

Modularity: Microservices are intentionally designed to be small in size and focus on specific business functions. Each microservice encapsulates its own logic and contributes to the overall functionality of the system. This modular approach minimizes interdependencies between modules, promoting low coupling among them. Consequently, the system becomes easier to maintain and more flexible in adapting to future requirements changes.

Simple communication: Microservices architecture emphasizes lightweight communication methods. The simplicity in communication facilitates modifications to individual services without affecting the overall communication of the entire system, ultimately enhancing scalability [23]. Common communication approaches include using HTTP (Hypertext Transfer Protocol) and the RPC (Remote Procedure Call) for request–response communication and employing lightweight messaging. HTTP requests are stateless and typically use text-based formats such as JSON and XML, commonly used in RESTful APIs. RPC messages are in various formats, including binary or text-based, and use protocols like gRPC [24]. While the RPC offers a higher performance than HTTP [25], its setup can be complex, requiring additional protocol buffers or code generation, and it has potential

security considerations [26]. HTTP, however, is widely supported and is more interoperable across different programming languages and platforms compared to the RPC.

2.5. Related Work

Few automated vulnerability remediation tools have been developed. At the point of preparing this paper, we found only one tool with that aim, a commercial product named Barracuda [27]. Barracuda provides a vulnerability remediation service to automatically "create security configurations customized to specific applications and vulnerabilities, eliminating errors in manual configuration" [28]. The approach Barracuda adopts is distinct from ours, which aims at mitigating the vulnerabilities in code. Citing research by the Software Engineering Institute (SEI), the U.S. Department of Homeland Security (DHS) states in its Software Assurance information sheet that "90% of reported security incidents result from exploits against defects in the design or code of software" [29]. If a vulnerability is identified in the code, it is better to patch the code than to address the vulnerability in the configurations.

As indicated in Sections 2.1 and 2.2, various mitigation and prevention strategies can be applied to tackle SQLi and XSS vulnerabilities. Tools can be designed to address a specific type of vulnerability with a strategy tailored exclusively to that vulnerability. For example, a study by Courant et al. [30] suggested a method to prevent SQL injection attacks by automatically generating prepared statements based on user input and implemented it to mitigate the SQLi vulnerabilities in Java applications. In contrast, CARES generalizes a mitigation strategy by identifying the root causes of vulnerabilities and accommodating specificity for addressing each individual vulnerability through realizing the general strategy. This approach facilitates the potential for CARES to expand its support to other injection vulnerabilities, such as the CRLF (Carriage Return Line Feed) Injection (CWE-93), which share similar root causes with SQLi and XSS.

Our study applies the Intercepting Filter pattern to mitigate XSS and SQLi vulnerabilities. Other secure design patterns can be used to tackle these vulnerabilities. Ratnaparkhi et al. [31] suggested applying the Secure Strategy Factory pattern to address XSS and SQLi vulnerabilities with sanitizing inputs. Both applications of the Intercepting Filter and Secure Strategy Factory have demonstrated effective results in mitigating these vulnerabilities.

3. The Design of CARES

CARES's vulnerability mitigation process involves several key functionalities, as shown in Figure 2. The client uploads a target application, which is stored in a GitHub repository. CARES imports the target application, preparing it for vulnerability assessment. Using a language-specific scanner tailored to the programming language of the project, CARES checks the application for SQLi and XSS vulnerabilities. Upon locating these vulnerabilities in the code, CARES proceeds with the remediation process. To address SQLi and XSS, it deploys an Intercepting Filters implementation, integrating filters designed specifically to counteract these vulnerabilities. CARES instantiates the respective filter associated with the vulnerability type and injects the filter object into the code at the identified vulnerability points, thereby mitigating these security risks.

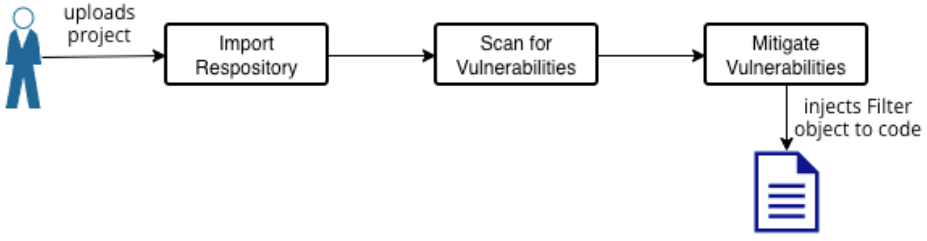

Figure 2. CARES vulnerability remediation process.

3.1. Design Decisions

The system design involved the following key design decisions:

- *Design decision 1* is to integrate an existing open source vulnerability scanner for the detection and allocation of XSS and SQLi vulnerabilities.
  Numerous mature scanners have been developed for identifying web application vulnerabilities. When selecting the most suitable scanner for developing CARES, we set the following criteria: Firstly, the tool should be an open-source static analysis tool so that we can integrate it to the system. Secondly, the scanner must demonstrate proficiency in analyzing Java projects. Thirdly, the tool must effectively identify Cross-Site Scripting (XSS) and SQL injection (SQLi) vulnerabilities in code. Lastly, it should not only report the existence of these vulnerabilities but also provide detailed reports specifying the exact locations, e.g., class names and line numbers, so that the system can fix the vulnerabilities in code. An ideal choice fitting these criteria is the open-source static analysis tool *SpotBugs*, which has the ability of identifying and allocating XSS and SQLi vulnerabilities in Java code, aligning well with the objectives of CARES.
- *Design decision 2* is to introduce secure patterns for mitigating SQLi and XSS vulnerabilities in code.
  The secure patterns are proven solutions to commonly occurring vulnerability challenges. This design choice allows us to move beyond simply fixing the immediate problems in the code to integrate best practices for secure design directly into the application.
- *Design decision 3* is to ensure the flexibility and extensibility of the application.
  We anticipate that CARES will be extensible, accommodating the addition of new features, such as mitigating more vulnerability types and supporting projects coded in languages beyond Java. The application's design should provide the flexibility to address each requirement change independently within specific modules, minimizing potential impacts on other parts of the application.

3.2. Architectural Design

As stated in *design decision 3*, we envision CARES as a flexible and extensible solution. The primary functionalities of the application include importing the target application, scanning it for vulnerabilities, and removing these vulnerabilities from the code. To achieve this, we intend to design the code importer, static code analyzer, and code fixer as loosely coupled components, allowing for independent deployment. While the current version of CARES focuses on Java applications, we plan to extend its support to applications developed in other programming languages, such as PHP, AgularJS, and more. Microservices are an ideal fit for addressing *design decision 3* due to their inherent characteristics. The modular nature of microservices allows us to create independent components, each responsible for a specific task. The simple communication methods used in microservices provide the flexibility to modify individual services without impacting the entire system. Thus, these components can evolve independently, facilitating future enhancements and extensions.

On the server side, we have designed three microservices: the *Repository Importer Service*, the *Vulnerability Checker Service*, and the *Fixer Service*. Each of these services caters to one of the three independent functionalities, including importing a project, analyzing the code, and fixing it. Specifically, the *Repository Importer Service* handles the cloning of the target repository from a remote Git resource and stores it in *File Storage*. The *Vulnerability Checker Service* scans the target application for potential security vulnerabilities, and the *Fixer Service* uses the report generated by the *Vulnerability Service* to locate and refactor the target application to eliminate vulnerabilities. In this version of CARES, we have configured *File Storage* as a file directory on the server. The architecture of CARES is illustrated in Figure 3.

The front-end has been developed as a single-page application, calling all three microservices directly. Clients interact with this single-page application to obtain results.

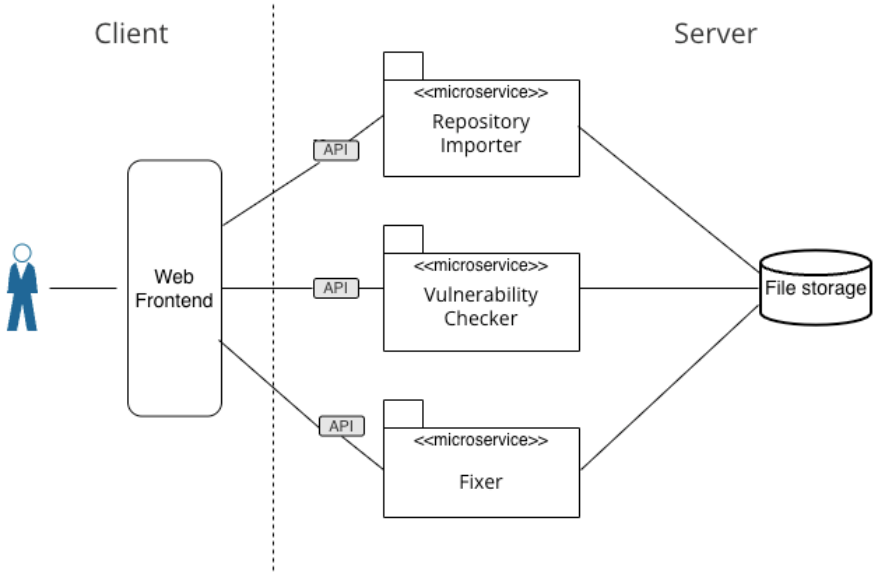

Figure 3. Architecture of CARES.

3.3. Design of Microservices

This section explores the design of the three key microservices, the *Repository Importer Service*, *Vulnerability Checker Service*, and *Fixer Service*, that form the foundation of the backend of CARES. Implementation code snippets of each microservice are provided to highlight the key functionalities. CARES was implemented using the open-source Java Spring Boot framework [32], which provides Java developers with a platform for easily creating auto-configurable, production-grade Spring applications.

3.3.1. Repository Importer Service

The role of the *Repository Importer Service* is to clone the target repository and then compress it for storage in the shared storage space. Compressing the repository provides easy management of the storage. Figure 4 illustrates the classes and their interactions in the *Repository Importer Service*.

Clients access this service by making a POST request to the endpoint provided by the *RepositoryResource* class, which creates a repository using the URL provided by the client. It interacts with the RepositoryDao (Data access object) to create a POJO (Plain old java object) for the particular repository. This object is added to a list of `Repository` objects for future reference. The list of repositories can be accessed using the method `retrieveURLs()`. The `Repository` class encapsulates a repository's url and provides methods for accessing it.

To implement the cloning of the repository into local storage, we utilize JGit [33], a Java library designed for Git operations. Listing 5 presents the code segment of the `clone()` method in the class *Importer*.

Listing 5. Cloning from a Git repository.

```
public void clone(String url){
    try{
        Git.cloneRepository()
            .setURI(url)
            .setDirectory(new File("../../clonedrepositories"))
            .call();
    } catch (Exception e){
        System.out.println("errors in cloning from git repository:"
        + e.toString());
    }
}
```

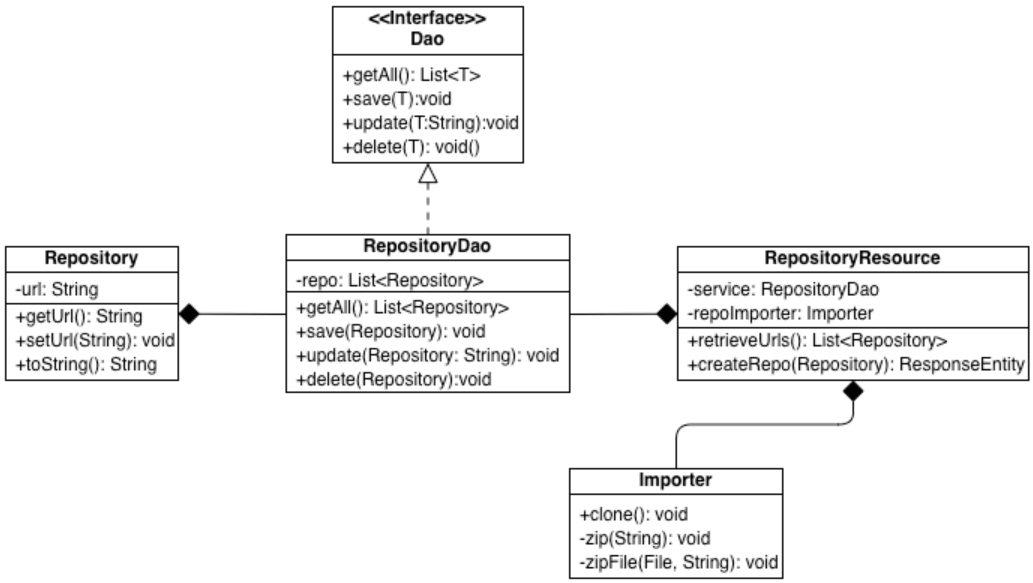

Figure 4. Design of the Repository Importer Service.

3.3.2. Vulnerability Checker Service

Once the repository has been successfully stored in the shared space, clients initiate a call to the *Vulnerability Checker* Service to identify potential vulnerabilities. This service employs the *SpotBugs* static analyzer to scan the target application for SQLi and XSS vulnerabilities. *SpotBugs* generates a detailed report outlining the location of vulnerabilities and their severity. The service places this report in the shared storage for access by the fixer service. In addition, it maintains a record of the URLs and the number of URLs submitted by users.

Figure 5 illustrates the design of the *Vulnerability Checker Service*. Clients use the endpoint provided by the *Checker* Service to initiate the scanning process of the target application. The *Checker* retrieves the repository from shared storage and decompresses it for scanning. The `newFile()` method serves as a helper utility for the `unzip()` method. After obtaining the repository, the *Checker* service proceeds to scan thetarget application for the SQLi and XSS vulnerabilities by invoking the *SpotBugs* application through the Command Line Interface (CLI). The `IOStreamConsumer` class is used internally to display command line output to the terminal.

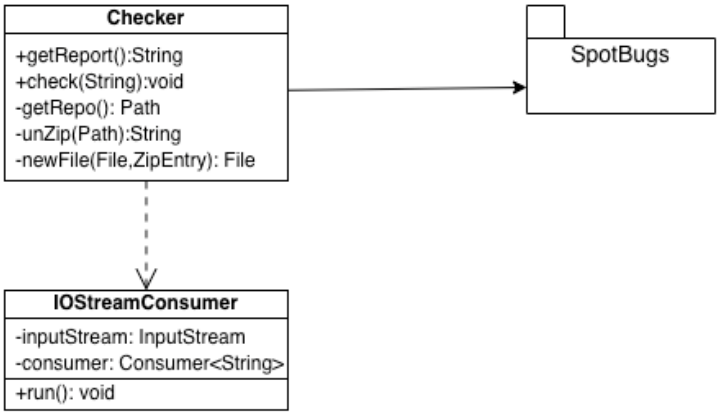

Figure 5. Design of the Vulnerability Checker Service.

Initially, the system checks the operating system to determine the correct command. CLI Commands in Linux and Windows differ. Two different commands are executed

sequentially: one scans for XSS, and the other scans for SQL injection vulnerabilities. Listing 6 presents the code snippet for invoking the XSS scanner and SQLi scanner in *SpotBugs* in Windows.

Listing 6. Invoking *SpotBugs*.

```
if (isWindows) {
    builder.command("cmd.exe", "/c", "\textit{SpotBugs}", "-textui",
        "-conserveSpace", "-pluginList", "findsecbugs-plugin-1.11.0",
        "-visitors", "CrossSiteScripting", "-nested:false", "-maxHeap",
        "2000", "-xml", "-output", "xssbugs.xml", "./");

    command2.command("cmd.exe", "/c", "\textit{SpotBugs}", "-textui",
        "-conserveSpace", "-pluginList", "findsecbugs-plugin-1.11.0",
        "-visitors", "SqlInjectionDetector", "-nested:false", "-maxHeap",
        "2000", "-xml", "-output", "sqlibugs.xml", "./");
}
```

3.3.3. Fixer Service

The *Fixer Service*, shown in Figure 6, uses the Intercepting Filter pattern to address SQli and XSS vulnerabilities. When a client requests the *Fixer* Service, the service is responsible for parsing the report generated by the *Checker* Service. Using the report, the *Fixer* Service identifies the location of vulnerable code in the target application. This information is used to retrieve input from the target application and pass it through a chain of filters for sanitization. Once the data are sanitized, they are returned to the target application.

The *Fixer* Service is called by clients through a REST endpoint exposed by the Fixer. Initially, the service retrieves the location of the target repository and then parses the vulnerability report generated by the Checker module to identify the type and location of the vulnerabilities in the target repository. Based on this report, a `BugInstance` is created, depending on whether the bug is an XSS or SQLi vulnerability. This is achieved using the Factory Method pattern [34]. The `XSSBug` and `SQLIBug` classes extend the `BugInstance` class, although they currently share the same implementation in their bodies. This design choice is made to enhance code extensibility. In the future, if additional states or behaviors are required, they can be easily incorporated.

This `BugInstance` is passed to the `fix()` method for remidiation. This method parses the target file to locate a vulnerable line of code. It then inserts a new line of code just before the vulnerable section, instantiating a filter based on the type of the vulnerability. The vulnerable code is passed as a parameter to this instantiation. The relevant filter is then copied to the target repository, effectively implementing the *Intercepting Filter* pattern.

The execution sequence in the target application is as follows: A client initiates a request to a module (Module_1) for information. Module_1 accepts the request as-is and passes it to the processor module, where the request is processed accordingly. However, Module_1 receives the raw request in a string format, which is vulnerable to SQLi and XSS attacks.

The *Fixer* Service identifies this vulnerable string, intercepts it, and passes it through a series of filters. A call to this new filtering process is then inserted into Module_1. Before forwarding the parameter string to the processor module, Module_1 invokes the `FilterManager` to process the string. The `FilterManager` determines which filter chain to activate based on the type of vulnerability. Currently, we are focusing on injection vulnerabilities. The input string undergoes sanitization in the `InjectionFilters`, using appropriate classes such as the `SQLiFilter` and `XssFilter`. The sanitized string is then returned to the original caller, which is Module_1, allowing for the continuation of normal execution processes.

Figure 7 presents the how the Intercepting Filter pattern is applied in CARES.

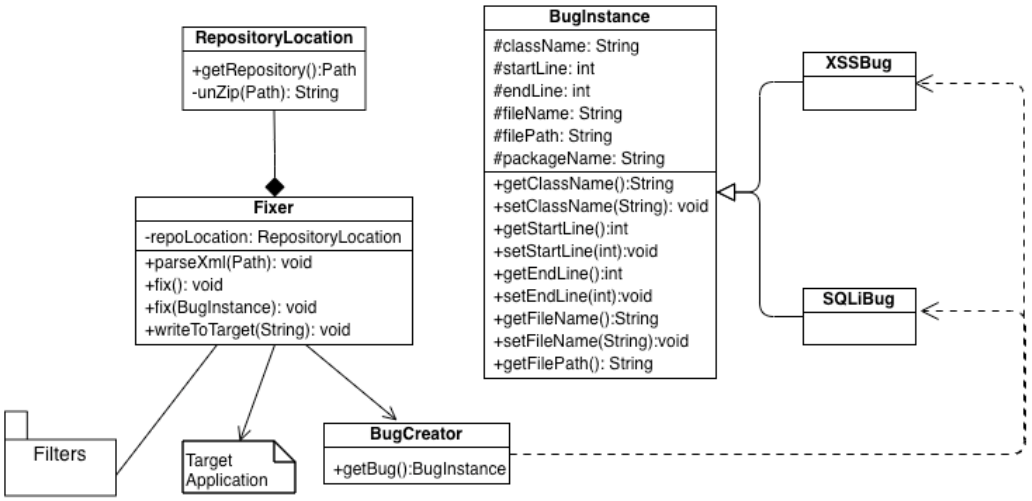

Figure 6. Design of the Fixer Service.

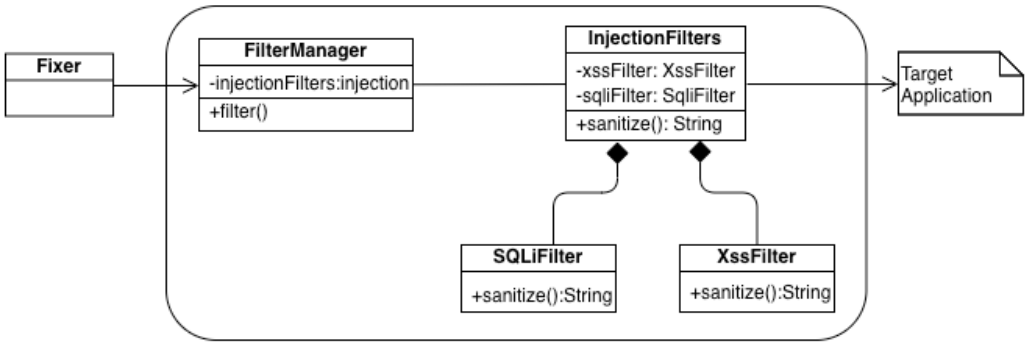

Figure 7. Application of Intercepting Filters in CARES.

The `FilterManager` is responsible for determining the category of filters to employ, whether they are injection filters or other types catering to different vulnerabilities. The code snippet of `FilterManager` can be found in Listing 7. In the scope of this study, our primary focus is on mitigating injection vulnerabilities. Consequently, a direct call to injection filter's `sanitize()` method has been incorporated. If CARES were to incorporate additional filters, a collection of additional filter classes would be developed and added to for a filter chain, and the `filter()` method would be responsible to identifying and sending the input to the appropriate filter in the filter chain.

Listing 7. FilterManager class.

```
public class FilterManager{
    private InjectionFilters injectionFilter;
    public FilterManager(){
      this.injectionFilter = new InjectionFilters();
    }
    public String filter(String input){
      return injectionFilter.sanitize(input);
    }
}
```

The `InjectionFilters` class handles injection vulnerabilities, specifically SQLi and XSS. As shown in Listing 8, in the `sanitize()` method, it determines whether the vulnerability is XSS or SQLi and then directs the input to the respective filters for sanitization. Since this version of CARES only deals with XSS and SQLi, we have simplified the hierarchy by omitting a `Filter` interface for `SQLiFilter` and `XssFilter` to implement.

Listing 8. InjectionFilters class.

```
public class InjectionFilters{
    private SQLiFilter sqlFilter;
    private XssFilter xssFilter;

    public String sanitize(String input){
      if(getType(input).equalsIgnoreCase("sqli")){
        sqlFilter = new SQLiFilter();
        return sqlFilter.sanitize(input);
      }
      else if(getType(input).equalsIgnoreCase("xss")){
        xssFilter = new XssFilter();
        return xssFilter.sanitize();
      }
      else{
        return input;
      }
    }
}
```

To mitigate XSS vulnerabilities, we follow the recommended strategies outlined by OWASP [35]. Using the Java HTML Sanitizer API, as illustrated in Listing 9, we create a policy for validating the input string, specifying all permitted tags within the policy. After sanitizing the string based on the policy, we encode it using the Java Encoder API to properly handle HTML tag escaping.

Listing 9. sanitize() in XssFilter.

```
public String sanitize(String input){
    PolicyFactory policy = new HtmlPolicyBuilder()
        .allowElements("p", "strong", "h1", "h2", "h3"
        , "h4", "h5", "h6", "div", "li", "ul", "ol")
        .toFactory();
    String safeString = policy.sanitize(input);
    safeString = Encode.forHtml(input);
    return safeString;
}
```

For sanitizing an input string to SQL injection, we utilize the Apache Commons library's `StringEscapeUtils` class, which sanitizes the string, rendering it suitable for execution. The code snippet is shown in Listing 10.

Listing 10. sanitize() in SQLiFilter.

```
public String sanitize(String input){
    String safeString = StringEscapeUtils.escapeSql(input);
    return safeString;
}
```

3.4. The Microservice Communications

We have chosen HTTP request–response communication to enable the independent development of each microservice. The communication speed is not a priority concern for CARES. However, as part of our future expansion plans, CARES will support different programming languages beyond Java. Using standard HTTP/HTTPS protocols allows compatibility with different languages and platforms without the requirement for distinct protocol buffers and code generators to be tailored to each language, as seen in gRPC. Thus, we have implemented the Representational State Transfer (REST) API, allowing access to each module through endpoints.

An API, or application programming interface, serves as a contract between an information provider and an information user, specifying the content required for both the consumer's request and the producer's response. REST is a set of architectural constraints that developers conform to while building the APIs. When a client request is made via a RESTful API, it transfers a representation of the resource's state to the requester or endpoint. This information is delivered in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, Python, PHP, or plain text. JSON, known for its language-agnostic nature and human-machine readability, is the most commonly used format.

As an example, Listing 11 illustrates a REST endpoint in the *Repository Importer* Service implemented in Spring Boot.

Listing 11. A typical REST endpoint in Spring Boot.

```
@PostMapping("/repos")
public ResponseEntity createRepo(@RequestBody Repository repo){
    service.save(repo);
    repoImporter.clone(repo.getUrl());
    URI location =
       ServletUriComponentsBuilder.fromCurrentRequest().build().toUri();

    return ResponseEntity.created(location).build();
}
```

4. Results

In this section, we present the application of CARES in remediating SQLi and XSS vulnerabilities in a health application, using the Tolven platform [10] as our case study. The Tolven platform is an electronic health record system, allowing both consumers and clinicians to manage their health records. Specifically, the Tolven ePHR is an intuitive web-based application designed for creating, viewing, storing, and sharing healthcare information. This application was developed using the Java Enterprise Edition (EE) with the Enterprise Java Beans (EJB) module and is an open-source project. The last update to its source code, dating back to 2016, is available on SourceForge [36]. Despite its powerful functionalities, the application has numerous security vulnerabilities, with over 500 instances of XSS and SQLi vulnerabilities identified during scanning with SpotBugs. In addition to security bugs, SpotBugs flags incorrectness coding, bad coding practices, performance issues, and more as bugs. However, CARES only retrieves bugs specifically indicated as SQLi or XSS vulnerabilities. Bugs falling outside the scope of SQLi and XSS are not reported within CARES to align with the focus of our study. When SpotBugs is used independently to scan the code of Tolven, it reports over 9000 bugs of different types. Among them, around 5% are designated as XSS and SQLi vulnerabilities.

To use CARES to remediate the vulnerabilities, the application code needs to be hosted on GitHub. Therefore, we downloaded the Tolven application and uploaded it to GitHub. The GitHub link to the application was provided in the *Repository URL* textbox in the CARES user interface, as depicted in Figure 8. The Importer Service cloned the repository to local storage. The duration of the cloning process may vary, particularly when dealing with larger repositories, and could take several minutes to complete.

After the repository was cloned, we initiated the scanning process by clicking the "Check" button. The *Checker* Service then called the *SpotBugs* application, which scanned the repository and generated a vulnerability report in XML. Figure 9 displays a partial report. The provided details includes the vulnerablity type (SQLi or XSS), the priority attribute indicating the severity of each vulnerability (with lower values signifying a higher severity), full class name, file path, start and end line numbers of each vulnerability instance, and bytecode information. While bytecode information can be used to enhance the execution speed, it falls outside the scope of this study. This report highlighted an SQLi vulnerability located in line 420 of the class `SnapshotBean` within the `org.tolven.analysis.bean`

package. Specifically, it identified the method `deleteCohortPlaceholder` and the vulnerable code location in `createQuery()`. The vulnerable code has been demonstrated in Listing 2. CARES analyzed and stored the type and location of each vulnerability for further remediation.

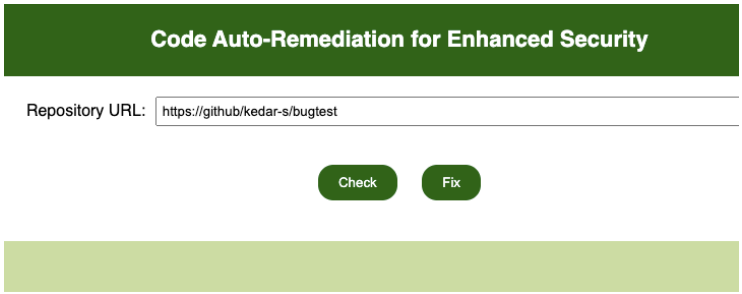

Figure 8. CARES user interface.

```
<?xml version="1.0" encoding="UTF-8"?>

<BugCollection version="4.2.2" sequence="0" timestamp="1638292130515" analysisTim
  <Project projectName="">
    <Jar>./</Jar>
  </Project>
  <BugInstance type="SQL_INJECTION_JPA" priority="2" rank="12" abbrev="SECSQLIJPA
  <BugInstance type="SQL_INJECTION_JPA" priority="2" rank="12" abbrev="SECSQLIJPA
    <Class classname="org.tolven.analysis.bean.SnapshotBean">
      <SourceLine classname="org.tolven.analysis.bean.SnapshotBean" start="61" en
    </Class>
    <Method classname="org.tolven.analysis.bean.SnapshotBean" name="deleteCohortP
      <SourceLine classname="org.tolven.analysis.bean.SnapshotBean" start="420" e
    </Method>
    <SourceLine classname="org.tolven.analysis.bean.SnapshotBean" start="420" end
    <String value="javax/persistence/EntityManager.createQuery(Ljava/lang/String;
    <String value="0" role="Sink parameter"/>
    <String value="org/tolven/analysis/bean/SnapshotBean.deleteCohortPlaceholder(
    <String value="not detected" role="Method usage"/>
    <SourceLine classname="org.tolven.analysis.bean.SnapshotBean" start="421" end
    <SourceLine classname="org.tolven.analysis.bean.SnapshotBean" start="422" end
  </BugInstance>
```

Figure 9. Partial vulnerability report.

The "Fix" button triggered the *Fixer* Service to address these two types of the vulnerabilities in code. The *Fixer* Service carried out this remediation by copying and integrating the filter classes, including `FilterManager` and `InjectionFilters`, into its target repository. Then, the *Fixer* Service instantiated filter objects corresponding to the vulnerability type and inserted them into the code at the reported location of each vulnerability. For instance, in the code shown in Listing 2, an SQLi vulnerability was identified at line 420. CARES addressed this by instantiating a `FilterManager` object, configuring the appropriate filter (`SQLiFilter` in this case) and applying it to sanitize the query string in the `createQuery()` method, transforming it into a secure query string. The auto-corrected code is presented in Listing 12.

Listing 12. Fixed SQLi vulnerability in code.

```
418  @override
419  public void deleteCohortPlaceholder(Account account,String cohortType){
420      Query query = em.createQuery(new FilterManager().filter(
421              "delete PlaceholderID pi where "
422              + "pi.account.id = :account and pi.extension LIKE '"
423              + cohortType + "%'"));
424      query.setParameter("account", account.getId());
425      query.executeUpdate();
426  }
```

Listing 4 exhibited an XSS vulnerability at line 181. To remediate this vulnerability, the *Fixer* Service followed the same procedure used for SQLi vulnerabilities, instantiated a `FilterManager` object, configured the filter `XssFilter`, and passed the argument in `writer.write()` to the filter for sanitization. As a result, the input for the Ajax request was secured, as demonstrated in Listing 13.

Listing 13. Fixed XSS vulnerability in code.

```
179  // prepare xml
180  writer.write("<ajax-response>");
181  writer.write(new FilterManager().filter("<response path=\"" + path
182          + "\" role=\"" + rolename +"\" defpath=\"" + defPathSuffix
183          + "\" " + " title=\"" + title + "\">" ));
```

The entire refactoring process from cloning the code to fixing the code takes around 7 min. Upon completing the code refactoring process, we further checked the refactored Tolven system with CARES. The outcome was highly successful, as all SQLi and XSS vulnerabilities identified in the original Tolven application had been successfully mitigated. SpotBugs no longer reported any instances of these two types of vulnerabilities in the refactored Tolven system.

The design of the CARES facilitates the development of the system. Using microservices in the architectural design, we developed and tested each microservice independently. By applying the Intercepting Filter pattern in the *Fixer Service*, we were able to implement each filter separately and test it thoroughly. In addition, we developed different versions of `SQLiFilter` and `XssFilter` with different mitigation algorithms to test their effectiveness. We selected the implementations presented Section 3.3. Due to the independence of the *Fixer Service* as a microservice and the flexibility of replacing filters in the Intercepting Filter pattern, the code modifications were limited to the most affected part in the *Fixer Service*.

5. Threats to Validity

This section summarizes the potential threats to the validity of our findings.

The first aspect concerns the implementation of the mitigation strategies. The current implementation of XSS mitigation, applying a whitelist of allowed HTML elements, poses validity challenges. While the whitelist prevents malicious script inclusion, it may unintentionally exclude legitimate content that requires other HTML elements for proper rendering and functionality. In addition, the whitelist lacks consideration for potential vulnerabilities within attributes of the allowed elements. The implementation of SQLi mitigation uses Apache Commons library functions for sanitizing input strings. Validity concerns for this approach are twofold: firstly, the system must stay updated with the latest version of the library; secondly, the library is designed for broad and general character patterns in SQLi, which may not cover all possible edge cases. Future enhancements will address these validity concerns, as indicated in Section 6.

The second aspect is that the current version of CARES is implemented for mitigating SQLi and XSS vulnerabilities in Java applications only. Thus, the validity of the testing and evaluations is limited to Java applications. CARES will be extended to support multiple languages as part of the research plan. Although the Intercepting Filter pattern originated in the Java world, its nature as a design pattern allows for implementation in different languages, which enables future extensions beyond Java applications.

6. Discussion

The outcomes of the current version of CARES in mitigating SQLi and XSS in Java applications indicate that applying the Intercepting Filter pattern is efficient. However, we have identified aspects that should be improved in CARES.

As indicated in Section 5, CARES uses a whitelist to allow specific HTML elements as the mitigation strategy for XSS vulnerabilities. Such an approach can still pose vulnerabilities when an attacker injects malicious script using an attribute. For example, the

following input would pass the `XssFilter`, yet it has a potential threat when the code in `onmouseover` is malicious:

```
<p onmouseover="bad script here">Hover</p>
```

To address this, we will implement context-aware sanitization [37] in addition to the existing whitelist input sanitization.

CARES applies the method `escapeSql()` to sanitize a string for use in an SQL query. The method does not specifically escape semicolons (;) or double hyphens (--). If an attacker includes a semicolon and a double hyphen in forming a malicious input, it may still pass the `SQLFilter`. Our enhancement to this approach is to introduce prepared statements as an additional strategy for the input sanitization.

To support multiple mitigation strategies, the *Fixer Service* will integrate the Secure Strategy Factory pattern in addition to the existing Intercepting Filters pattern.

With future extension and modification to CARES in mind, we employed microservices to design each component on the server-side, making CARES a foundational step towards a more extensive application for code refactoring. Every functionality in the application has the potential to expand into a broader domain, facilitated by the independent deployability and development of microservices.

Currently CARES's *Repository Importer* Service focuses on cloning projects solely from the GitHub repository. This capability could be extended to include the cloning of repositories from various version control systems or even local sources. For example, projects utilizing Centralized Version Control (CVS), Apache Subversion, or Mercurial can be accommodated by modifying the *Repository Importer* Service. Enabling these options offers users greater flexibility. Any necessary changes can be facilitated by re-implementing the *Importer* class within the *Repository Importer* Service.

In the current version of CARES, the *Checker Service* uses *SpotBugs* to identify and locate SQLi and XSS vulnerabilities in Java code. This capability could be expanded to scan applications developed in different programming languages. For example, Security Code Scan for .NET platform (https://security-code-scan.github.io/ (accessed on 1 January 2024)) is an open-source static vulnerability analysis tool for .NET applications. Incorporating such a tool within the *Checker Service* would facilitate the identification and location of SQLi and XSS vulnerabilities in .NET applications. If CARES is to be extended to support the identification of other vulnerabilities in applications developed in various programming languages, and no suitable open-source static analysis tools are available, custom scanner development will be needed to integrate with the *Checker Service*.

The *Fixer Service* in CARES could be extended to address applications developed in various programming languages. In this scenario, an implementation of the Intercepting Filters pattern using the specific language would be added. CARES is flexible to expand the supported vulnerability types, such as addressing CRLF vulnerabilities. For this extension, new `Filter` classes tailored to the newly introduced vulnerability type would be developed as part of the implementation of the Intercepting Filters pattern.

7. Conclusions

In this study, we designed and developed CARES, a novel code refactoring application that automatically mitigates the SQLi and XSS vulnerabilities from Java web applications. By integrating the *SpotBugs* static analysis tool to locate vulnerable code in target applications and employing the Intercepting Filters pattern to dynamically inject specific code addressing vulnerability types, CARES effectively eliminates XSS and SQLi vulnerabilities from the target application.

Designed with a microservice architecture, CARES demonstrates extensibility and flexibility to adapt to future changes in the application. These changes may include alternative repository options for importing target applications, support for mitigating more vulnerability types, the addition of secure design patterns for multi-level mitigation strategies, and more. Each change can be addressed by modifying one microservice at a time without impacting others.

Moving forward, several enhancements can further enhance the system and enrich user experience:

- *Enhanced front-end:* The current front-end application is minimalistic. Enhancements could include postscan features, such as displaying the number of identified bugs within the target project and possibly presenting selected or all bugs in an organized report. This report could report the details of each bug type and its severity.
- *User profiles:* Implementing a profile-based system allows users to create accounts to store records of their past actions, including scanned repositories and generated reports.
- *Support for diverse programming languages:* CARES will expand its capabilities to scan and fix applications developed in different programming languages other than Java. A new version of CARES will target vulnerabilities in applications built with .NET.
- *Broader vulnerability support:* In alignment with OWASP's top ten vulnerability categories, CARES will be expanded to address a wider range of injection-based security vulnerabilities, such as *Path Traversal* (CWE-22).

Author Contributions: Conceptualization, Y.L.; methodology, Y.L. and K.S.; software, K.S.; validation, K.S. and Y.L.; writing—original draft preparation, K.S. and Y.L.; writing—review and editing, K.S. and Y.L.; All authors have read and agreed to the published version of this manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The study used the publicly available dataset "Tolven Health Record" to evaluate CARES. The dataset can be accessed at the following link: https://sourceforge.net/projects/tolven.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Edgescan. 2023 Vulnerability Statistics Report. 2023. Available online: https://www.edgescan.com/intel-hub/stats-report/ (accessed on 1 January 2024).
2. Veracode. State of Software Security 2023: Annual Report on the State of Application Security. Available online: https://www.veracode.com/state-of-software-security-report (accessed on 1 January 2024).
3. Edgescan. 2021 Vulnerability Statistic Report Press Release, 2021. Available online: https://www.edgescan.com/2020-vulnerability-statistic-report-press-release/ (accessed on 1 January 2024).
4. O'Driscoll, A. 25+ Cyber Security Vulnerability Statistics and Facts of 2023. Available online: https://www.comparitech.com/blog/information-security/cybersecurity-vulnerability-statistics/ (accessed on 1 January 2024).
5. Vulnerability Scanning Tools. Available online: https://owasp.org/www-community/Vulnerability_Scanning_Tools (accessed on 1 January 2024).
6. Wapiti. The Web-Application Vulnerability Scanner. Available online: https://wapiti-scanner.github.io/ (accessed on 1 January 2024).
7. Higgins, J.K. The Cost of Fixing an Application Vulnerability. Available online: https://www.darkreading.com/risk/the-cost-of-fixing-an-application-vulnerability/d/d-id/113104 (accessed on 1 January 2024).
8. Ross, A. Why Fixing Security Vulnerabilities Is Not That Simpley. Available online: https://securityintelligence.com/posts/why-fixing-security-vulnerabilities-is-not-that-simple/ (accessed on 1 January 2024).
9. CWE-89: Improper Neutralization of Special Elements Used in an SQL Command ('SQL Injection'). Common Weakness Enumeration. Available online: https://cwe.mitre.org/data/definitions/89.html (accessed on 1 January 2024).
10. Mathis, B. The "Unified Platform" That Delivers All-in-One EHR/PHR/HIE. Available online: https://www.openhealthnews.com/articles/2014/tolven-%E2%80%9Cunified-platform%E2%80%9D-delivers-all-one-ehrphrhie (accessed on 1 January 2024).
11. Janot, E.; Zavarsky, P. Preventing SQL injections in online applications: Study, recommendations and Java solution prototype based on the SQL DOM. In Proceedings of the OWASP Application Security Conference, Ghent, Belgium, 19–22 May 2008.
12. Wei, K.; Muthuprasanna, M.; Kothari, S. Preventing SQL Injection Attacks in Stored Procedures. In Proceedings of the Australian Software Engineering Conference (ASWEC'06), Sydney, Australia, 18–21 April 2006; IEEE: Piscataway, NJ, USA, 2006; p. 8.
13. CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-Site Scripting'). Common Weakness Enumeration. Available online: https://cwe.mitre.org/data/definitions/79.html (accessed on 1 January 2024).

14. Liu, M.; Zhang, B.; Chen, W.; Zhang, X. A Survey of Exploitation and Detection Methods of XSS Vulnerabilities. *IEEE Access* **2019**, *7*, 182004–182016. [CrossRef]
15. Franken, G.; Van Goethem, T.; Desmet, L.; Joosen, W. A Bug's Life: Analyzing the Lifecycle and Mitigation Process of Content Security Policy Bugs. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 3673–3690.
16. Chen, J.; Jiang, J.; Duan, H.; Wan, T.; Chen, S.; Paxson, V.; Yang, M. We still {Don't} have secure {Cross-Domain} requests: An empirical study of {CORS}. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 1079–1093.
17. Khodayari, S.; Pellegrino, G. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 1590–1607.
18. Dougherty, C.; Sayre, K.; Seacord, R.C.; Svoboda, D.; Togashi, K. *Secure Design Patterns*; Software Engineering Institution, Carnegie-Mellon University: Pittsburgh, PA,USA, 2009.
19. Fernandez, E.B. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*; John Wiley & Sons: Hoboken, NJ, USA, 2013.
20. Ratnaparkhi, A.; Ezenwoye, O.; Liu, Y. From Vulnerability Anti-Patterns to Secure Design Patterns. In Proceedings of the International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, PA,USA, 1–10 July 2021.
21. Alur, D.; Crupi, J.; Malks, D. *Core J2EE Patterns: Best Practices and Design Strategies*; Gulf Professional Publishing, Houston, TX, USA, 2003.
22. Fowler, M. Microservices: A Definition of This New Architectural Term. Available online: https://martinfowler.com/articles/microservices.html (accessed on 1 January 2024).
23. Cerný, T.; Donahoo, M.J.; Pechanec, J. Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. In Proceedings of the International Conference on Research in Adaptive and Convergent Systems, Krakow, Poland, 20–23 September 2017.
24. Google. Introduction to gRPC, 2021. Available online: https://grpc.io/docs/what-is-grpc/introduction/ (accessed on 1 January 2024).
25. Fernando, R. Evaluating Performance of REST vs. gRPC, 2019. Available online: https://medium.com/@EmperorRXF/evaluating-performance-of-rest-vs-grpc-1b8bdf0b22da (accessed on 1 January 2024).
26. Barnea, B.; Harpaz, O. Critical Remote Code Execution Vulnerabilities in Windows RPC Runtime, 2022. Available online: https://www.akamai.com/blog/security/critical-remote-code-execution-vulnerabilities-windows-rpc-runtime (accessed on 1 January 2024).
27. Baracuda. Available online: https://www.barracuda.com/ (accessed on 1 January 2024).
28. Barracuda Automates Web Application Vulnerability Remediation and Security Policy Enforcement. Available online: https://solutionsreview.com/backup-disaster-recovery/barracuda-automates-web-application-vulnerability-remediation-and-security-policy-enforcement/ (accessed on 1 January 2024).
29. Software Assurance. Available online: https://www.us-cert.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf (accessed on 1 January 2024).
30. Courant, J. Developer-Proof Prevention of SQL Injections. In Proceedings of the International Symposium on Foundations and Practice of Security, Montreal, QC, Canada, 1–3 December 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 82–99.
31. Ratnaparkhi, A.; Liu, Y. Towards Tackling Common Web Application Vulnerabilities Using Secure Design Patterns. In Proceedings of the IEEE International Conference on Electro Information Technology, Mt. Pleasant, MI, USA, 14–15 May 2021.
32. VMware. Spring Boot, 2023. Available online: https://spring.io/projects/spring-boot (accessed on 1 January 2024).
33. Eclipse Foundation. JGit: Java Implementation of Git. Available online: https://www.eclipse.org/jgit/ (accessed on 1 January 2024).
34. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley Professional: Boston, MA, USA, 1994.
35. KirstenS. Cross Site Scripting (XSS). Available online: https://owasp.org/www-community/attacks/xss/ (accessed on 1 January 2024).
36. Tolven. Tolven Health Record, 2016. Available online: https://sourceforge.net/projects/tolven/ (accessed on 1 January 2024).
37. Gupta, S.; Gupta, B.B. CSSXC: Context-sensitive Sanitization Framework for Web Applications against XSS Vulnerabilities in Cloud Environments. *Procedia Comput. Sci.* **2016**, *85*, 198–205. [CrossRef]