BOOKHIVE LIBRARY MANAGEMENT SYSTEM (LMS)

WWW Structures, Techniques and Standards
CSI 3140

Static Code Analysis Report
Phase 3

Team #33

Student Number	Name	Email
300163562	Lara Lim	llim067@uottawa.ca
300173889	Amani Farid	afari094@uottawa.ca
300184721	Samy Touabi	stoua026@uottawa.ca
300098986	Kian Zahrai	kzahr091@uottawa.ca

Date Submitted: July 23rd, 2023
Professor: Khalil Abuosba

Introduction

One of the fundamental building blocks of software is code quality. Improved software quality is linked to high-quality code. Your source code's quality correlates with whether your app is secure, stable, and reliable. To sustain quality, many development teams embrace techniques like code review, automated testing, and manual testing.

While code review and automated tests are important for producing quality code, they will not uncover all issues in software. Because code reviewers and automated test authors are humans, bugs and security vulnerabilities often find their way into the production environment. Snyk is primarily known for its capabilities in identifying security vulnerabilities and issues in open-source dependencies within a codebase. It offers both static code analysis and dependency scanning to detect potential security risks in each codebase. The following describes how the tool operates:

1. **Codebase Scanning**: Snyk integrates with different version control systems like Git and scans the codebase for dependencies, including direct and transitive dependencies. It looks for various package manifests (e.g., package.json for JavaScript, requirements.txt for Python) to identify the dependencies used by the application.
2. **Dependency Analysis**: Once Snyk identifies the dependencies, it checks them against a vulnerability database, which includes known security issues and vulnerabilities reported for various packages. This database is regularly updated with the latest security information.
3. **Vulnerability Detection**: Snyk then matches the dependencies found in the codebase against the vulnerability database. If any known security vulnerabilities are detected, Snyk generates a report detailing the specific issues found, along with information on the severity of each vulnerability.
4. **Remediation Guidance**: Snyk not only provides vulnerability reports but also offers guidance on how to fix or mitigate these vulnerabilities. This may involve updating the affected packages to a secure version or finding alternative dependencies that are not affected by the vulnerabilities.
5. **Continuous Monitoring**: Snyk can be set up to provide continuous monitoring, alerting developers whenever new vulnerabilities are reported in the dependencies used in the codebase. This feature helps ensure that any new vulnerabilities that may arise are promptly addressed.
6. **Integration with CI/CD**: Snyk can be integrated into the Continuous Integration/Continuous Deployment (CI/CD) pipeline to automate security scanning during development and deployment stages. This integration helps catch vulnerabilities early in the development lifecycle and ensures that only secure code is deployed to production.

As a result, this report shall present the findings and code issues of the tool, as the software project is integrated for the capabilities of the tool to operate upon.

Overall Static Code Analysis:

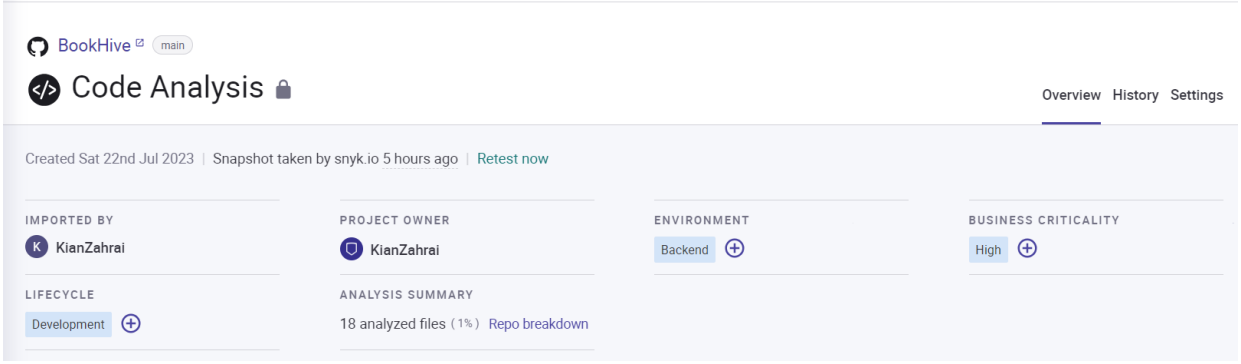
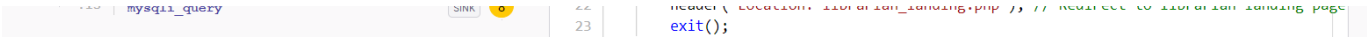


Exhibit A: SQL Injection



Error Code: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

In an SQL injection attack, the user can submit an SQL query directly to the database, gaining access without providing appropriate credentials. Attackers can then view, export, modify, and delete confidential information; change passwords and other authentication information; and gain access to other systems within the network. This is one of the most exploited categories of vulnerability but can be avoided through good coding practices.

Details:

In the file `librarians/librarian_login.php`, unsanitized input from an HTTP parameter flows into `mysqli_query`, where it is used in an SQL query. It refers to data that has not been meticulously cleaned or filtered to remove potentially harmful characters or escape sequences before being used in an operation. In the context of SQL injection, it means that the data received from the user has not been sanitized for special characters that could alter the intended behavior of an SQL query. This may result in an SQL Injection vulnerability. the input data from the user's email address, retrieved from `$_POST["email"]`, is considered "tainted" because it is coming from an insecure source (the user) and has not been validated for malicious content or harmful characters.

In the source line, data is 'tainted' if it comes from an insecure source such as a file, the network, or the user. the email input data flows into `mysqli_query` without being sanitized. This means that the data is directly used in an SQL query without any validation or proper handling, leaving the application vulnerable to SQL injection attacks. This tainted data is passed to a sensitive sink. Sinks are the operations that must receive clean data and that you would not want an attacker to be able to manipulate.

Source: `$email = $_POST["email"];`

Sink: `$result = mysqli_query($conn, $query);`

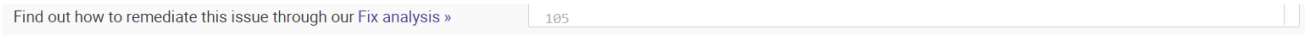
The combination of "tainted data" and "unsanitized input" in this situation can lead to a serious security risk. Attackers may craft malicious input that includes SQL commands, and if this input is not properly sanitized, it can be executed as part of the SQL query, leading to unauthorized access, data manipulation, or even the potential to gain control over the database or the entire system. To prevent such vulnerabilities, it is crucial to validate, sanitize, and properly escape user input before using it in any sensitive operations, especially when dealing with databases and SQL queries.

Fix Analysis:

- Avoid passing user-entered parameters directly to the SQL server.
- Avoid using string concatenation to build SQL queries from user-entered parameters.
- When coding, define SQL code first, then pass in parameters. Use prepared statements with parameterized queries. Examples include `SqlCommand()` in .NET and `bindParam()` in PHP.
- Use strong typing for all parameters so unexpected user data will be rejected.
- Where direct user input cannot be avoided for performance reasons, validate input against a strict allowlist of permitted characters, avoiding special characters (`?`, `&`, `/`, `<`, `>`, `;`, `-`, `'`, `"`, `\`, and spaces). Use a vendor-supplied escaping routine if possible.
- Develop your application in an environment and/or using libraries that provide protection against SQL injection.
- Harden your entire environment around a least-privilege model, ideally with isolated accounts with privileges only for tasks.

The following exhibits are also of type SQL injections, with various sources and sinks.

Exhibit B: SQL Injection



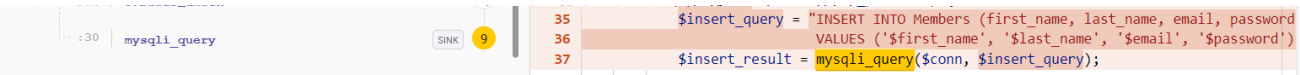
Error Code: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Details:
In the file `librarians/librarian_return.php`, unsanitized input from an HTTP parameter flows into `mysqli_query`, where it is used in an SQL query. This may result in an SQL Injection vulnerability. In the lines:

Source:
```
$book_id = $_POST['book_id'];
changeStatusToAvailable($book_id);
```

Sink:
```
$update_result = mysqli_query($conn, $update_query);
```

Exhibit C: SQL Injection



Error Code: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Details:
In the file `librarians/librarian_add_student.php`, unsanitized input from an HTTP parameter flows into `mysqli_query`, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Source: `$password = $_POST['password'];`

Sink: `$insert_result = mysqli_query($conn, $insert_query);`

Exhibit D: SQL Injection

```
65
66                displayBooks($result);
```

Error Code: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

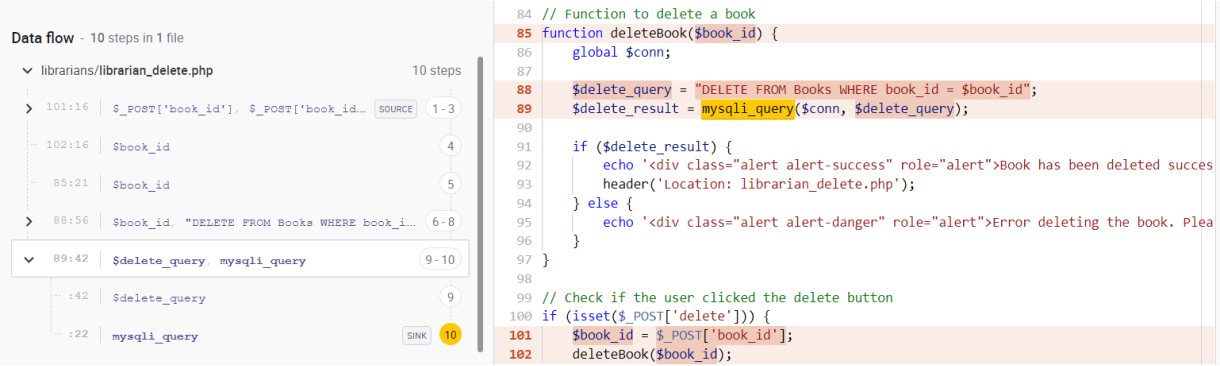
Details:

In the file `members/recommendations.php`, unsanitized input from an HTTP parameter flows into `mysqli_query`, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Source: `$userThoughts = $_POST["thoughts"];`

Sink: `$result = mysqli_query($conn, $sql);`

Exhibit E: SQL Injection



Error Code: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

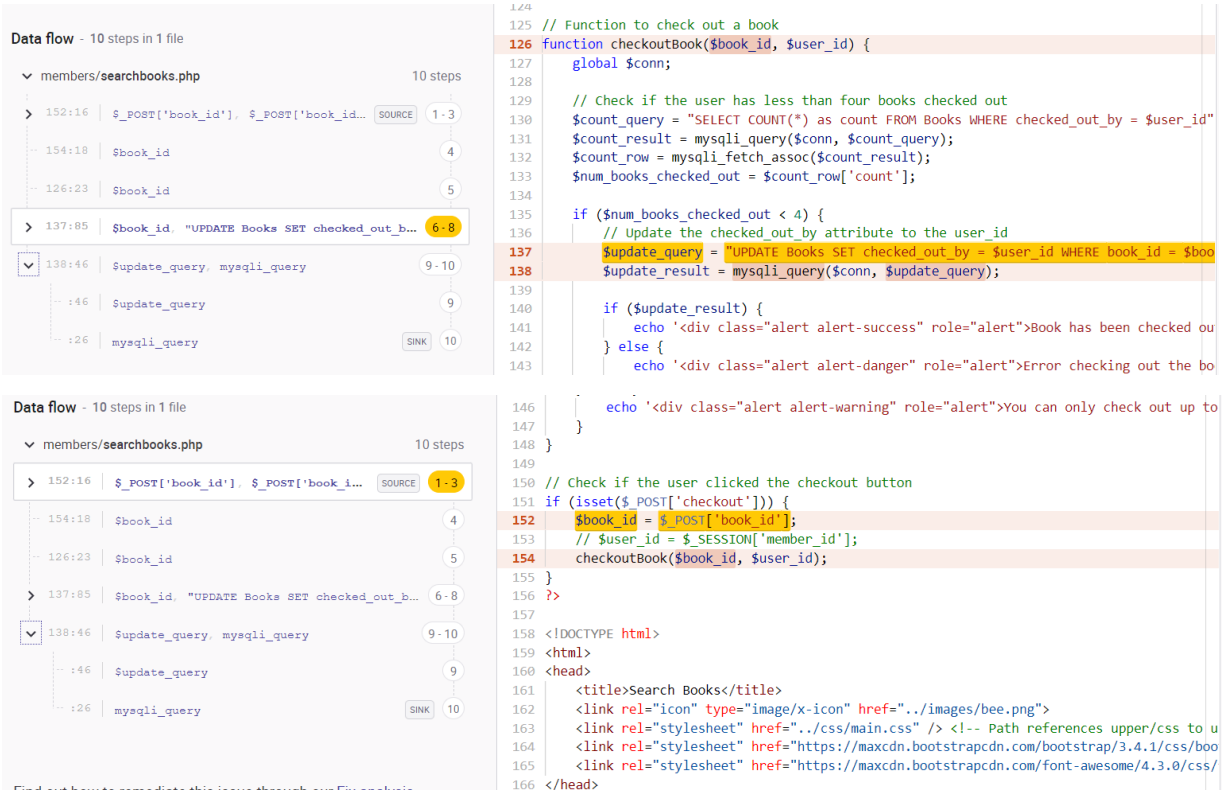
Details:

In the file `members/librarian_delete.php`, unsanitized input from an HTTP parameter flows into `mysqli_query`, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Source: `$book_id = $_POST['book_id'];`

Sink: `$delete_result = mysqli_query($conn, $delete_query);`

Exhibit F: SQL Injection



Error Code: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

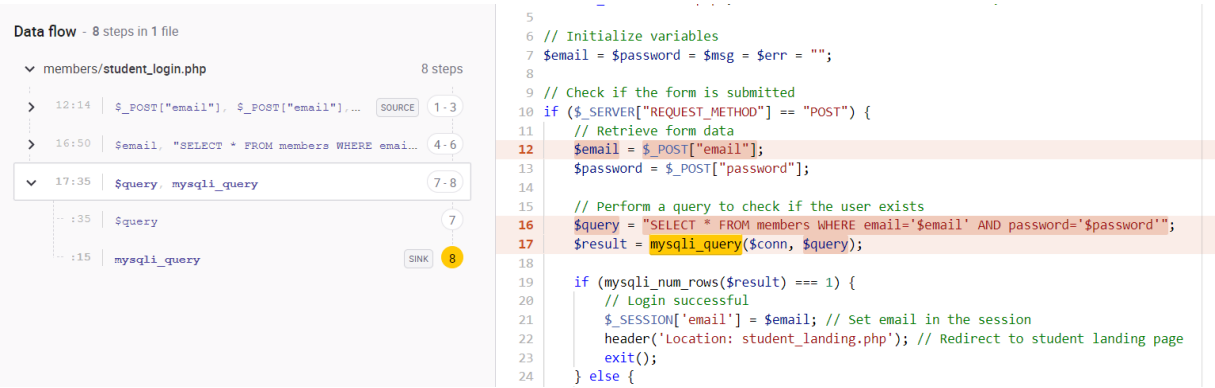
Details:

In the file `members/searchbooks.php`, unsanitized input from an HTTP parameter flows into `mysqli_query`, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Source: `$book_id = $_POST['book_id'];`

Sink: `$update_result = mysqli_query($conn, $update_query);`

Exhibit G: SQL Injection



Error Code: CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

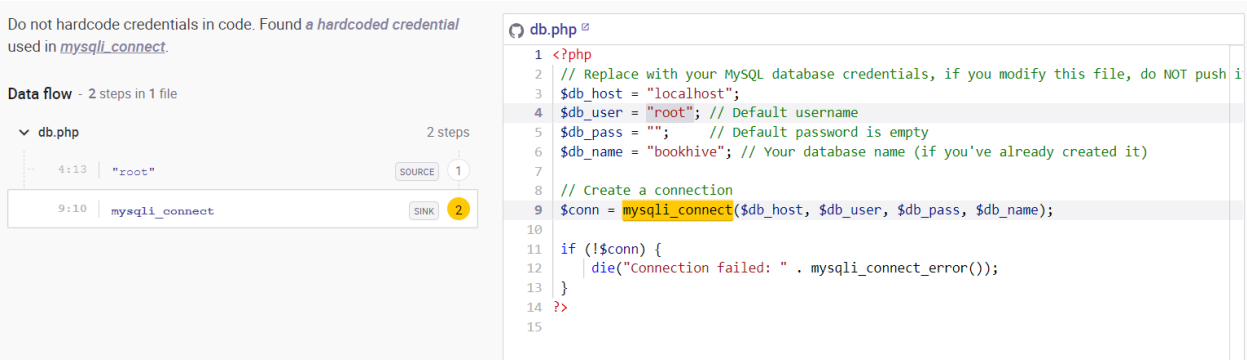
Details:

In the file `members/student_login.php`, unsanitized input from an HTTP parameter flows into `mysqli_query`, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Source: `$email = $_POST["email"];`

Sink: `$result = mysqli_query($conn, $query);`

Exhibit H: Use of Hardcoded Credentials



Error Code: CWE-798: Use of Hard-coded Credentials
CWE-259: Use of Hard-coded Password

Developers may use hardcoded credentials for convenience when coding in order to simplify their workflow. While they are responsible for removing these before production, occasionally this task may fall through the cracks. This also becomes a maintenance challenge when credentials are re-used across multiple applications.

Once attackers gain access, they may take advantage of privilege level to remove or alter data, take down a site or app, or hold any of the above for ransom. The risk across multiple similar projects is even greater. If code containing the credentials is reused across multiple projects, they will all be compromised.

Details:

Do not hardcode credentials in code. Found a hardcoded credential used in `mysqli_connect` in the `db.php` file.

Source: `$db_user = "root"; // Default username`

Sink: `$conn = mysqli_connect($db_host, $db_user, $db_pass, $db_name);`

Fix Analysis:

- Plan software architecture such that keys and passwords are always stored outside the code, wherever possible.
- Plan encryption into software architecture for all credential information and ensure proper handling of keys, credentials, and passwords.
- Prompt for a secure password on first login rather than hard code a default password.
- If a hardcoded password or credential must be used, limit its use, for example, to system console users rather than via the network.
- Use strong hashes for inbound password authentication, ideally with randomly assigned salts to increase the difficulty level in case of brute-force attack.