

National College of Ireland

Project Submission Sheet – 2021/2022

Student Name: Simon Lowry

Student ID: x21168938

Programme: MSCCYBE\_JANOL\_O Year: 2 of 2

Module: Secure Programming for Application Development

Lecturer: Liam McCabe

Submission Due Date: 30th April 2023

Project Title: Project - Secure Programming for Application Development

Word Count: 8680

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project. **ALL** internet material must be referenced in the references section. Students are encouraged to use the Harvard Referencing Standard supplied by the Library. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action. Students may be required to undergo a viva (oral examination) if there is suspicion about the validity of their submitted work.

Signature: *Simon Lowry*

Date: *30/04/2023*

PLEASE READ THE FOLLOWING INSTRUCTIONS:

1. Please attach a completed copy of this sheet to each project (including multiple copies).
2. Projects should be submitted to your Programme Coordinator.
3. **You must ensure that you retain a HARD COPY of ALL projects**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. Please do not bind projects or place in covers unless specifically requested.
4. You must ensure that all projects are submitted to your Programme Coordinator on or before the required submission date. **Late submissions will incur penalties.**
5. All projects must be submitted and passed in order to successfully complete the year. **Any project/assignment not submitted will be marked as a fail.**

<b>Office Use Only</b>	
Signature:	
Date:	
Penalty Applied (if applicable):	

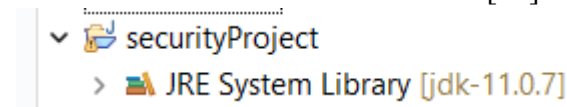
<b><u>Table of Contents</u></b>	<b><u>Pg</u></b>
Part 1	3
Part 2	7
Part 3	10
Part 4	14
Part 5	16
Part 6	22
Part 7	26

## Part 1

The two core elements of this application which are central to security paradigms which ought to be applied are Java and the MySQL database. These will be analyzed in depth in this section.

Java is used ubiquitously throughout the globe and is literally in billions of devices at this point in time. Java offers two different flavours with Java LTS which is no longer being maintained and thus has no updates or the more secure Java Non-LTS which has a release every 6 months. Opting for this version can help to stay up to date with the latest security features. [13]

Java's latest version is Java 20 which was released in March 2023 and this project however is created with JDK 11.0.7. [14]



Some high risk vulnerabilities in this version of Java include CVE-2020-14583, CVE-2022-21476, CVE-2023-21930. CVE-2020-14583 is given a 8.3 high risk by Snyk CVSS posing a high risk to confidentiality, integrity and availability. Successful attacks could lead to attackers gaining unauthenticated access to Java SE, with a variety of protocols and would affect this current application. It is however difficult to exploit and would require user interaction. [22] CVE-2022-21476 is an easy to exploit vulnerability with a high rating of 7.5 and low attack complexity. It can result in authorized access to sensitive data or complete compromise of the application. [23] Finally, CVE-2023-21930 could result in the unauthorized creation, modification or deletion of data vital to Java applications. [24] These vulnerabilities are just some of the vulnerabilities associated with that earlier version of java and show the importance of maintaining vigilance for updates. Running this project ought to be done with the latest version of java and all of the dependencies brought up to date.

Java has a whole host of security features directly imbued into the language and compilation process making it a popular choice for secure applications and can help in these efforts. Some of these include strict type checking, bytecode verification, compile time checking, not using pointers unlike other languages, cryptographic APIs with industry standard algorithms, public key infrastructure & secure communication in transmission. It has a plethora of libraries dedicated to security directly in its own portfolio and also other open source frameworks like Spring security which can be a powerful feature for setting up a secure application. This section will expand on some of these how they are an asset to this application and keeping things secure here. Beyond that there is inclusion of how the application could be made more secure with other Java features and libraries.

Java incorporates strict type checking which is an asset for this application. If an attacker was looking to add in some malicious types of characters that were not included in a String or a string when an int was expected, this would cause an error. This can help protect against injection attacks and arbitrary code execution as well limiting what is accepted for a given type.

Java contains a whole host of cryptographic packages to help maintain both confidentiality and integrity utilizing industry standard algorithms. They are also available natively in packages such as `java.security` and `javax.crypto`. These include AES, RSA & DSA signing algorithms and also the SHA family of algorithm for hashing and message integrity. [10] The open source libraries of Spring Security also offers more help in this domain with password encoders using algorithms such as `bcrypt` which can be applied here. This application direly lacks in mechanisms for both providing confidentiality and integrity for the data stored by the application. `Bcrypt` offers an effective way of helping to protect passwords at rest in this application instead of in plaintext as they are currently done. The latest version of Spring security is 6.0.3 as of writing and it's an open sourced library. [11] This is a powerful library that's been geared specifically towards some of the core aspects of security. By using these cryptographic algorithms we can help prevent data exposure, leakage or tampering.

Authorisation appears to be direly lacking this application and once a user is accessed they can do as they please on the given parts of the application they have direct access to. Authorisation checks ought to be performed on each action taken by a user. For example a user looking to delete a book in the application, it ought to be checked if they have the necessary permissions to do so. A way of doing this effectively is both through the user of RBAC (Role based access control) and having a centralizaed authorisation plane. For each action, a given user ought to be checked whether they have permission or capabilities to perform that action. A given user ought not to be able to access the books of another user for example. Spring security in java can help here providing a means to setup role based access control and an authorisation plane which can be passed through for every action a given user is looking to take. [12] This can prevent unauthorized activity occurring in the application and malicious users performing actions they ought to be able to do.

Effective use of access modifiers is an important aspect of applying java security mechanisms in this application. Java has four access modifiers, default, public, private & protected. Public affords accessibility from anywhere. Protected accessible within the same package or subclasses in different packages. Private means only accessible in the given class in which the class, method or variable is declared. Default is accessible only in the same package. These can be applied to instance variables, methods and classes. The use of public methods in the majority of the DAO classes is not restrictive enough and allows the methods to be accessible anywhere. These ought to be set to protected instead to limit the access to only within the package. The access modifiers for a database connection string and a username ought never to be public like they are in the DB class:

```
7 public class DB {  
8  
9     public static String user = "root";  
0     public static String connection = "jdbc:mysql://localhost:3306/library";  
1
```

They ought not even to be in the code either and this is expanded upon in a later section.

There doesn't appear to be the use of any input sanitization performed on any of the forms. The application is taking in untrusted data from the user and as a result of this is making itself susceptible to injection attacks. The use of libraries which contain regular expressions with classes in it's `java.util.regex` package such as `Pattern` and `Matcher` can help here. Through these classes the use of regular expressions can check

for an allowlist set of characters which have been defined by the developer. This can reduce the attack surface on these inputs and provides a means of rejecting all other input.

The Authentication of this application is poorly setup and insecure. It could be improved by offering an account lockout feature to protect against brute force attacks for at least one feature. Password complexity is not required for users, making it easier to perform dictionary attacks and brute force attacks. Passwords ought to be not stored in plaintext as well. Leveraging input validation with regexes, effective hashing algorithms and other secure features could help here. The remediations for these are expanded upon in part 5 with actual code. JAAS or Java Authentication and Authorization Service is also another tool that can help securing functionality in these regions as well.

There doesn't appear to be any session management at all. So a given user could theoretically stay logged in forever and then someone else if they were to use the same system could exploit this or purposefully get access to their system for exploitation. Sessions can help to restrict the time a given user is using an application without any actions before requiring them to re-authenticate. This challenge can stop attackers from gaining access to their session. Spring security can again help with that. Here's an example of some code that could be helpful:

```
http.csrf().disable()
.sessionManagement()
.sessionCreationPolicy(SessionCreationPolicy.ALWAYS)
.maximumSessions(1).and().sessionFixation().changeSessionId()
.and().logout().logoutRequestMatcher(new AntPathRequestMatcher(LOGOUT_URI))
.deleteCookies(CookieNames.JSESSIONID.toString()).invalidateHttpSession(true)
.clearAuthentication(true);
```

This is applied in a security configuration file and has session management, creation policy, limitations on number of sessions, mechanisms to protect against session fixation and then session invalidation on logout as well. On top of this session length ought to be checked in the code and taken into account for user actions and whether they would need to re-authenticate or not.

The application also lacks logging with sending all of its logs directly to the console which means they are not maintained. This can stop security incidents being tracked and is not capturing the necessary data to ascertain potential hacker operations and also information related to security incidents that could occur. It's a missed opportunity to be able to note information that can employ non-repudiation, and give time of event information. The application also has stack trace information being output which is bad practice for security, since it can help the attacker to launch further attacks. Some logging tools that have had some recent major vulnerabilities include the Log4j vulnerability which was a global issue and other logging tools like logback and other variants are probably a better option here without any major vulnerabilities like this.

Some other java features around security include features to combat against insecure deserialization, this was added in Java 17. It protects against arbitrary code execution that can occur through the use of serialized data obscuring the malicious code. [16] There are more features behind the scenes which help keep this application secure in Java. Java doesn't support the user of pointers which can be exploited to perform

unauthorized reads and writes of data. The JVM also performs checks for viruses and other malicious files that may have been inserted and this happens at the time of program execution. This is done with built in byte code verification. Compile time checking will go and prevent the accessing of a private method or variable while also not causing a denial of service by fully bringing the system down, instead it causes an error. The developer is not responsible for allocation and deallocation of memory, this garbage collection is performed automatically protecting the user from themselves and inadvertently creating memory related security vulnerabilities. [17]

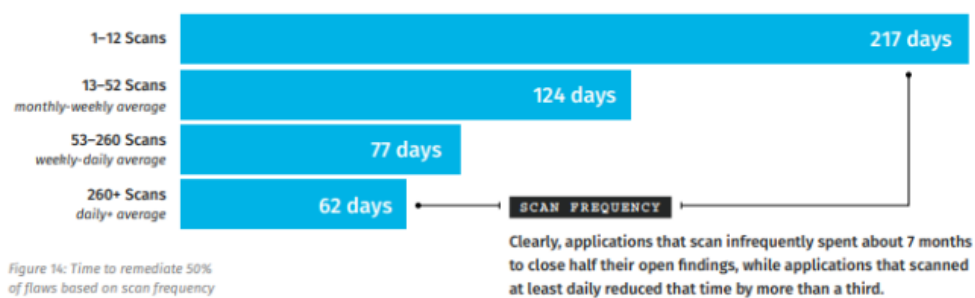
Another core component of this application is the MySQL database, we'll examine that next and it's security features.

### **MySQL**

The latest version of MySQL workbench is 8.0.33 [4]. It's a really good option for security offering a plethora of capabilities which have been a core part of the application from it's inception. MySQL offers multiple authentication methods including Old Password Authentication and Secure Password Authentication. [1] With Secure Password Authentication, it utilises SHA1 hashing to store passwords. It offers protection for data in transit with SSL/TLS support. In this application it's clear that it's not making use of the TLS support which could encrypt the traffic going to the database. It has a lot of tracking and auditing functionality for high level tracking of operations as well as on internal functioning. It's able to assess and identify tampering occurring within the database schema, whether that be on a particular table as well on sign and when authentication attempts are made. Firewall rules can also be set up as well dictating which traffic can be allowed in or out. There's a number of backup options which can be set for both offline and online as well resource limits can be placed on accounts and also account locking. In terms of authentication it gives at least two methods including Secure Password Authentication and Old Password Authentication. When adopting the Secure Password Authentication, it makes use of SHA1 to store passwords. The MySQL database as a selection affords both strong integrity and data reliability. Some of the drawbacks are that it requires vertical scaling which is a downside on availability. They can also be exploited by SQL injection attacks and knocked offline by DDoS attacks where attackers could go and usually make use of a botnet to bombard the database. They are also susceptible to race conditions when multiple actors are accessing resources at the same time and finally preauth user enumeration as well. These need to be addressed at the application level for SQL Injection and race conditions. [2] [3] Then with other tools for DDoS such as load balancing to distribute the traffic to lighten the load on individual MySQL databases. With all of this considered, MySQL is a widely used and strong option with a lot of security features to help protect our data and sustain the running of an application. Tailoring this library application towards it's many advantages by taking the time to properly configure the above mentioned features can help to improve the overall security of the application. Beyond that addressing the other concerns for things that go beyond the scope of MySQL like protections against SQL injection and race conditions ought to be taken care of in the confines of the DAO classes of the application.

## Part 2

Due to the sheer volume of code in modern applications this can be a very daunting and unfeasible task to be completely performed by manual inspection. These tools can be very useful at scale across large codebases, being applied in DevSecOps pipelines and across organizations. They can be placed to be run git hooks for a given push and play an important part of reducing the feedback loop for developers to quickly identify vulnerabilities and fix them with greater speed. This approach integrates into modern software development which heavily relies on Agile development which demands speed and continuous integration and continuous development require tools that can meet these demands and modern SAST and DAST tools are helpful here.



[15]

An interesting finding shows a high correlation between those who conduct more frequent scans close their open vulnerabilities findings much faster. For those that scan on a daily basis the time they rate at which they mitigated their vulnerabilities was reduced by a third over those that did it on a weekly basis. This highlights the value in incorporating scans directly into every push that goes to a repo in a CI/CD pipeline. It gives fast feedback that is visible to everyone for each push and this could help in sustaining it's presence in the minds of the developers ensuring that they go and actually fix them instead of potentially ignoring them. [15]

The two main categories of security testing falls into static and dynamic security testing. SAST tools operate on source code that is not in operation and is a form of white box testing, whereas, in contrast DAST look to effectively attack the system with payloads while running and see if they are able to identify vulnerabilities here. This is a black box method of security testing since it doesn't require knowing the inner workings of the actual code. Both of these tools can be a cost effective way of getting a sense of some security vulnerabilities in an application and can be particularly beneficial for organisations or teams with limited resources. These tools can help to detect these vulnerabilities and prevent potential breaches of the application and subsequent loss of revenue, reputation and or compromising of sensitive data.

Static Application Security Testing tools (also known as Source Code Analysis Tools) can help detect security vulnerabilities in source code. This is one of the first steps remediating those potential security vulnerabilities. An effective SAST tool can cover not just source code, but also configuration settings, dependencies used in the project and structural issues that could increase the likelihood of security issues. They have a

broad coverage going across entire codebases, they do have code visibility, they provide you with remediation advice and can be placed into the software development lifecycle easily. They are usually offering broad platform support & some can be easy to setup. 7 out of 10 open source libraries were found to have vulnerabilities in them and a number of Java applications can be up to 97% composed of third party code. This is drastically higher than many other languages, making this identification of dependency related vulnerabilities an important aspect of SAST tool selection with Java. [15] Beyond that, SAST are able to assert with a confidence level that they have found a specific vulnerability. This can help security engineers and developers alike to prioritize which vulnerabilities to assess first.

They can highlight well known vulnerabilities such as injection attacks like SQL injection and also Buffer overflow vulnerabilities and cross site scripting issues to name a few. The output provided by these tools can help to discern exactly where the vulnerability lies as well to speed up the process.

Some downsides of SAST tools include is that they can be prone to false positives which can result in time wasted by developers and high amounts of results which can create a false impression of vulnerabilities that don't actually exist. Other downsides include that modern SAST tools are only capable of identifying a subset of possible vulnerabilities and not enough to rely on for your complete security checking on your code. They may also struggle with checking some configuration issues,

Some important considerations for a given selection of a tool are whether it supports the programming language used on your project, the false positive and false negative rates given by the tool, can it run in your given IDE or pipeline for your specific use case, is it free or will your organisation be willing to pay the license fee, is it able to find the main types of security vulnerabilities you're concerned with and vulnerabilities included in the OWASP Top 10 for example. [18] Other considerations could include how much configuration is required, is it complicated to set up, is it still being maintained (popular SAST tools such as FindBugs are no longer maintained).

## **DAST**

Dynamic Application Security Testing is a type of black box testing that is applied while an application is running (at runtime) and is used to find security vulnerabilities. Since it's black box testing, the tools do not have direct access to the source code but instead perform attacks which are similar to actual attackers in the attack payloads. It applies system input and analyzes the outputs. We get the added benefit of seeing how the system reacts under scrutiny of different kinds of payloads in a live environment. In contrast to the SAST tooling, they tend to have a lower false positives rate and can show exactly how a given exploit was exploited. They do also offer information about given vulnerabilities that they discover. Again, since they are black box, they can offer broad platform support. They can be fairly language agnostic as a result. They can also be effective in regression testing to assess whether a fixed vulnerability is actually fixed. DAST tools are able to find these issues before they make it to production.

They can help to find some of the most prominent vulnerabilities in web applications in particular. These include external XML entities attacks, Cross site request forgery, SQL injection, cross site scripting as well. They can help to provide some vulnerability assessment reports to speed up the remediation process and can be integrated in




DevSecOps pipelines. They can comprehensively cover a broad amount of your APIs to give some level of assessment of your current security posture. They are more cost effective approach than having a pentester being set to perform a pentest for example. [21]

The downsides of DAST is based on the that black box approach, they don't give actual code insights, they're only checking the inputs and outputs which can make tracking down the actual area of the vulnerability in the code more time consuming for developers compared to SAST tools which can show you the direct line and class where they vulnerability is. Depending on where they payloads need to be deployed and how your application is structured, they can also make the testing process more slow. The deployment of payloads with going through various paths can need to be performed before their payloads can be deployed for each iteration. They are limited to only the APIs they can access which may limit their effectiveness. They don't have any real context of the application itself but instead are just firing payloads that have been specified. They're not able to detect all kinds of different vulnerabilities. [20]

Next we'll dive into the tools selected for testing this application.

### Spot bugs & Find Security Bugs



#### SpotBugs Eclipse plugin 3.1.5.r201806132012-cbbf0a5

SpotBugs is a defect detection tool for Java that uses static analysis to look for more than 400 bug patterns, such as null pointer dereferences, infinite... [more info](#)

by [SpotBugs Team](#), LGPL  
[java quality bugs analysis defects](#)

★ 902 🔄 Installs: **179K** (1,561 last month) Installed

The two tools selected for security testing were SpotBugs and Find Security Bugs. SpotBugs is the successor to FindBugs which had previously been used as a detection tool but subsequently stopped being maintained. As seen above in the Eclipse plugin version of this, it's a very popular plugin with 179K installs in Eclipse alone. It's current version is 4.7.3 and it can run with Java 1.8 and greater. On it's own, it's capable of finding 400 different bug patterns and it can be run as a standalone application or incorporated into maven, gradle or even as an eclipse plugin.

It categorizes based on rank and confidence. The rank aspect has a value of 1 to 20 with 1 to 4 being categorized as scariest, 5 to 9 being scary, 10 to 14 troubling and then finally, 15 to 20 of concern. The confidence level categories include 1 which is high confidence, 2 as normal confidence and 3 as low confidence. Using both of these scores together can serve to denote which potential vulnerabilities to prioritize first with investigations.

Another tool being used here is OWASP's Find Security Bugs. The latest version of this tool is 1.12.0. It's an open source tool that can be integrated into continuous integration pipelines with Jenkins or SonarQube. It has the capability to detect 141 different vulnerabilities and 823 unique API signatures. It supports Spring-MVC,

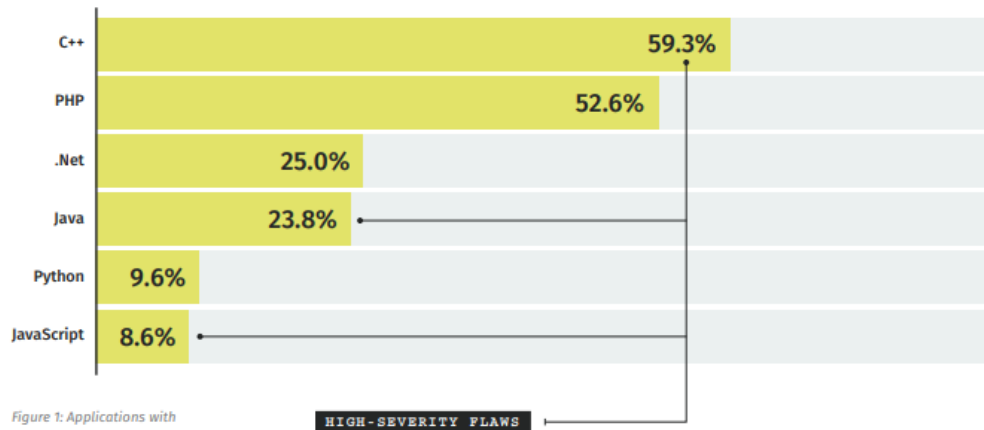
Struts, Tapestry to name a few frameworks. It's capable of being integrated into IDEs such as Netbeans, Eclipse & IntelliJ. It also has a CLI (command line interface) option as well. It gives extensive references for each vulnerability that it finds which include OWASP Top 10 information and CWEs. [19] This has the capacity to be combined with SpotBugs and extend it's capabilities to produce even more security rules for the code to be scrutinized with and that's how it's being implemented in this project.

On top of the two other tools being used SpotBugs and FindSecBugs, another tool considered was Snyk. Snyk could provide us with a number of features like secrets and credentials scans, scans on dependencies using it's own database of vulnerabilities, container scans and also source code scanning as well. It could have increased the spectrum of parts of the application being checked for vulnerabilities. Snyk is not able to run with this project due to the fact that the project does not contain a POM for Maven, or gradle. This was a recurring theme with trying out and searching for different tools. Effectively it, along with other tools like OWASP ZAP, the DAST tool are not able to run on this since it's not a web application. This project shows that there are a lot less security tools available for non-web applications that are also free. It made it more difficult and time consuming to find another one that would run since most of the other options like Checkmarx, Code Sight from Synopsys and others all required a commercial license which is not feasible for a college project. I tried to install and setup a whole bunch of them across multiple IDEs including Netbeans, Eclipse & Visual Studio Code but only two worked effectively. These two tools SpotBugs and FindSecBugs were as a result the main choice for the tooling automation for this project. DeepSource was also configured and the result of that explained later.

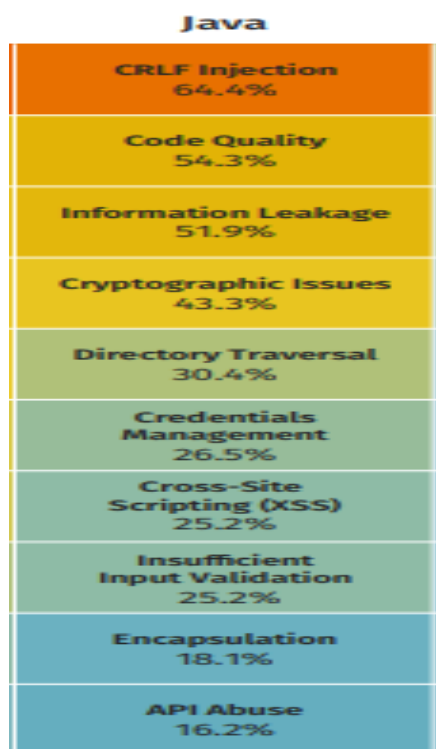
### **Part 3**

In order to conduct security testing a security roadmap ought to be put in place that guides where to begin the testing, where are high critical areas, what is the sensitive data we're trying to protect, how is it being stored. What regulations need to be considered like GDPR, PCI or HIPAA throughout the codebase and data. One aspect to conducting this security testing is identifying and isolating particular vulnerabilities that tend to be most regularly associated with Java applications. Knowing the specific vulnerabilities that are most likely to have a bigger prevalence and impact on Java applications can serve as a first port of call for investigations and considerations with this particular application.

According to a survey conducted by Veracode Java was found to have High severity flaws in 23.8% of applications that were involved in the study. This was considerably less than it's counterparts C++, php and slightly less than .Net at 25%



Some of the highest flaws faced were in CRLF Injection, information leakage, cryptographic issues and code quality as shown below.

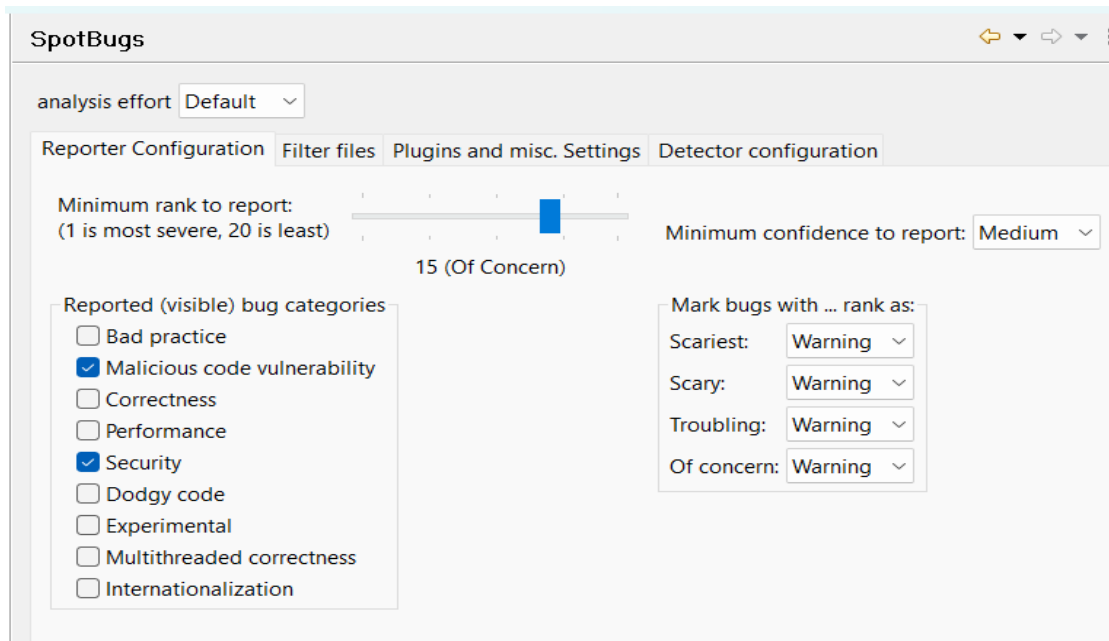


With this information in mind, it has helped in terms of designating considerations of where to spend increased levels of focus for reviewers, developers and architects alike and will serve as a road map for initial considerations for this Java application's security testing. However, in the case with CRLF injection, this is mostly more occurring in web applications, an example of which is with HTTP Response splitting which illustrates that point in contrast to this application without HTTP responses. The main security control for protection against that type of vulnerability in this application is input sanitization and allow lists of characters which we expand more on later but this is not a web application so will not be present here. Looking at these high priority areas associated with cryptographic issues, credential management, information leakage in any big Java application help to get a greater yield of results in terms of finding vulnerabilities. It narrows the scope in what can otherwise potentially be like looking for a needle in a haystack otherwise when the codebase is particularly large. Even in this more relatively smaller applications, starting with these kinds of areas is a good first step to be taken here.

Beyond this, other considerations that ought to be taken into account are core aspects of security applications which are common almost all applications. This includes: authentication, authorisation, session management, password management, complexity, reuse. Some questions worth posing: Is the application securing data in transit and at rest? Can authentication be bypassed or is it susceptible to brute force attacks, dictionary attacks, rainbow attacks? Are there credentials exposed in plaintext, codebases? Is there a mechanism for changing them regularly? Are they using secure cryptographic algorithms or are they using algorithms with known weaknesses? Are they rotating their keys on a regular basis? Are there protections in place to prevent privilege escalation? Are they using security principles like least privilege, defence in depth? How are the database connections setup and are they secure? Is it using a secure database option and making the most of the security available to that application? Is there adequate logging in place that provides non-repudiation and useful information for tracking potential security incidents? Is that logging information exposed or is it revealing sensitive information or information that an attacker could exploit like a stack trace? Is the logging library itself free of vulnerabilities or is subject to critical vulnerabilities like log4j? Are the open source dependencies up to date or they contain vulnerabilities that could be exploited? Is there protections in place against common vulnerabilities like SQL injection, cross site scripting, session hijacking and fixation, external xml entity attacks, cross site request forgery, buffer overflow? Is there potential memory allocation issues or potential concurrency issues that could result in availability concerns or the integrity of the data being compromised? Is it possible to perform denial of service directly via the application or through calls to the database that overwhelm either of them?

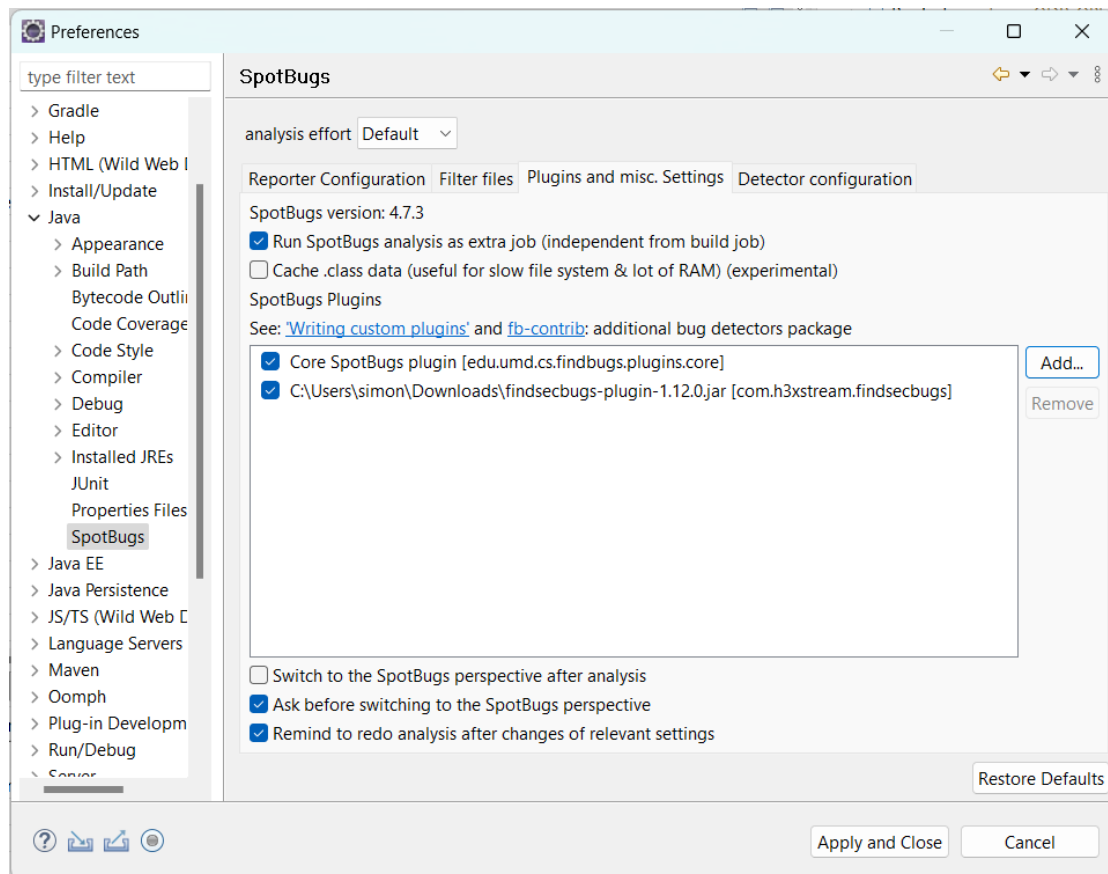
It's important to think like an attacker in these scenarios and look to see how you could potentially exploit the system. Within the answering of these questions in our testing we're able to identify prominent security gaps and potential vulnerabilities where attackers could be able to exploit the system or it's data. Mapping out the system and it's attack surface will play an important role in seeing where these vulnerabilities may lie. This includes API endpoints, containers, dependencies, infrastructure as code and much more. A more recent term of having an SBOM, software bill of materials which has a nested inventory of what makes up the software is increasingly more and more important. Many applications in particular in Java are largely compromised of open source libraries and their level of risk and vulnerabilities and version must be kept up to date and accessible. [25] As the security tester these dependencies could really serve as low hanging fruit to compromise the application with some having exploits readily available online by simply searching the dependency and it's version. Maintaining a software patching strategy that keeps these dependencies up to date is increasingly more a necessity for these applications even if it can require a level of tedium and pull developers away from other tasks. It's needed to keep the application as secure as it can be. All of this is taken into consideration throughout the security testing being conducted.

Applying automation is going to be an important part of any security testing being carried out for all of the reasons previously mentioned. Below are some of the settings being used to configure Spotbugs and Find Security Bugs.



Some SAST tools can be prone to false positives so the confidence required has been set to Medium in order to help eliminate unnecessary findings that are not likely to be legitimate vulnerabilities. The bug patterns and categories that will be applied to the code will be from the Security and Malicious Code Vulnerability. Those two are relevant for finding security vulnerabilities. It's necessary to not bloat the results with non-relevant findings that won't be congruent with the effort to find security vulnerabilities.

Augmenting the capabilities of Spot bugs with find-sec-bugs from OWASP. That's included in here through external plugins:



Again, as mentioned, this provides a whole host of increased additional attack patterns and security bugs being investigated, increasing the spectrum of possible vulnerabilities that could be found here with this additional tool. It also provides it's own advice for dealing with these vulnerabilities as well. This will be helpful for the stage of fixing those very vulnerabilities.

## **Part 4:**

This section will show the output of the two testing tools which were able to obtain results. This is the results of Spotbugs and Find Security Bugs analysed and discussed.

### **Spot bugs & Find Security Bugs output**

FindBugsSummary	
timestamp	Sat, 22 Apr 2023 22:01:29 +0100
total_classes	130
referenced_classes	450
total_bugs	127
total_size	3981
num_packages	1
java_version	19
vm_version	19+36-2238
cpu_seconds	13.95
clock_seconds	11.55
peak_mbytes	1376.06
alloc_mbytes	2048.00
gc_seconds	0.39
priority_2	119
priority_1	8

Spot bugs was able to identify a potential 127 issues within the code. 8 of these reached the TROUBLING and SCARY threshold for risk on security vulnerabilities.

securityProject (8)	
Scary (5)	
High confidence (3)	
Potential JDBC Injection (3)	
This use of java/sql/Statement.executeQuery(Ljava/lang/String;)Ljava/sql/ResultSet; can be vulnerable to SQL injection (with JDBC) [Scary(5), High confidence]	
This use of java/sql/Statement.executeQuery(Ljava/lang/String;)Ljava/sql/ResultSet; can be vulnerable to SQL injection (with JDBC) [Scary(5), High confidence]	
This use of java/sql/Statement.executeQuery(Ljava/lang/String;)Ljava/sql/ResultSet; can be vulnerable to SQL injection (with JDBC) [Scary(5), High confidence]	
Normal confidence (2)	
Hard coded password (1)	
Information Exposure Through An Error Message (1)	
Troubling (3)	
High confidence (3)	
Nonconstant string passed to execute or addBatch method on an SQL statement (3)	

One of these posited vulnerabilities was correctly identified to be a security vulnerability, having a hard coded password in the codebase:

BugInstance	
type	HARD_CODE_PASSWORD
priority	2
rank	7
abbrev	SECHCP
category	SECURITY
first	1
Class	
classname	mainlibrary.DB
SourceLine	
Method	
classname	mainlibrary.DB
name	getConnection

```

26         props.put("user", user);
27         Hard coded password found [Scary(7), Normal confidence] password here";
28         props.put("useUnicode", "true");

```

This is a legit vulnerability and could give an attacker the password as well as username for accessing the database used for the application. This would result in data exposure, GDPR issues, potential tampering as well as opportunities to perform Denial of Service by dropping tables or deleting the database itself.

Another was potentially releasing sensitive information:

```

55     } catch (Exception e) {
56         e.printStackTrace(System.out);
57     }

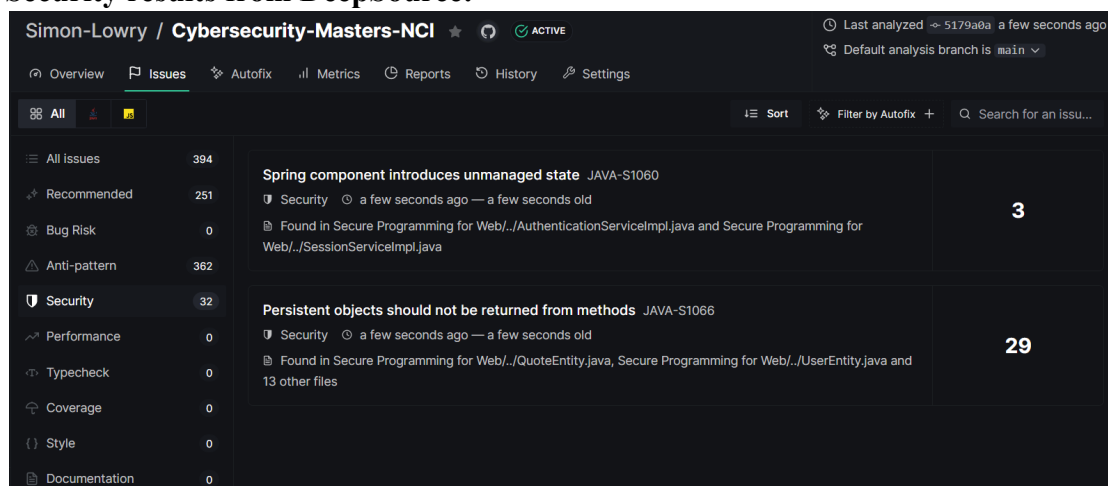
```

```
55 } catch (Exception e) {  
56     Possible information exposure through an error message [Scary(7), Normal confidence]  
57 }
```

This was a line of code printing a stack trace when an any exception occurred in the execution of a SQL query. This was in the AllStudents class and method with the same name. It was deemed a Scary vulnerability risk with normal confidence. Having checked it out, this is again a legitimate vulnerability and risk. Stack trace should not be printed to the console or any logging. This can otherwise help the attacker with developing further exploits.

Another potential vulnerability was where the code passes a nonconstant String to an execute on an SQL statement. These were deemed to be in the Troubling category. It also revealed information about some SQL injection vulnerabilities. These were already found through the manual code review process. The manual code review was conducted first and is outlined in the next section.

## Security results from DeepSource:



As noted previously, using DeepSource required running this on an entire repo which actually has multiple other projects not related to this current project as well. This required some siphoning through the mentioned security vulnerability results to assess whether any were relevant to the current project.

However, it appears as though the tool was unable to find any meaningful results related to the project at hand, only finding results for another project. The full reasons for this was not clear and due to time constraints was not fully able to investigated. It may again be due to not being a web application like some others and not containing a pom file or gradle setup.

## Part 5: Manual Code Review

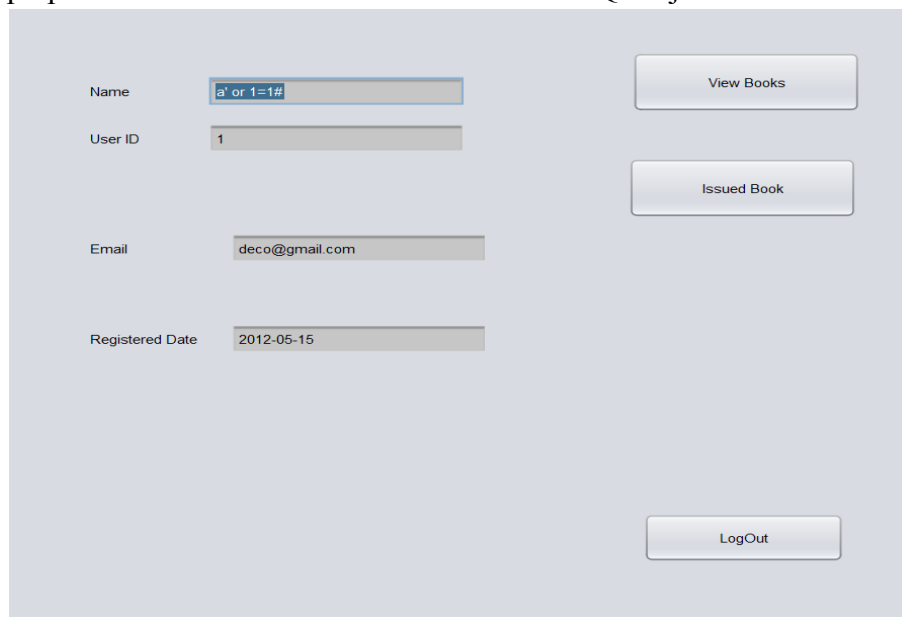
The manual source code review has focused on two classes in particular and these were UsersDao and LibrarianDao. These two are central to some of the core functionality of the application. They are at an important juncture placing queries to the MySQL database and the security in their operations is paramount to protecting the data and availability of the system.



### Manual secure code review, class 1: UsersDao

One of the classes selected for manual code review was UsersDao. This is performing the database operations for the Users table in the database. Its operations include three methods, Validate(), CheckIfAlready() and AddUser().

The Validate() method is used in relation to the user login. After assessing its code through manual review, there is no prepared statements being used for the database operation which is a select statement looking to check if a certain user with a provided username and password exists. This is used to authenticate the user. The lack of a prepared statement makes this vulnerable to SQL injection:

A screenshot of a web application interface. On the left, there are four input fields: 'Name' with the value 'a' or 1=1#', 'User ID' with the value '1', 'Email' with the value 'deco@gmail.com', and 'Registered Date' with the value '2012-05-15'. On the right, there are three buttons: 'View Books', 'Issued Book', and 'LogOut'.

This was confirmed by performing a SQL injection attack against the system. Here the payload entered was:

a' or 1=1#

as shown in the above image.

This was executed by the following lines of code:

```
public class UsersDao {  
  
    public static boolean validate(String name, String password) {  
        boolean status = false;  
        try {  
            Connection con = DB.getConnection();  
            String select = "select * from Users where UserName= '" + name + "' and UserPass='"+ password +"'";  
            Statement selectStatement = con.createStatement();  
            ResultSet rs = selectStatement.executeQuery(sql:select);  
        }  
    }  
}
```

The database connection is established, the SQL statement is then created and executed. There is input sanitization on the untrusted data provided by the users and there is no prepared statements to separate the execution of the SQL query from the untrusted data. As a result of this, the attacker is able to bypass both the authentication and authorisation required to be able to gain access to the library system. They have accessed the user's account with the userId of 1 and username of Deco. Here they could perform whatever nefarious activity they so choose on that given user's account.

We're also able to get the full status of every book in the library breaching authorisation checks after exploiting the SQL injection vulnerability:

Name	Genre	Author	Publisher	Shelf	Row	Available
3 Harry Potter and Goblet of Fire	Fiction	J. K. Rowling	Pottermore	B	12	Not Issued
4 Harry Potter and Deathly Hallow	Fiction	J. K. Rowling	Pottermore	23	D	Not Issued
5 Famous Five	sd	ds	dsd	3	A	Not Issued
6 akjkd	hdfjd	ksjdgh	hghth	d	h	Issued
13 The da Vinci Code	Thriller	Dan Brown	Doubleday	5	r	Not Issued
14 Pride and Prejudice	Romantic	Alexander Dumas	Pearson	9	a	Not Issued
15 To Kill A Mocking Bird	Emotional	Harper Lee	McGraw	8	z	Not Issued
16 The Perks Of being A Wallflower	Drama	Stephen Chbosky	Klett	1	g	Not Issued
17 The Hunger Games	Action	Suzanne Collins	Pearson	7	t	Not Issued
18 Madhusala	Life	H R Bacchan	Pushpalata	6	h	Not Issued
19 V for Vendetta	Action	Alan Moore	Cambridge	9	a	Not Issued

Logging and exception handling of all three methods is also subpar and is only outputting all exceptions to the console. E.g.

```
catch (Exception e) {

    System.out.println(e);

}
```

This is a catch all without specific conditions that could mitigate different scenarios of exceptions differently. It also provides no information to be utilized about the individual carrying out the action making it lack in auditability, non-repudiation and tracking for security incidents. While this is not a security vulnerability per se, it's poor secure coding practices and ought to be remedied.

There is no use of any access modifiers. The access could be restricted to being protected to prevent being used outside of the constraints of the package.

The Connection, String (with the SQL command) and the Statement ought to be set to be Final and constants. This prevents any alteration from the time they have been setup beyonds to prevent any tampering or loss of integrity for the SQL command or established database connection.

One positive practice in place is that they are closing the database connections directly after individual use preventing any unintended or malicious use of a left open database connection. However, in the event of an exception, the database connection is not being closed.

```
ResultSet rs = selectStatement.executeQuery(select);
status = rs.next();
con.close();
} catch (Exception e) {
    System.out.println(e);
}
```

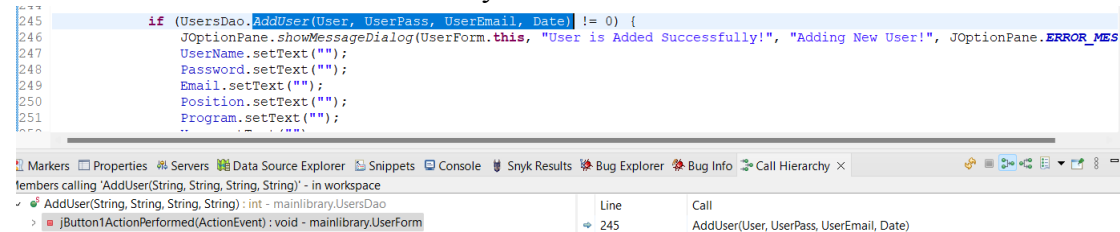
So for example, if an exception occurs at selectStatement.executeQuery(select) this would leave the database connection. This is not good security practice for the protection of nefarious usages of that database connection.

Another concerning aspect of this class is this commented out exception:

```
public static int AddUser(String User, String UserPass, String UserEmail, String Date) {
    //throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templat
```

It's a commented out exception for UnsupportedOperationException to show that this functionality is not supported yet. This ought to be cleared up with the developers in a real world situation. Are we having code accessible that does not have proper authorisation and checks in place? There should not be commented out lines like that

in a production application and it raises doubts about the legitimacy of whether this code ought to be in codebase at this time or whether it's properly been set up and doesn't lead to an unintended security incident.



The breadcrumb above reveals it's usage and it ought to be discussed with the developers.

Another SQL injection vulnerability is present in the method CheckIfAlready(). Again it's similar to the previous example, where untrusted input in the form of the username is being fed into a SQL query which is not a parameterized query.

```
public static boolean CheckIfAlready(String UserName) {
    boolean status = false;
    try {
        Connection con = DB.getConnection();
        String select = "select * from Users where UserName= '" + UserName + "'";
        Statement selectStatement = con.createStatement();
        ResultSet rs = selectStatement.executeQuery(select);
        status = rs.next();
        con.close();
    } catch (Exception e) {
        System.out.println(e);
    }
    return status;
}
```

There's the same lack of closing the connection in a *finally* block instead of inside the try block itself to ensure it's closed regardless of what happens. The exception is generic and could be tailored better to handle different scenarios and reveal only the necessary information to logs instead of a console output. The lack of constants for the String that's the SQL query and the other objects that are not mutable would also be a concern.

Overall, the most pressing vulnerability in this class was the SQL injection. SQL injection vulnerabilities currently fall under Injection Attacks in the OWASP Top 10 and are placed at number 3. It's a heavily prominent vulnerability with 94% of the applications they tested presenting it. [5] There are other lower risk vulnerabilities in there as well and they ought to be remediated when time allows.

### **Manual Secure Code Review, Class 2: LibrarianDao**

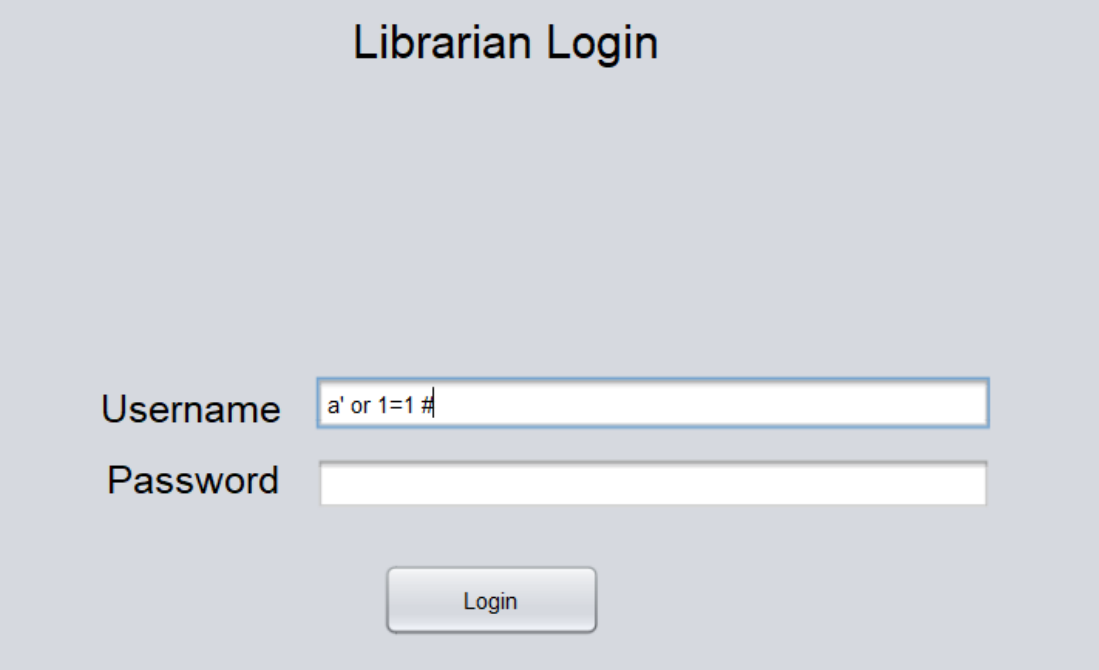
The second class that has been investigated with manual source code review is LibrarianDao. It contains three methods, save, delete and validate. The method validate is used to authenticate librarians in the librarian login form.

```
DB.java x BookDao.java x LibrarianDao.java x TransBookDao.java x UsersDao.java x UserForm.java x LibrarianLogin.java x
History
    status = ps.executeUpdate();
    con.close();
} catch (Exception e) {
    System.out.println(x: e);
}
return status;
}

public static boolean validate(String name, String password) {
    boolean status = false;
    try {
        Connection con = DB.getConnection();
        String select = "select * from Librarian where UserName= '" + name + "' and Password='" + password + "'";
        Statement selectStatement = con.createStatement();
        ResultSet rs = selectStatement.executeQuery(sql:select);

        status = rs.next();
        con.close();
    } catch (Exception e) {
        System.out.println(x: e);
    }
    return status;
}
```

Again, the validate method contains a SQL injection vulnerability. It's not using prepared statements and thus untrusted user data can be injected directly into the SQL query being executed. In this case the following was executed:



The screenshot shows a web application titled "Librarian Login". It features two input fields: "Username" and "Password". The "Username" field contains the text "a' or 1=1 #". Below the fields is a "Login" button. The background is a light gray.

Payload is again the same due to the structure of the SQL statement and it executes a SQL injection attack to validate that the vulnerability exists.

Name

Library ID

Email

Contact No.

From this the attacker effectively has admin access to the system and could delete books, or identify and enumerate students for further attacks. They have access to all of their account data including the passwords which are stored in plaintext.

Users

☒ Name 
 ☐ Email 
 ☐ ALL

User ID	UserPass	RegDate	UserName	Email
1	1234	2012-05-15	Deco	deco@gmail.com
2	12345678	2016-11-07	Mark	markide4@gmail.net
3	012345	2016-11-25	NewHero	newhero@gmail.com

This is an issue with exposes a wide ranging security issues covering authentication, authorisation, lack of encryption on sensitive data. This could also be an issue of PII and personally identifiable information exposure and could potentially result in GDPR fines for the library system owner. The integrity of the data for the books could be compromised and manipulated by the attacker as well.

This is an example of Identification and Authentication failure and also Injection Attacks in regards to the OWASP Top 10. These are respectively placed at number 3 and 7 in the current list. The data has not been validated, or sanitized and there's no prepared statements in use, breaching the authentication security features in place. Cryptographic failures is currently number in the OWASP Top 10 and is centered around the protection of data in transit and also at rest. It looks for specific protection for sensitive data such as passwords. These ought to be encrypted with industry standard algorithms to protect them. [6]

## **Part 6**

### **1. SQL injection vulnerability -> UsersDAO, method: validate**

For the SQL injection vulnerabilities highlighted previous in UsersDAO, in order to fix this vulnerability, instead of having this code:

```
String select = "select * from Users where UserName= '" +  
name + "' and UserPass='"+ password + "'";
```

```
Statement selectStatement = con.createStatement();
```

Use prepared statements instead like the following code outlines:

```
PreparedStatement ps = con.prepareStatement("select *  
from Users where UserName=? and UserPass=?");
```

```
ps.setString(1, name);
```

```
ps.setString(2, password);
```

This separates the execution of the sql statement from the untrusted data provided by the user. Thus, this neutralizes any would be SQL injection payloads, rendering attacks here not possible for SQL injection.

On top of that, the username could be sanitized first like this:

```
Pattern pattern = Pattern.compile("(?=.*[A-Za-z-',.  
]).{1,20}$");
```

```
Matcher matcher = pattern.matcher(password);  
boolean isValidUserName= matcher.find();
```

```
if(isInvalidUserName)  
    throw new IllegalArgumentException("Invalid  
username.");
```

Here if the username doesn't match a given regex which describes the acceptable values in an allowlist of characters, an exception is thrown rejecting the user data.

The passwords ought to be also hashed with a hashing algorithm like bcrypt. This is further expanded upon next.

### **3. Cryptographic failure, user passwords not being hashed in database and accessible to staff of library.**

Passwords ought to be stored in hashed format in the database. Applying bcrypt for password hashing is due to it's strength relative to other hashing options. It's stronger than PBKDF2, as well as argon2 and scrypt for memory requirements that are less than 4MB which is the case here, making it a solid option. [8]

For this application it is possible for Librarians to be able to see directly the user's passwords.No-one should have access to a given user's password. Whether they are

staff of the library or not, this is a breach of privacy and can lead to insider attacks and data leaks.

```
import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
...

public String generatePasswordWithBCrypt (String
plainPassword) {

    String encodedPassword =
bcryptEncoder.encode (plainPassword);

    return encodedPassword;

}

public boolean isExpectedPassword (String enteredPassword,
String encodedPassword) {

    return bcryptEncoder.matches (enteredPassword,
encodedPassword);

}
```

generatePasswordWithBCrypt simply encodes the String password provided by the user with the BCrypt encoding algorithm, Then isExpectedPassword can be used to determine whether the used provided password for logging in matches the bcrypt encoded password which would be stored in the database.

#### **4. Cryptographic failure, database credentials exposed in code**

As mentioned previously, there is the database credentials in the class called DB. With the database credentials in the codebase, anyone with access to the code would be able to access the database, or even in the event that it's compiled code, this can also be reverse engineered by attackers to obtain the credentials. A better approach here would be to make use of a secrets manager to store the credentials. These are specifically designed for storing sensitive information like credentials and do so with encryption, credential rotation and access control in place.

[7]

#### **5. Cryptographic failure, lack of password complexity.**





PASSWORD LENGTH	POSSIBLE COMBINATIONS	TIME TO CRACK	
		S = SECONDS M = MINUTES	H = HOURS Y = YEARS
4	45697	<1	S
5	1 188 1376	<1	S
6	308915776	<1	S
7	8031810176	~4	S
8	208827064576	~1.5	M
9	5429503678976	~45	M
10	1 411 677 095 653 376	~19	H
11	3670344486987780	~.1	Y
*12	95428956661682200	~1.5	Y
13	248115287320374E4	~39.3	Y
14	645099747032972E5	~1,022.8	Y
15	167725934228573E7	~26,592.8	Y
16	436087428994289E8	~691,412.1	Y
17	113382731538515E10	~17,976,714	Y
18	2947951020001390E10	~467,394,568	Y

This can help protect individual users from having the passwords easily guessed, or cracked by brute force or dictionary attacks with some mediation against that.

## 5. Information Disclosure through error message

Exceptions ought to be handled gracefully and not reveal any potential sensitive information. This case was demonstrated that the stack trace was being output. This should be removed to not disclose any information that could be used by an attacker to launch an attack. Instead a more tailored message could be tracked in the logs for developers and/or information that does not help an attacker be output to the screen.

## 6. passes a nonconstant String to a SQL execution and lack of other constant use

Throughout the various classes that execute SQL queries like LibrarianDAO, UsersDAO, TransBookDAO there is a pervasive lack of using constants for the SQL query being executed and the objects being constructed to implement these queries. For the Strings and objects in use like below:

```
try {
    Connection con = DB.getConnection();
    String select = "select * from Users where UserName= '" + name + "' and UserPass='"+ password +"'";
    Statement selectStatement = con.createStatement();
    ResultSet rs = selectStatement.executeQuery(select);
```

these ought to be set to final and as a constants to protect against intentional or unintended tampering of queries being made and results obtained.

e.g

```
final String select = "select ..." // shortened for simplicity
```

```
final Statement selectStatement = con.createStatement();
```

```
final ResultSet rs = selectStatement.executeQuery();
```

Other additional security improvements could be account lockout to protect against brute force attacks and dictionary attacks, sessions being applied to increase the protections of user sessions as well.

## **Part 7 - Conclusion**

Leveraging the state of the art research and testing this was able to identify a number security vulnerabilities. While the security tools were effective at identifying important security vulnerabilities, we're able to see value in the manual source code review augmenting this process. Utilizing the research of Veracode which showed the main vulnerabilities found for Java served as a good road map for areas to focus on and proved to be accurate here. Some of the core vulnerabilities that were identified in the Veracode research common to Java turned up in this application. This included multiple cryptographic issues (also found in 43.3% of applications by Veracode), credential management vulnerabilities (found in 26.5% of applications), information leakage (found in 51.9%), and insufficient input validation (found 25.2%) were all also present in this application. We can see from this the value of gaining insight into specific vulnerabilities that tend to be more common in a given programming language and interweaving that with state of the art tools & methodologies can yield effective results. The same applies for applying an attacker's mindset and different methodologies to try and uncover as many vulnerabilities possible. Focusing on the most likely areas of vulnerabilities in Java was proven to be a helpful starting point for finding vulnerabilities.

We can see both the value of these different automated technologies and the disadvantages at play in this application. They helped to cover a wide breadth of classes that wouldn't have been possible to fully assess in the given timeframe and they also help to identify legitimate risks but also the downside was that some were missed and some were false positives. Having them incorporated in ways that cause the least amount of friction with current software processes like Agile and producing business results on functionality, increases the likelihood of their adoption and adherence to. This is where their use in DevSecOps pipelines with Continuous Integration and Continuous Deployment matters and increases the likelihood of bugs both being found and being fixed by teams.

It's clear also that a spread of different approaches including manual code inspection and code review has its place in going beyond the current tooling as well. Manual inspectors have the advantage of a greater understanding of the wider context of each integrated pieces and potentially knowledge which help uncover more vulnerabilities and disprove false positives as well. Security testing and other security approaches are necessary throughout the full software development life cycle to gain as much impact as possible. Starting from a shift left mindset with aspects of design including threat modeling and going all the way through also security testing integrated into DevSecOps, manual inspection on code reviews, increase the likelihood of finding security vulnerabilities. Applying them in a way which suits the business to get the most out of the resources they have available to them is needed. No one aspect here on its own is going to give maximal results but together with a defence in depth approach, they can increase the likelihood of finding vulnerabilities and protecting applications against potentially costly breaches. Thus saving organisations on revenue, reputational damage, customer trust and increasingly costly fines like those from GDPR.

### **References:**

- [1] Poojary K., (2011). *How Hackers Use Trusted Users For Their Exploits*, [Online] Available:  
<https://www.computerweekly.com/tip/CSRF-attack-How-hackers-use-trusted-users-for-theirexploits>
- [2] Shahriar H., (2017). *Security Vulnerabilities of No SQL and SQL Databases*, International Journal of Digital Society (IJDS), Volume 8, Issue 1, March 2017, [Online] Available:  
<https://infonomics-society.org/wp-content/uploads/ijds/published-papers/volume-8-2017/Security-Vulnerabilities-of-NoSQL-and-SQL-Databases-for-MOOC-Applications.pdf>
- [3] Satori Cyber, *MySQL Security Common Threats and Best Practices*[Online] Available:  
<https://satoricyber.com/mysql-security/mysql-security-common-threats-and-8-best-practices/>
- [4] MySQL. *MySQL Downloads*, [Online] Available:  
<https://dev.mysql.com/downloads/mysql/>
- [5] OWASP. (2021). *A03:2021 Injection* [Online] Available:  
[https://owasp.org/Top10/A03\\_2021-Injection/](https://owasp.org/Top10/A03_2021-Injection/)
- [6] OWASP. (2021). *A02:2021 Cryptographic Failures*, [Online] Available:  
[https://owasp.org/Top10/A02\\_2021-Cryptographic\\_Failures/](https://owasp.org/Top10/A02_2021-Cryptographic_Failures/)
- [7] Jog S., (2020). [Online] Available:  
<https://www.linkedin.com/pulse/storing-database-credentials-securely-siddhesh-jog/>
- [8] NIST. (2002). *Secure Hash Standard*, [Online] Available:  
<https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf>
- [9] Oracle. *Pattern*, [Online] Available:  
<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>
- [10] Oracle, *Java Security Overview* [Online] Available:  
<https://docs.oracle.com/javase/9/security/java-security-overview1.htm>
- [11] Spring Security, *Spring Security Repo*, [Online] Available:  
<https://github.com/spring-projects/spring-security>
- [12] Parashiv E. (2023). [Online] Available:  
<https://www.baeldung.com/role-and-privilege-for-spring-security-registration>
- [13] Snyk, *Java Security Explained*, [Online] Available:  
<https://snyk.io/learn/java-security/>
- [14] CodeJava. (2023). *Java SE Version History*, [Online] Available:

<https://www.codejava.net/java-se/java-se-versions-history#:~:text=The%20latest%20version%20of%20Java,years%20of%20support%20from%20Oracle>).

[15] Veracode. (2020). *Volume 11: State of Software Security*, [Online] Available: <https://www.veracode.com/sites/default/files/pdf/resources/sossreports/state-of-software-security-volume-11-veracode-report.pdf>

[16] Vermeer B., (2020). *Serialization and Deserialization in Java*, [Online] Available: <https://snyk.io/blog/serialization-and-deserialization-in-java/>

[17] JavaTPoint, *Why Java is Secure*, [Online] Available: <https://www.javatpoint.com/why-java-is-secure>

[18] OWASP. *Source Code Analysis Tool*, [Online] Available: [https://owasp.org/www-community/Source\\_Code\\_Analysis\\_Tools](https://owasp.org/www-community/Source_Code_Analysis_Tools)

[19] Find Security Bugs. (2022). *Find Security Bugs*, [Online] Available: <https://find-sec-bugs.github.io/>

[20] Bashvitz G., (2023). *What is Dynamic Application Security Testing*, [Online] Available: <https://brightsec.com/blog/dast-dynamic-application-security-testing/>

[21] Snyk. *Dynamic Application Security Testing*, [Online] Available: <https://snyk.io/learn/application-security/dast-dynamic-application-security-testing/>

[22] Snyk. (2020). *CVE-2020-14583*, [Online] Available: <https://security.snyk.io/vuln/SNYK-DEBIAN10-OPENJDK11-584592>

[23] Snyk. (2023). *CVE-2022-21476*, [Online] Available: <https://security.snyk.io/vuln/SNYK-DEBIAN10-OPENJDK11-2770093>

[24] Snyk. (2022). *CVE-2023-21930*, [Online] Available: <https://security.snyk.io/vuln/SNYK-DEBIAN10-OPENJDK11-5430887>

[25] CISA. (2018). *Software Bill of Materials (SBOM)*, [Online] Available: <https://www.cisa.gov/sbom#:~:text=A%20%E2%80%9Csoftware%20bill%20of%20materials,that%20make%20up%20software%20components>.