Bucharest Academy of Economic Studies

Faculty of Cybernetics, Statistics and Economic Informatics

IT&C Security Master

Creation of a security system for an online library

Scientific coordinator: Alin ZAMFIROIU

Graduate: Stan Sebastian Daniel

Bucharest

2023

Contents

1. Introduction 3
2. Presentation of informatic system 5
2.1 Use Case Diagram 5
2.2 Activity Diagrams 5
2.2.1 Log In Activity in Mobile app 5
2.2.2 Book Download in Mobile app 6
2.3 Deployment Diagram 7
2.4 Communication Diagram 7
2.5 Class Diagram 8
2.6 Database Diagram 9
3. Tehnology stack 9
3.1 Android 9
3.2 Android Studio 10
3.3 Angular 11
3.4 Node.js 12
3.5 Express 13
3.6 Firebase 9
4. Aplication Components 15
4.1 Mobile Application Component 16
4.2 Web application Component 17
4.3 Server Component 17
5. Importance of Security Standards 17
5.1 Mobile Application security standards OSWASP 17
5.2 Web application Security Standard of OWASP 20
6. Implementation of security standards 23
6.1 Mobile Application 23
6.2 Web Application 34
Conclusion 44
Table of Figures 45
Bibliography 46

1. Introduction

The development of secure systems for online libraries has become increasingly important in recent years. This is due to the growth of electronic payment systems [3], identity theft [4], and the need to secure sensitive data in online databases [5]. In this context, NoSQL databases have become popular alternatives to traditional SQL databases [6]. However, they are not immune to attacks, and NoSQL injection vulnerabilities have been discovered in various systems [8]. As such, it is important to consider the security of the database management system when designing a secure online library.

Mobile devices are often used to access online libraries, and securing them is crucial. Android is one of the most widely used mobile operating systems and has been the subject of various security studies [17]. Studies have proposed different techniques to secure mobile applications, including reducing the attack surface by automatically securing permission-based software, source code obfuscation [12], and biometric authentication [18].

Microservices architecture is increasingly used to build scalable and resilient systems. However, it presents new security challenges, such as distributed denial-of-service attacks, communication security, and container security [14]. Various approaches have been proposed to secure microservices, including role-based access control, encryption, and identity management [16].

The rapid increase in online libraries has led to a rise in the need for secure online library systems. This dissertation aims to develop a security system for an online library, to ensure the protection of sensitive information and prevent unauthorized access to resources.

The objective of my dissertation is the case study of the process of securing an online multi-platform library. The library is composed of 2 major part the web app part in which different publishers can publish their books and manage their accounts and a phone app where users can purchase and read the books. The security system will be designed to include several components, including user authentication, access control, encryption, and intrusion detection. The

authentication process will be implemented using multi-factor authentication techniques, such as biometrics or two-factor authentication, to ensure that only authorized users can access the system.

Access control will be implemented using role-based access control (RBAC) to restrict access to sensitive information to only authorized users, users will be divided in 2 groups readers and publishers, readers being able to only allowed to read and purchase the books, write reviews, download the books after the purchase, while publishers can do everything readers do and publish books. Encryption will be used to protect data stored on the system from unauthorized access, while intrusion detection will be implemented to detect and respond to any attempted security breaches.

The rapid development of online libraries has increased the need for effective security measures to protect the sensitive data of users. Mobile devices are becoming an essential tool for accessing online libraries, and secure mobile devices have become an important topic of research [3]. Mayrhofer proposes an architecture for secure mobile devices that includes a hardware security module and an operating system designed for secure booting, which could potentially improve the security of mobile devices used for accessing online libraries.

In the first part we will talk about the presentation of the informatic system by use of diagrams, we will talk about a few data and action flows found in the application

In the second part we will talk about the technology stack, that being some of the software and programing languages used in the development of the thesis solution.

The third segment is a general discussion about the components of the application and the security standards used to check if the application is safe.

In the final part the topic of discussion is describing the ways one can use or that I used to protect against the security threats presented in the previous chapter

2. Presentation of informatic system

2.1 Use Case Diagram

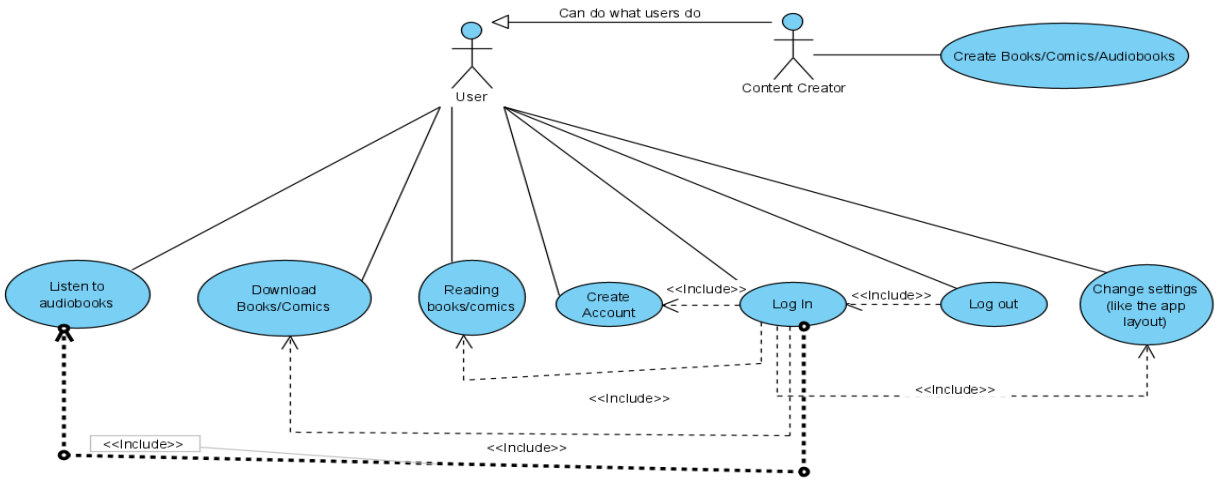

Figure 1 *use case diagram*

In the basic part of the application there will be 2 types of users, normal users that can buy the created books and can read, listen and download said books and Content creators that can create books for users to buy, they will receive a share of the revenue generated by their created collection.

2.2 Activity Diagrams

2.2.1 Log In Activity in Mobile app

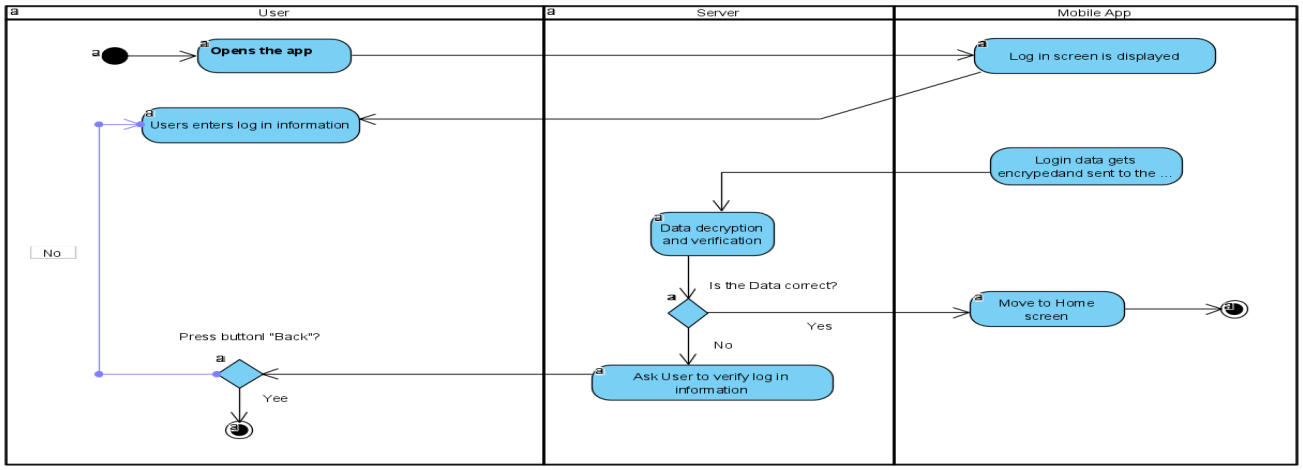

Figure 2 *Log in Activity Diagram*

The above process shows the Activity diagram for a user logging into the mobile application. First the user opens the app where the users will be greeted by the log in screen where he can enter the log in information that will be encrypted and sent to the server in order to confirm the user's identity and authenticate the user into the application, once the data has been received by the server and it has been decrypted and confirmed correct the mobile app will receive a response to move the user to the home screen of the application. If the information sent by the user is not correct he will receive a message to verify the entered log in details and try again. If the users presses the "Back" button while in the log in screen while not in any text box used to enter information it will take him out of the application.

2.2.2 Book Download in Mobile app

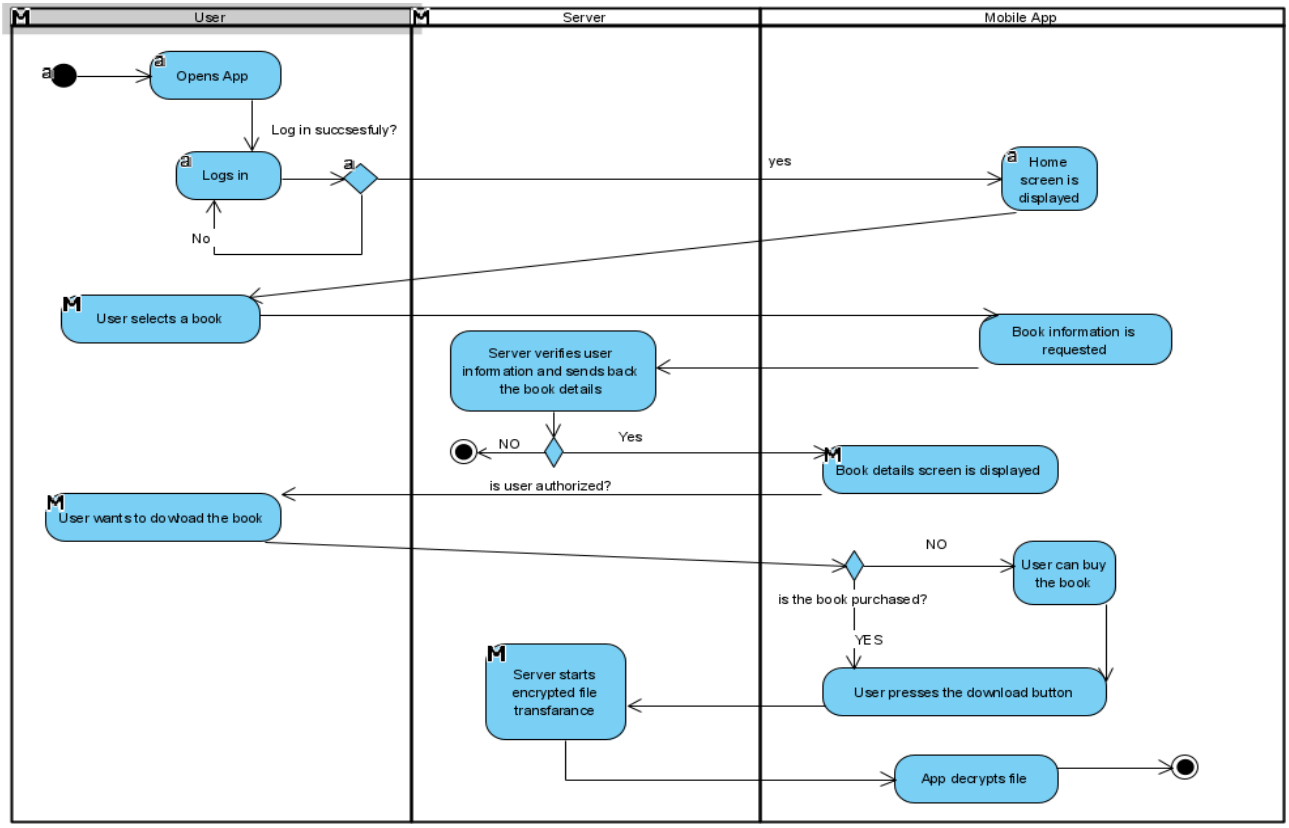

Figure 3 *Download book Activity Diagram*

The activity diagram starts by the user initiating the log in activity described in the point above, after the user can browse the home screen where a variety of books are displayed or the user can select a book from another section in the app such as favorite books and select a book he wishes to see the details of , here a request to the server will be sent where the server will verify the user

information and authentication token to make sure the user can access such information and to defend the server and app against session hijacking, Replay Attacks, CSRF (Cross-Site Request Forgery) Attacks. After that the user sees the book details and can select to download the chosen book it the user has purchased the book if not the user can purchase the book and then download it. Once the user presses the download button the process of sending the file begins, the file first is encrypted by the server and then sent to the application where the application will decrypt it so that the user can view the selected downloaded book.

2.3 Deployment Diagram

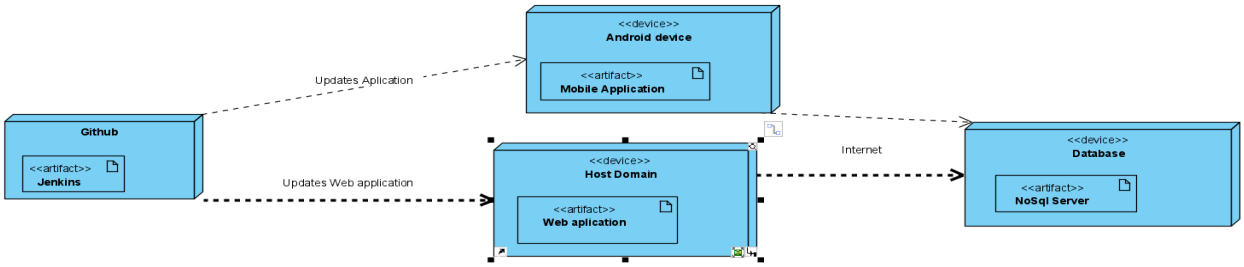

Figure 4 *Deployment Diagram*

The code of the application will be on github and will use a pipe in order to deploy new versions of the apps. The apps connect through the internet to the database where the server is running.

2.4 Communication Diagram

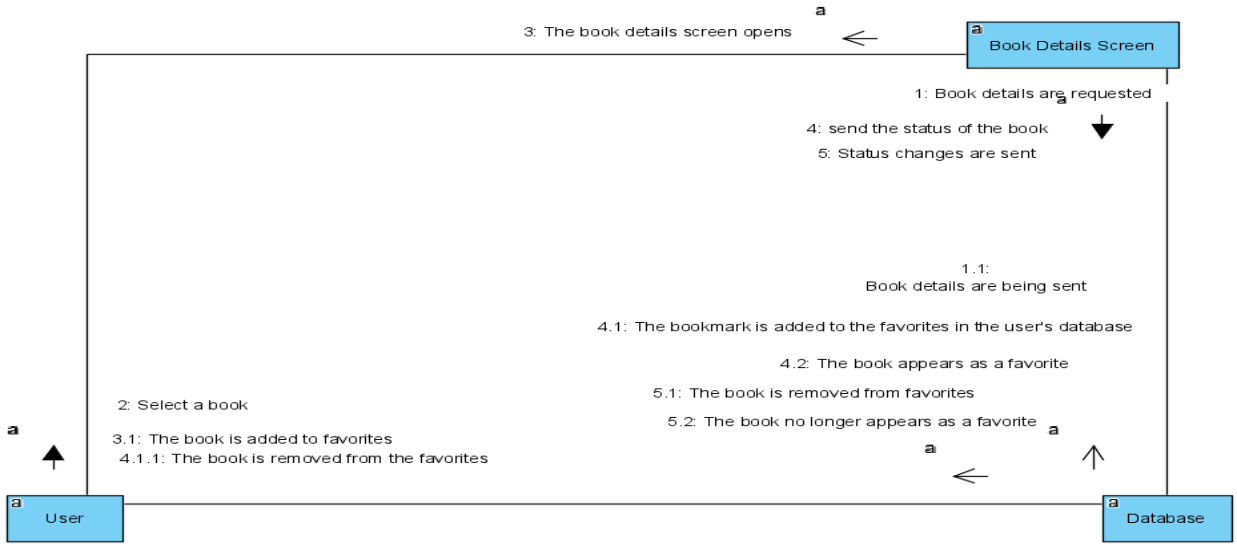

Figure 5 *Communication Diagram*

In the communication diagram we can see the process of adding a book to the favorite area in the mobile application , here first the users enters the details screen of the and request the details of the book, the server sends the details from the database back to the app and the screen opens , the user adds the book to the favorites section through the icon found in the user interface and the mobile app sends a status change post to the database , if the user presses the favorite button again the it sends a request to the database for another status update in the database and the book is removed form the favorite section

2.5 Class Diagram

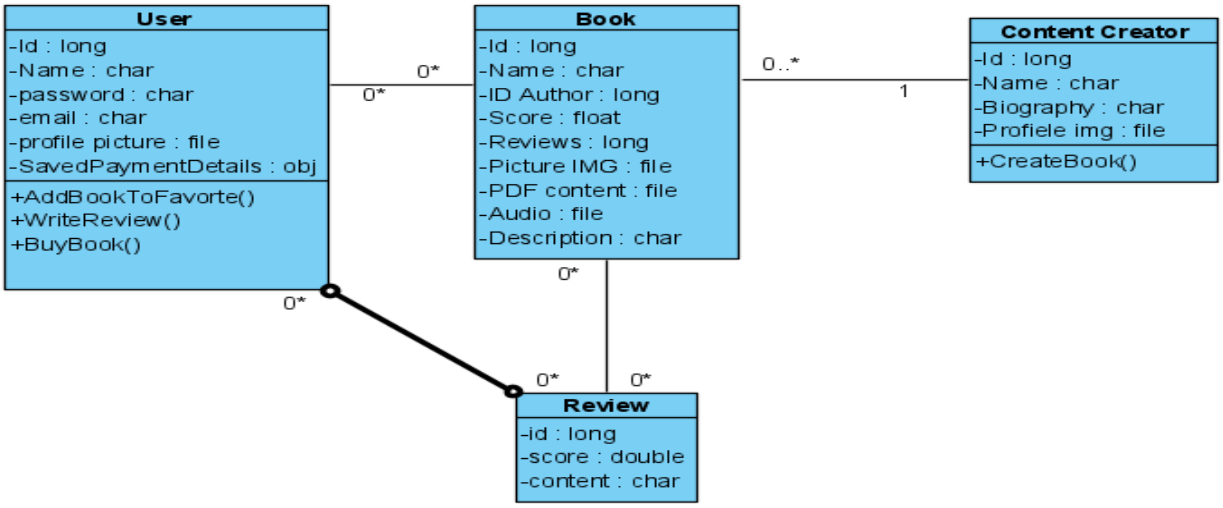

Figure 6 *Class diagram*

In the above diagram there is shown the association of the 4 most important classes in the application, those being User, Book, Content Creator, and Review. As shown by the diagram a User can have zero or as many books bought as it pleases, and can write as many reviews as well. A book can be associated with many or zero review or users but all books have at one Content Creator.

2.6 Database Diagram

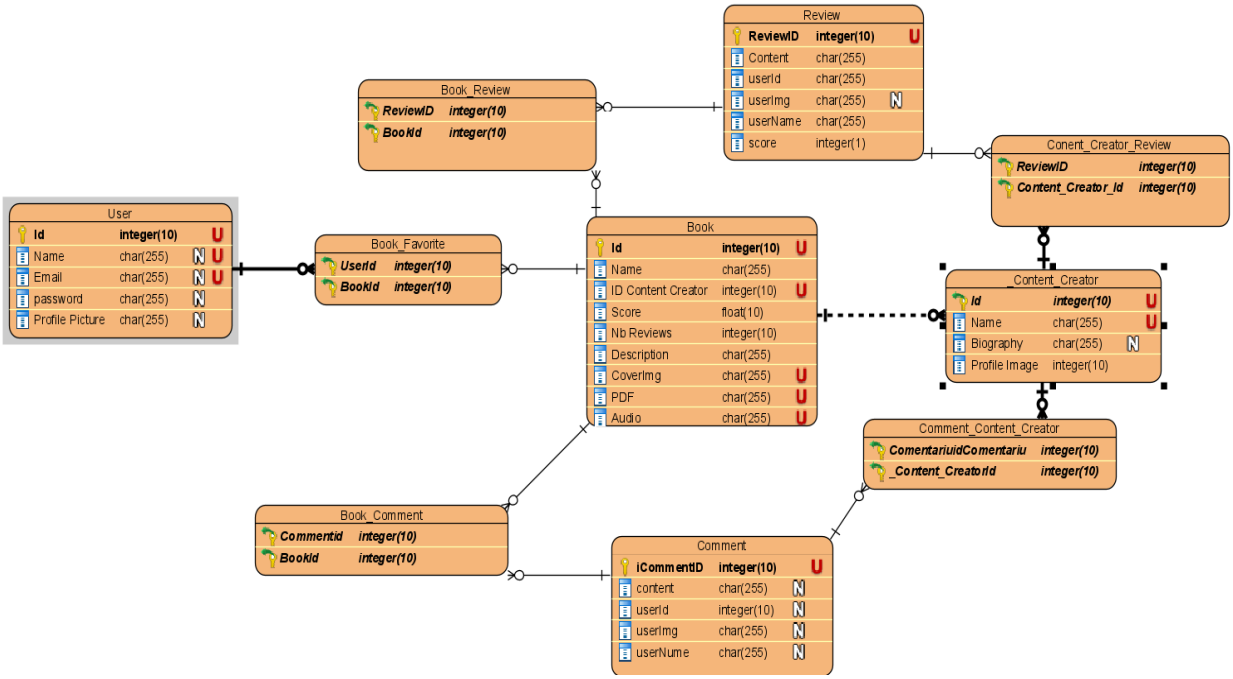

***Figure** 7 Database Diagram*

In the above diagram its is displayed the Database Diagram and the relations of each of its components. For example User that has an id that is a primary key so it is unique, an Name , email fields which should be unique too, and a password which cant be null as are the fields mentioned above, a Profile picture which is represented by a location on the server.

3. Technology stack

3.1 Android

Since its debut in September 2008, Android has swiftly become the leading mobile operating system. Its open-source, Linux-based nature allows for continual innovation and modification, making it a popular choice among developers and consumers alike.

Android's possibilities are astounding, with complete APIs and tools available to developers. Because of its sophisticated graphics rendering, multi-threading capability, and efficient memory management, it is possible to create aesthetically attractive and high-

performance programs. The integration with Google services improves functionality, and the Google Play Store provides a global marketplace for software distribution.

There are various advantages to developing for Android. Because it is open-source, it supports a collaborative developer community and provides substantial resources. Android Studio and documentation make development easier, and the Google Play Store provides a large user base for app distribution. Android's adaptability and market reach make it an excellent choice for developers looking to create creative and popular mobile applications.

3.2 Android Studio

Android Studio, which is based on JetBrains' Intelij IDEA software, has a long history that began with its unveiling on May 16, 2013 at the "Google/IO" conference. Google's annual conference in Mountain View, California, acts as a venue for the introduction of revolutionary technologies. Android Studio rose to prominence swiftly, displacing the outmoded "Eclipse Android Development Tools" (E-ADT) and establishing itself as the go-to IDE for developing native Android applications. Its first stable version was launched in December 2014, marking an important step forward in its development. Android Studio has a number of notable features.

Android Studio offers a user-friendly and intuitive interface that aids in the efficient creation, testing, and debugging of Android applications. With its seamless integration with the Android Software Development Kit (SDK) and the availability of powerful tools like the Layout Editor and Virtual Device Manager, Android Studio streamlines the development process and enhances productivity. Moreover, it provides comprehensive support for various programming languages, such as Java and Kotlin, enabling developers to choose their preferred language.

However, it is important to highlight an unusual aspect of the program: the absence of an option to disable the autosave protocol. Alongside this, the IDE offers [1]:

- GUI drag-and-drop functionality for intuitive development.
- Android-specific tools for code refactoring and quick fixes.
- Integration and signing of applications using "ProGuard."
- A powerful layout editor facilitating drag-and-drop placement of UI components, as well as previewing layouts across multiple screen configurations
- A highly adaptable Gradle-based build system.

- An emulator with extensive capabilities and swift compilation speed.
- A unified development environment capable of producing applications for various Android systems.
- Seamless integration with GitHub and code mocking, simplifying the design of common features and the importation of code mocks.
- Advanced testing tools and frameworks.
- A virtual Android Device emulator for running and debugging applications developed in Android Studio.

3.3 Angular

Angular is a popular JavaScript framework developed and maintained by Google. It was first introduced in 2010 as AngularJS, and later revamped and rebranded as Angular in 2016. Over the years, Angular has become a powerful tool for building dynamic and robust web applications. Its rich history and continuous development have contributed to its widespread usage and adoption within the development community. Angular's primary usage lies in creating single-page applications (SPAs) and web applications with complex requirements. It provides developers with a structured and scalable approach to building applications, following the Model-View-Controller (MVC) architectural pattern. By separating the concerns of data, presentation, and application logic, Angular allows for easier code maintenance, reusability, and testability. One of Angular's key advantages is its extensive set of features and capabilities. It offers a comprehensive set of tools for building dynamic user interfaces, managing data binding, handling form validation, and implementing client-side routing. Angular also includes a powerful dependency injection system, facilitating the management and sharing of components and services across an application.

Advantages of Angular:

- Robust framework: Angular was selected as the framework of choice for this web development project due to its robustness and maturity. With Google as its maintainer and a thriving community of developers, Angular provided a solid foundation for building large-scale web applications.
- Comprehensive feature set: Angular's extensive range of built-in features and capabilities played a pivotal role in the decision-making process. The framework offers powerful tools for data binding, dependency injection, form validation, routing, and more. By leveraging

these features, it was possible to streamline development and reduce reliance on external libraries.

- Modularity and reusability: A significant advantage of Angular that influenced the decision was its emphasis on modularity and component-based development. By utilizing Angular's component architecture, it was possible to build a highly modular application, resulting in cleaner code, improved maintainability, and accelerated development.
- Two-way data binding: Angular's two-way data binding feature was a critical factor in choosing the framework. It enabled seamless synchronization between the application state and the user interface, significantly reducing the need for manual DOM manipulation and enhancing the overall development experience.
- Type safety with TypeScript: The decision to adopt Angular was reinforced by its integration with TypeScript, a statically typed superset of JavaScript. TypeScript's type safety, enhanced tooling, and improved developer productivity provided a strong foundation for writing maintainable code and catching errors at compile-time.
- Testability: Angular's comprehensive testing framework, supporting unit testing, integration testing, and end-to-end testing, contributed to its selection. The ability to write robust test suites facilitated the identification of issues early in the development process, ensuring the reliability and stability of the application.
- Performance optimizations: Angular's performance optimization techniques, such as Ahead-of-Time (AOT) compilation, lazy loading, and tree shaking, were key factors in choosing the framework. These optimizations resulted in smaller bundle sizes, faster rendering, and overall improved performance, enhancing the user experience.

In conclusion, Angular has a rich history as a powerful JavaScript framework used for building dynamic web applications. Its usage and adoption have grown steadily over the years, thanks to its comprehensive set of features, modularity, and strong community support. Angular's advantages lie in its structured approach to development, extensive capabilities, testability, and performance optimizations.

3.4 Node.js

I chose Node.js because of its long history, versatility, and tremendous features. Ryan Dahl introduced Node.js in 2009, and it has since grown in popularity and widespread usage among

developers. It is based on the Chrome V8 JavaScript engine, which allows it to run JavaScript code outside of the browser. This discovery opened up new avenues for server-side and backend development, making Node.js an excellent alternative for developing scalable and efficient web applications.

Node.js is widely used in a variety of domains, particularly web development. Its non-blocking, event-driven architecture allows it to handle a large number of concurrent connections efficiently. Node.js is ideal for real-time applications, streaming services, and APIs that demand fast response times. Furthermore, its package manager, npm, provides a wide ecosystem of reusable modules and libraries, allowing developers to build apps quickly by reusing existing solutions.

Node.js provides various benefits to my research project. Its lightweight and scalable design allows me to optimize resource utilization and efficiently manage a high number of concurrent requests. The event-driven, non-blocking architecture improves the application's responsiveness and scalability, making it suited for real-time updates or handling a large number of concurrent connections. Furthermore, the large Node.js community and its dynamic ecosystem provide access to a variety of tools, tutorials, and community support, ensuring I have the assistance I require throughout the development process.

The capabilities of Node.js reinforce my decision to use it in my project. It enables me to develop my application's server-side and client-side components in a single language—JavaScript. This speeds up development and allows for seamless connection and data sharing between the server and the client. Furthermore, Node.js excels at managing I/O tasks like file system interactions and network requests, making it ideal for developing high-performance apps.

3.5 Express

Because of its long history, extensive use, and robust capabilities, I chose Express, a popular Node.js web application framework. Express was first launched in 2010 and has since grown to become one of the most popular frameworks in the Node.js ecosystem. It offers a simple yet strong collection of functionality for developing web apps, making it an excellent choice for my project.

Express's features confirm my decision to employ it in my dissertation project. It provides a lightweight and unbiased approach to web development, laying the groundwork for customization and flexibility. To increase the functionality of my application, Express allows me to easily interact

with numerous templating engines, databases, and middleware. Furthermore, Express has strong routing capabilities, allowing me to specify different endpoints and efficiently handle HTTP methods.

Express is frequently used to develop server-side apps and APIs. Its ease of use and adaptability make it appropriate for a wide range of applications. Express helps me to establish routes, handle requests and replies, and build middleware to boost functionality and preserve code cleanliness, whether I'm developing a small-scale application or a large-scale API.

3.6 Firebase

Figure 8 *Suit of Firebase services*

The figure above shows the suite of services that a developer who wants to work with Firebase can call upon.

The first suite of functionality expressed in the column on the left of figure 8

shows functionality made to build better apps:

- ✓ Auth — login and user identity
- ✓ Realtime Database — real-time, cloud-hosted, NoSQL database
- ✓ Cloud Firestore — Realtime, cloud hosted, NoSQL database
- ✓ Cloud Storage — highly scalable file storage
- ✓ Cloud Functions — "serverless", event-driven backend
- ✓ Firebase Hosting — Global Web Hosting
- ✓ ML KIT —SDK for common ML tasks

Firebase Authentication: is a complete user authentication system that makes safe login and registration functionality in web and mobile applications easier to build. It supports a variety of authentication methods, such as email and password, phone number, and social network accounts. To improve the security of user accounts, developers can use Firebase Authentication's built-in capabilities such as email verification and password reset. It also enables multi-factor authentication, which adds an extra degree of security. Developers can easily handle user authentication with Firebase Authentication, allowing users to safely access their accounts and offering personalized experiences within the application.

Cloud Firestore: Firebase's Cloud Firestore is a highly scalable and adaptable NoSQL document database. It provides real-time data synchronization, which implies that any data changes are promptly disseminated to all linked devices. This allows for real-time collaboration and ensures that users always get the most recent information. Offline functionality in Cloud Firestore allows programs to continue running even when the device is not connected to the internet, effortlessly synchronizing data once the connection is restored. It also includes a robust querying framework that allows developers to quickly extract specified data subsets. Developers may use Cloud Firestore to create responsive and collaborative applications that expand easily as user demand grows.

Cloud Storage: Another important Firebase service is Cloud Storage, which enables dependable and scalable object storage for online and mobile apps. It enables developers to store and serve user-generated material such as photographs, videos, and files in a safe manner. The straightforward API of Cloud Storage allows for simple file uploading, downloading, and sharing. It also provides access controls and at-rest encryption to protect data security. Cloud Storage works smoothly with other Firebase services, allowing developers to take advantage of its capabilities in conjunction with other Firebase components. Cloud Storage provides a stable storage solution with the scalability to address expanding storage needs, whether it's hosting static assets or managing user-generated content.

4. Application Components

As time flows and technology advances so does the ways to store and enjoy humanity’s greatest joy stories, and due to human to the rapid advancements of the last centuries has gotten us the methods of enjoying stories have transformed as well, until the general population of most

western countries started benefiting from internet the most common way to obtain a book was going out physically search for it and buy it, then in order to store it you just deposited it somewhere in your home, an alternative to that was going to libraries and enjoying their stock of books but that too was limited as depending on the size of the library the collection of books found in it could vary wildly. As most people nowadays know having physical books can be both a blessing and a bother. They are nice to display and many people enjoy more actually holding a book while reading having a physical book presents many problems one of the biggest ones is storage and preservation, as a physical item it takes space and you need to be careful around it as you can damage it easily , another big problem is availability as previously stated your chance to find the book you desire can vary drastically depending upon its popularity , if you are not near the place you store or borrow books from you need to carry the book with you something that increases drastically the danger to the item and generally limits your access to it.

So having such problems when the opportunity to store them digitally and distribute it through the use of the internet was available many software solutions flooded the market in order to resolve this conundrum. And so do I throw my coin into the pond in order to satisfy the many people's desire to have their favorite book easily and readily accessible to them, that is why the solution I bring is both on the web for desktops and laptops and on mobile in order cast as wide a net as possible

4.1 Mobile Application Component

The incipient part of my thesis project is the mobile application as it was the subject of my end bachelor's degree thesis subject. The application offers the users the ability to download , read online , and listen to a wide variety of books and even comic books but if I wanted to actually publish it I need to find ways to not only to keep the application optimized so that it will not slow down the user's phone but also be as accessible as possible so that many users can use it meaning that I should use the lowest version of mobile operating system as possible in order to let it be usable on as many phones as possible. One major problem that appears when you want to appease the lager masses is that if the application is readily available it will malicious eyes upon it.

So in order to protect the application I had to redesign parts of it in order to make it more secure, to do that I look at the top 10 OWASP Risk and tried to resolve them

4.2 Web application Component

The web application is similar to the mobile variant of the application with one key difference here users with Content creator accounts will be able to use the web variant to publish their works by uploading their books in pdf variant trough a page in the site, then said pdf will be sent to the server signed and encrypted with a key created upon the account's creation. Due to it being an web application it shares some similar variabilities with the Mobile version. In order to ensure the maximum degree of protections I followed the OSWAP Top 10 vulnerabilities

4.3 Server Component

The final part is represented by the server onto which the vast majority of data resides inside a Firebase database accessible by the server through secure microservices that connect to the applications

5. Importance of Security Standards

As the subject of my dissertation is the process of securing of an online library the method by which I will verify it the quality of my solution is by looking at what are the general security standards and if I apply them correctly.

5.1 Mobile Application security standards OSWASP

For the Mobile Application I followed the OWASP top 10 Mobile Security risk.

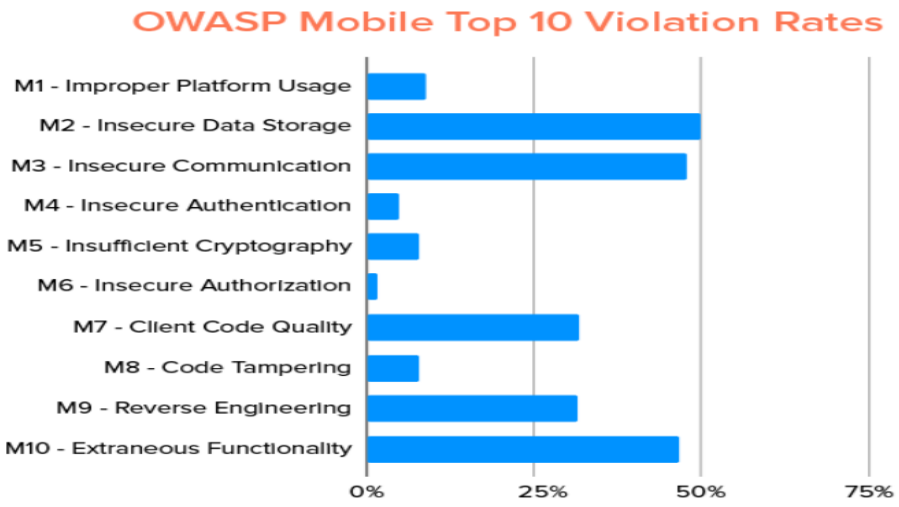

Figure 9 *OWASP top 10 mobile violations [21]*

1. **Improper Platform Usage:** This violation occurs when mobile applications do not adhere to platform-specific security guidelines and best practices. Each mobile platform (such as iOS or Android) has its own security requirements, and failure to follow them can lead to vulnerabilities. For example, using insecure storage methods or insecure inter-process communication can result in unauthorized data access or privilege escalation. Properly understanding and implementing platform-specific security measures is essential to mitigate this risk.
2. **Insecure Data Storage:** Mobile applications often store sensitive data locally, such as passwords, authentication tokens, or personal information. Insecure data storage vulnerabilities arise when this sensitive data is stored in an unencrypted or easily accessible format. Attackers can exploit this vulnerability by gaining unauthorized access to the device or extracting sensitive information. Proper encryption, secure key management, and secure data storage practices should be implemented to protect sensitive data.
3. **Insecure Communication:** This violation pertains to insecure communication channels used by mobile applications to transmit data. Mobile applications that do not use secure communication protocols (such as SSL/TLS) are susceptible to interception, eavesdropping, or man-in-the-middle attacks. Attackers can exploit insecure communication to steal sensitive information or tamper with data in transit. Mobile applications should implement secure communication protocols and adhere to encryption best practices to protect the integrity and confidentiality of transmitted data.
4. **Insecure Authentication:** Insecure authentication vulnerabilities occur when mobile applications have weak or inadequate authentication mechanisms. This can include improper session management, weak password policies, or lack of multi-factor authentication. Attackers can exploit these vulnerabilities to bypass authentication, gain unauthorized access to user accounts, or perform session hijacking attacks. Implementing secure authentication measures, such as strong password policies, secure session management, and multi-factor authentication, is crucial to prevent unauthorized access.
5. **Insufficient Cryptography**: Mobile applications often rely on cryptography for various purposes, such as protecting sensitive data, securing communication, or implementing digital rights management. Insufficient cryptography violations occur when weak or outdated cryptographic algorithms or improper implementation practices are used.

Attackers can exploit weak cryptography to decrypt sensitive data, tamper with encrypted information, or bypass security controls. Mobile applications should use strong cryptographic algorithms, properly implement encryption and decryption processes, and follow established cryptographic best practices.

6. **Insecure Authorization:** Insecure authorization vulnerabilities arise when mobile applications fail to properly enforce authorization checks, leading to unauthorized access to privileged functionalities or data. Common issues include insufficient privilege validation, insecure access control logic, or insecurely stored access tokens. Attackers can exploit insecure authorization to escalate privileges, access unauthorized functionalities, or compromise sensitive data. Implementing secure authorization mechanisms, such as role-based access control (RBAC), granular permission controls, and proper access control checks, is essential to prevent unauthorized access.
7. **Client Code Quality:** The quality of the client-side code in mobile applications plays a crucial role in security. Poorly written or insecure client-side code can introduce vulnerabilities such as buffer overflows, input validation flaws, or insecure data storage. Attackers can exploit these vulnerabilities to execute arbitrary code, manipulate data, or compromise the application. Mobile developers should follow secure coding practices, perform code reviews, and use secure coding frameworks and libraries to mitigate these risks.
8. **Code Tampering:** Code tampering vulnerabilities occur when mobile applications do not implement sufficient protections to prevent unauthorized modification of the application's code or resources. Attackers can tamper with the application's code to introduce malicious functionality, bypass security controls, or extract sensitive information. Implementing code integrity checks, applying code obfuscation techniques, and utilizing runtime application self-protection (RASP) mechanisms can help prevent code tampering.
9. **Reverse Engineering:** Reverse engineering vulnerabilities occur when mobile applications are not adequately protected against reverse engineering techniques. Attackers can reverse engineer the application to understand its inner workings, extract sensitive information, or identify vulnerabilities for exploitation. Implementing code obfuscation, using anti-reverse engineering techniques, and applying binary protection mechanisms can help deter reverse engineering and protect the application's intellectual property.

10. **Extraneous Functionality:** Extraneous functionality refers to the presence of unnecessary or unused features or components within mobile applications. These features may introduce additional attack surfaces or increase the application's complexity, leading to vulnerabilities. It is essential to regularly review and remove any unnecessary functionality, third-party libraries, or debug code that could potentially introduce security risks. By minimizing the application's attack surface, organizations can reduce the likelihood of successful attacks.

Addressing the OWASP Top 10 Mobile violations is crucial to ensure the security and integrity of mobile applications. Organizations and developers should prioritize secure coding practices, stay updated on platform-specific security guidelines, and regularly assess and mitigate these vulnerabilities to protect user data, prevent unauthorized access, and maintain user trust in mobile applications.

5.2 Web application Security Standard of OWASP

For the Web application I chose to follow the OWASP (Open Web Application Security Project) Top Ten that is a list of the most critical web application security vulnerabilities. These vulnerabilities represent the most prevalent and impactful security risks faced by web applications. Here is a short synopsis of the OWASP Top Ten web vulnerabilities:

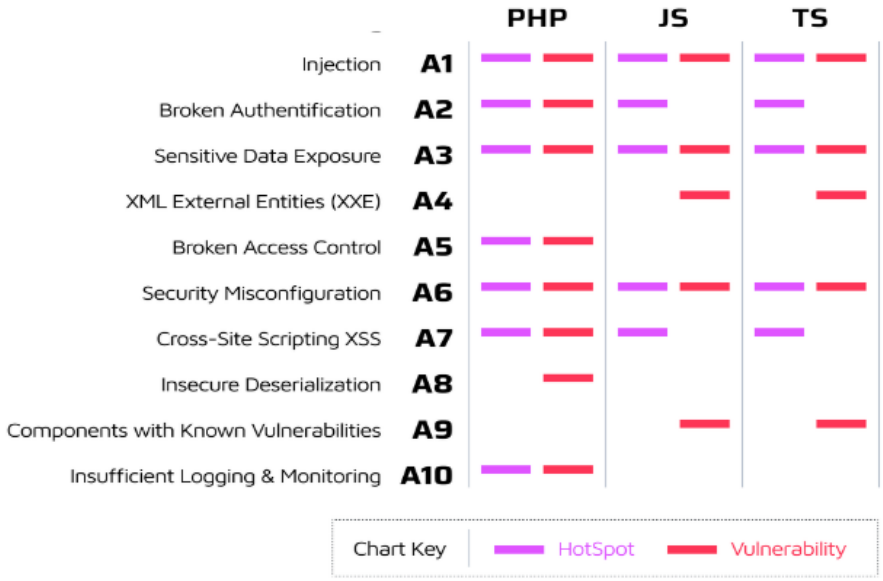

Figure 10 *Prevalence of OWASP top 10 detected by SonarQube divided by web programing languages [22]*

Injection: Injection flaws occur when an application fails to properly validate and sanitize user input. Attackers can exploit this vulnerability by inserting malicious code (such as SQL, OS commands, or LDAP queries) into user-supplied data, tricking the application into executing unintended commands. For example, SQL injection can allow an attacker to manipulate database queries, potentially gaining unauthorized access to sensitive data or modifying the database. Injection attacks are particularly dangerous as they can lead to data breaches, data loss, unauthorized access, or even complete system compromise. To mitigate this risk, input validation, parameterized queries, and the principle of least privilege should be applied.

1. Broken Authentication: This vulnerability refers to weaknesses in user authentication and session management mechanisms. Attackers can exploit weak authentication to compromise user accounts, impersonate legitimate users, or gain unauthorized access to sensitive areas of the application. Common weaknesses include weak password policies, improper session timeouts, or flaws in session token generation. To address this vulnerability, applications should implement secure authentication practices, such as using strong password hashing algorithms, enforcing password complexity requirements, implementing multi-factor authentication, and ensuring proper session management and token handling.
2. Sensitive Data Exposure: This vulnerability occurs when sensitive information is not adequately protected, making it accessible to unauthorized individuals. Examples of sensitive data include personal identifiable information (PII), credit card numbers, healthcare records, or any other confidential information. Attackers can exploit this vulnerability to steal or manipulate sensitive data, leading to identity theft, financial fraud, or reputational damage. Proper protection measures such as encryption, secure storage, and secure transmission protocols (such as HTTPS) should be implemented to safeguard sensitive data. Additionally, ensuring compliance with data protection regulations, such as GDPR or PCI DSS, is crucial.
3. XML External Entities (XXE): This vulnerability arises when an application processes XML data insecurely, allowing attackers to exploit external entities defined in the XML document. By leveraging XXE, attackers can read sensitive data from the server, perform denial-of-service attacks, or even execute arbitrary code. Older XML technologies that are not properly configured, such as SOAP, can be especially vulnerable to XXE attacks. To

mitigate this risk, applications should disable external entity references, use proper XML parsers, and implement secure XML processing practices.

4. Broken Access Control: This vulnerability occurs when an application fails to enforce proper access controls and authorization mechanisms. Attackers can exploit this weakness to gain unauthorized access to sensitive functionalities or data. Common issues include insufficient privilege validation, direct object references, or lack of proper access control checks. Effective access control measures, such as role-based access control (RBAC), fine-grained permissions, and regular security testing, should be implemented to ensure that only authorized users can access the appropriate resources.
5. Security Misconfigurations: Security misconfigurations refer to the improper setup or configuration of application components, servers, frameworks, and platforms. Common misconfigurations include default or weak settings, unnecessary services or features enabled, or outdated software versions. Attackers can take advantage of these misconfigurations to gain unauthorized access, extract sensitive information, or exploit known vulnerabilities. Proper configuration management practices, secure default configurations, regular patching, and vulnerability scanning are essential to address this vulnerability.
6. Cross-Site Scripting (XSS): XSS vulnerabilities occur when an application allows user-supplied data to be embedded in web pages without proper sanitization or encoding. Attackers can inject malicious scripts (usually JavaScript) into web pages, which are then executed by unsuspecting users' browsers. XSS attacks can lead to session hijacking, defacement of websites, phishing attacks, or theft of sensitive information. To mitigate XSS vulnerabilities, applications should implement input validation and output encoding techniques, use security libraries or frameworks that provide protection against XSS, and employ Content Security Policy (CSP) to restrict the execution of untrusted scripts.
7. Insecure Deserialization: Insecure deserialization vulnerabilities arise when an application deserializes untrusted or manipulated data from an external source. Attackers can exploit this vulnerability to execute arbitrary code, perform remote code execution attacks, or conduct denial-of-service attacks. Insecure deserialization is particularly dangerous as it can lead to full system compromise or unauthorized access to sensitive data. To mitigate

this risk, applications should avoid deserializing untrusted data or implement strict input validation and integrity checks during deserialization.

8. Using Components with Known Vulnerabilities: Many web applications rely on third-party components such as libraries, frameworks, or plugins. However, these components may contain known vulnerabilities that attackers can exploit. Organizations must regularly update and patch these components to ensure they are free from known vulnerabilities. Failure to do so can result in unauthorized access, data breaches, or compromise of the entire application. It is crucial to maintain an inventory of components, monitor security advisories, and promptly apply patches or updates when they become available.
9. Insufficient Logging and Monitoring: Insufficient logging and monitoring can hinder an organization's ability to detect and respond to security incidents effectively. Without proper logs and monitoring in place, it becomes difficult to identify and investigate malicious activities, detect attacks in real-time, or determine the full extent of a breach. To address this vulnerability, organizations should implement comprehensive logging mechanisms, including logging of security-relevant events, proper log storage and retention practices, and real-time monitoring and alerting systems to promptly respond to potential security incidents.

6. Implementation of security standards

In this section we will discus examples, methods, and ideas of what should be done to adhere to the principles of cybersecurity described in the previous section, this is an crucial step in understanding Results are represented by the remediation of the security vulnerabilities defined in the previous chapter. By use of SonarLint and SonarQube for detection of common vulnerabilities and reporting on Clean Code principal violations and re-coding unrepairable code parts.

6.1 Mobile Application

1. Improper Platform Usage

To protect against Improper Platform Usage, it's important to properly manage read and write permissions in an Android online library application. These permissions control the app's access to the device's external storage, allowing it to read and write files. By correctly managing

these permissions, you can prevent unauthorized access to user data and ensure that the app operates within the intended scope.

First, let's consider the scenario of reading data from external storage. To request the necessary read permission, you would need to include the following code in your AndroidManifest.xml file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Figure 11 *Android XML permissions*

In this example, the <uses-permission> tags are added inside the <manifest> element to request the READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions. These permissions are necessary for reading and writing data to the external storage.

```
private static final int STORAGE_PERMISSION_REQUEST_CODE = 101;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED
            || ContextCompat.checkSelfPermission(this, Manifest.permission.WRITE_EXTERNAL_STORAGE)
            != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.READ_EXTERNAL_STORAGE,
                Manifest.permission.WRITE_EXTERNAL_STORAGE}, STORAGE_PERMISSION_REQUEST_CODE);
    }
```

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
    if (requestCode == STORAGE_PERMISSION_REQUEST_CODE) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED && grantResults[1]
        == PackageManager.PERMISSION_GRANTED) {
            // Storage permissions granted, proceed with file operations
        } else {
            // Storage permissions denied, handle accordingly
            Toast.makeText(this, "Storage permissions denied", Toast.LENGTH_SHORT).show();
        }
    }
}
```

Figure 12 *Improper Platform Usage Code segments*

ContextCompat.checkSelfPermission() is used in this example to check if both READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE permissions are permitted. If the permissions are not granted, ActivityCompat.requestPermissions() is called to ask the user for the permissions. The onRequestPermissionsResult method handles the permission request result, where you can conduct the necessary actions based on whether the permissions were granted or refused.

By properly managing read and write permissions in the online library application, you can ensure that access to external storage is controlled and secure. This helps protect user data, prevent unauthorized access to files, and ensures the application functions as intended.

2. Insecure Data Storage:

The vulnerability of storing sensitive data in an unencrypted or easily accessible manner, rendering it vulnerable to unwanted access or tampering, is referred to as insecure data storage. It is critical to employ secure data storage methods in the online library application to protect against this issue. Let's look at some major ideas and code examples for protecting Android data storage.

2.1 Use Encryption: One of the fundamental methods to protect sensitive data is to encrypt it before storing it on the device. Android provides the javax.crypto package, which includes classes and APIs for cryptographic operations.

2.2 Use Android Keystore: The Android Keystore system provides a secure storage area for cryptographic keys. It ensures that keys are protected and inaccessible to unauthorized users or malicious apps.

2.3 Secure Shared Preferences: Android's ‘SharedPreferences’ is a common storage mechanism for app settings and small amounts of data. However, it is not suitable for storing sensitive information without proper encryption. To secure data stored in ‘SharedPreferences’, you can encrypt it using a cryptographic algorithm such as AES.

2.4 Secure File Storage: When storing files on the device, it's important to ensure they are stored securely. One approach is to encrypt the files using cryptographic algorithms.

By using these tactics, you may considerably improve the security of data storage in the Android online library app. Encryption, in conjunction with the Android Keystore, secure storage methods

such as SharedPreferences, and secure file storage strategies, aids in the protection of sensitive information, the prevention of illegal access, and the confidentiality and integrity of user data. These safeguards are critical in developing a strong security system for an online library, inspiring trust in users, and limiting the hazards associated with unsafe data storage.

3. Insecure Communication:

Vulnerabilities in the transmission of data between the online library application and its backend servers or other third-party services are referred to as insecure communication. To guard against this risk, secure communication protocols that ensure the confidentiality, integrity, and authenticity of the data being transmitted must be implemented. Let's look at some essential strategies and code examples for Android communication security.

3.1 Use HTTPS for Network Requests:

The first step to secure communication is to use the HTTPS protocol for all network requests. HTTPS encrypts the data being transmitted between the client and the server, protecting it from eavesdropping and tampering. Android provides the HttpsURLConnection class for making secure network requests.

3.2 Implement Certificate Pinning:

Certificate pinning is a technique that ensures the server's identity by comparing its public key or certificate against a known or pre-defined value. This prevents attackers from intercepting the communication by presenting their own malicious certificates. Android provides the 'CertPinManager' class for implementing certificate pinning. In the following image we can see a way of implementing certificate pinning.

```
public static void enableCertificatePinning() {
    try {
        TrustManager[] trustManagers = new TrustManager[]{new PinningTrustManager()};
        SSLContext sslContext = SSLContext.getInstance("TLS");
        sslContext.init(null, trustManagers, null);
        HttpsURLConnection.setDefaultSSLSocketFactory(sslContext.getSocketFactory());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

```
@Override
public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException {
    for (X509Certificate certificate : chain) {
        byte[] publicKey = certificate.getPublicKey().getEncoded();
        String pinnedPublicKey = "sha256/" + android.util.Base64.encodeToString(publicKey, android.util.Base64.NO_WRAP);
        if (PINNED_PUBLIC_KEY.equals(pinnedPublicKey)) {
            return; // Certificate is trusted
        }
    }
    throw new CertificateException(msg: "Certificate not trusted");
}

@Override
public X509Certificate[] getAcceptedIssuers() {
    return new X509Certificate[0];
}
```

Figure 13 *Implement Certificate Pinning code examples*

In the code example, the 'CertPinManager' class implements a custom X509TrustManager. The server's public key is extracted from the certificate and compared with a pre-defined pinned public key value. If the public key matches, the certificate is considered trusted, and the communication proceeds. If the public key does not match, an exception is thrown, indicating that the certificate is not trusted. This helps protect against attacks where an attacker presents a fraudulent certificate.

3.3 Implement Secure Socket Layer (SSL)/TLS Configuration:

To further enhance communication security, it is important to configure secure SSL/TLS protocols and cipher suites. Android allows customization of SSL/TLS settings by creating a custom 'SSLSocketFactory'. Proper SSL/TLS configuration ensures that strong encryption algorithms and secure protocols are used for communication.

3.4 Enable Network Security Configuration:

Android provides a Network Security Configuration file (network_security_config.xml) that allows defining security settings for network requests made by the app. This configuration file enables specifying trusted certificate authorities, enabling/disabling specific TLS versions and cipher suites, and other security-related options. One can improve the security of communication in the online library application on the Android platform by using these measures. You construct a secure channel for data transmission by using HTTPS for network requests, implementing certificate pinning, defining SSL/TLS settings, and employing the Network Security Configuration file. This protects against eavesdropping, tampering, and unauthorized access. These steps help to

develop a strong security system for the online library while also increasing user trust and preserving the privacy and integrity of user data.

4. Insecure Authentication:

Insecure Authentication refers to vulnerabilities in the authentication process that can lead to unauthorized access to the online library system. To protect against this vulnerability, it is crucial to implement robust authentication mechanisms that verify the identity of users and protect their credentials.

4.1 Ensure high password strength by asking the user to have a password length of 8 characters that should contain at least one lower and upper case letter, a number and a special character

4.2 Implement Secure Session Management:

Proper session management is crucial to prevent session hijacking and unauthorized access. Android provides the ‘SharedPreferences’ class for storing session-related. the ‘SessionManager’ 'class uses the ‘SharedPreferences’ class to store and retrieve the session token. The token is saved securely in the shared preferences, which can only be accessed by the application itself. Clearing the session is also provided to ensure proper session termination when the user logs out or the session expires.

4.3 Implement Account Lockout Mechanism:

To mitigate brute-force attacks and password guessing, implement an account lockout mechanism. After a set number of failed login attempts, this method temporarily freezes user accounts. Android includes a 'LoginTracker' feature.

Strong password policies, proper session management, and an account lockout mechanism all help to protect user accounts and prevent illegal access. These steps improve the security of the online library system by fostering user trust and ensuring the confidentiality and integrity of user data.

5. Insufficient Cryptography Risks:

Insufficient Cryptography Risks are vulnerabilities in the encryption and cryptography methods that can lead to the compromise of sensitive data in an online library system. To mitigate such risks, it is critical to use strong, up-to-date cryptographic methods and best practices. Strong cryptographic algorithms, proper key management, secure transport layer protocols, secure key exchange, and secure storage of cryptographic material all contribute to the online library's comprehensive security system. These steps improve overall system security and protect critical user information.

6. Insecure Authorization Risks:

On the Mobile version of the application only a single type of user can access the application’s functionalities so there are no problems derived from role-based access controls.

7. Poor Code Quality Risks:

Poor Code Quality Risks refer to vulnerabilities that arise from insecure coding practices, such as lack of input validation, improper error handling, and insecure storage of sensitive data. To mitigate these risks and ensure a secure online library system, it is essential to follow best coding practices and implement robust security measures.

7.1 Input Validation:

Proper input validation is crucial for preventing common security vulnerabilities, such as SQL injection and cross-site scripting (XSS) attacks. Android provides various validation mechanisms, such as regular expressions and built-in validation methods

7.2 Error Handling and Logging:

Effective error handling and logging are essential for identifying and addressing security issues. Android provides the Log class for logging information, warnings, and errors.

7.3 Secure Data Storage:

Properly storing sensitive data, such as user credentials and API keys, is critical for maintaining the security of the online library system. Android provides several options for secure data storage, such as ‘SharedPreferences’ and the Android Keystore system.

7.4 Code Reviews and Static Analysis:

Regular code reviews and static analysis help identify security vulnerabilities and code quality issues. Android provides various static analysis tools, such as SonarQube and SonarLint, to analyze your code for potential issues. Additionally, conducting code reviews with peers can uncover potential security risks. The following code review checklist for security is a good way of ensuring good code is written:

- Ensure proper input validation is implemented.
- Check for secure storage and handling of sensitive data.
- Verify that error handling and logging practices are in place.
- Validate that encryption and hashing mechanisms are correctly implemented.
- Identify any hardcoded secrets or credentials that should be removed.
- Ensure the use of secure network communication protocols.

By incorporating code reviews and static analysis into your development process, you can catch security vulnerabilities and improve code quality before deploying the online library system.

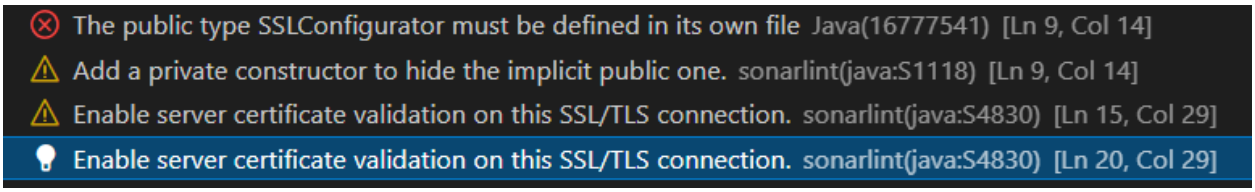

Figure 14 *SonarLint code problems highlight*

7.5 Secure Network Communication:

Securing network communication is crucial to protect user data from eavesdropping and tampering. Android provides the 'HttpsURLConnection' class for secure HTTP communication over SSL/TLS.

```
HttpsURLConnection connection = (HttpsURLConnection) url.openConnection();
connection.setRequestMethod("GET");
connection.setConnectTimeout(5000);
connection.setReadTimeout(5000);
connection.setSSLSocketFactory(getSSLSocketFactory());
connection.setHostnameVerifier(getHostnameVerifier());
```

Figure 15 *Secure Network Communication code example*

Through implementation of these strategies for protecting against Poor Code Quality Risks, you can enhance the security and reliability of your Android-based online library system. Proper input validation, error handling and logging, secure data storage, code reviews, and secure network communication contribute to mitigating security vulnerabilities and ensuring a robust codebase.

8. Code Tampering Risks:

Code Tampering Risks refer to vulnerabilities that arise from unauthorized modifications or tampering of the application code, which can lead to security breaches or unauthorized access. To protect against these risks and ensure the integrity of the online library system, it is essential to implement measures that detect and prevent code tampering attempts. By code signing the apk through the use of android studio apk code signing we ensure that the released version is a verified one.

8.1 Code Obfuscation:

Code obfuscation is a technique that transforms the application's code into a more complex and less readable form, making it harder for attackers to understand and modify the code. Android provides the ProGuard tool for code obfuscation.

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Figure 16 *Android ProGuard*

By enabling ‘minifyEnabled’ and specifying the ProGuard configuration file, the code will be obfuscated during the release build process. Code obfuscation makes it challenging for attackers to modify the code or inject malicious code snippets.

8.2 Code Integrity Checks:

Implementing code integrity checks helps detect any modifications or tampering attempts on the application's code. You can calculate and verify checksums or digital signatures of critical code components. Here's an example of verifying the checksum of a file:

```
private boolean verifyChecksum(File file, String expectedChecksum) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        FileInputStream fis = new FileInputStream(file);
        byte[] buffer = new byte[8192];
        int bytesRead;
        while ((bytesRead = fis.read(buffer)) != -1) {
            digest.update(buffer, 0, bytesRead);
        }
        fis.close();
        byte[] checksumBytes = digest.digest();
        String calculatedChecksum = bytesToHex(checksumBytes);
        return calculatedChecksum.equals(expectedChecksum);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
```

Figure 17 Code Integrity Checks code example

In this example, the 'verifyChecksum' method calculates the SHA-256 checksum of the APK file and compares it with the expected checksum. By performing code integrity checks, you can detect any unauthorized modifications to your application's code.

8.3 Certificate Pinning:

Implementing certificate pinning helps prevent Man-in-the-Middle (MitM) attacks by ensuring that the application only communicates with servers using trusted certificates.

8.4 Binary Protections:

Applying binary protections can make it harder for attackers to reverse engineer or modify the code. Android provides tools like ProGuard and DexGuard that offer additional binary protection features, such as code encryption and anti-debugging techniques. By leveraging these tools, you can add an extra layer of protection to your application's binary code.

8.5 Tamper Detection Libraries:

There are third-party libraries available that specialize in detecting code tampering attempts. These libraries provide features like runtime integrity checks, anti-debugging measures, and jailbreak/root detection. One such library is the SafetyNet API provided by Google Play Services. It offers tamper detection capabilities and can be integrated into your application to detect unauthorized modifications.

9. Reverse Engineering Risk:

Reverse Engineering Risks refer to the process of analyzing and understanding an application's code to extract sensitive information or discover vulnerabilities. To protect against these risks and safeguard the intellectual property and security of the online library system, it is essential to implement measures that make reverse engineering challenging and discourage malicious attempts. Some of the methods were described above , these are Code Obfuscation, Binary Protection, Code Integrity Checks, in addition to those there are 2 more methods of insuring production in case of this vulnerability.

9.1 Native Code Protection:

If your program makes use of native code libraries, you might want to implement security measures such code obfuscation and anti-debugging strategies. Additionally, you can use JNI (Java Native Interface) and encryption technologies to safeguard delicate native code activities.

9.2 Remote Code Execution:

Offloading delicate activities to remote servers can help to reduce the danger of intruders hacking and tampering with vital code. You can lessen the risk of sensitive code being exposed in the client application by building a server-side component and carrying out important computations or actions on the server.

10. Extraneous Functionality Risk:

Extraneous Functionality Risks refer to the presence of unnecessary or unused functionality within an application that can introduce security vulnerabilities. To protect against these risks and ensure the security and stability of the online library system, it is crucial to implement measures that identify and remove any unnecessary or unused functionality. The following are ways of protecting against the mentioned risk

10.1 Code Review and Analysis:

Performing regular code reviews and analysis is essential to identify and remove any extraneous or unused functionality. Review your codebase to identify features, libraries, or APIs that are no longer required or are potentially vulnerable. By removing unnecessary code, you can reduce the attack surface and improve the overall security posture of the online library system.

10.2 Permissions and API Usage:

Review the permissions and API usage within your application to ensure that only the necessary permissions and APIs are requested. Unnecessary permissions and excessive API access can introduce security risks

10.3 Third-Party Libraries:

Third-party libraries can provide additional functionality to your application but may also introduce extraneous or unused features. Regularly reviewing the libraries used in the online library system and ensure that you are only including the necessary components. Remove any unused or unnecessary libraries to minimize the risks associated with them.

10.4 Security Testing:

Conduct comprehensive security testing, including penetration testing and vulnerability assessments, to identify any extraneous functionality that may introduce security vulnerabilities. By systematically testing your application's features and functionalities, you can uncover potential risks and address them before deploying your online library system.

6.2 Web Application

1. Injection:

For web applications, especially Angular sites, injection attacks like SQL injection and XSS (Cross-Site Scripting) can be catastrophic. These attacks entail the introduction of malicious code into user inputs or dynamically generated information, giving attackers the ability to influence or carry out unauthorized operations. It is essential to put preventive measures in place to safeguard your online library system against Injection assaults and preserve its security. Let's examine some essential techniques and sample code for preventing Injection attacks in an Angular online application.

1.1 Parameterized Queries:

Use prepared statements or parameterized queries rather than immediately concatenating user input into the query when working with databases. This stops harmful input from being included into the logic of the query.

1.2 Input Validation and Sanitization:

Always validate and sanitize user inputs to avoid processing or displaying harmful data. Validators and sanitizers are integrated into Angular and can be added to form controls or input fields. Here's an example of how to use the built-in validation and sanitization functions in Angular:

```
<input type="text" [(ngModel)]="commet" [pattern]="/^[a-zA-Z0-9]*$/" required>
    {{ comment | sanitize }}
</input>
```

Figure 18 *Input Validation and Sanitization code example*

1.3 Content Security Policy (CSP):

Implementing a material Security Policy (CSP) to limit the sorts of material that can be loaded or performed on the Angular site. You may reduce the danger of code injection attacks by setting a rigorous policy that whitelists reliable sources for scripts, stylesheets, and other resources. Here's an example of a Content Security Policy being specified in the server response headers:

1.4 Security Headers:

To provide an additional layer of protection against Injection attacks, I will implement security headers. For instance, I will set the 'X-XSS-Protection' header to 1; mode=block to enable the browser's built-in XSS protection mechanisms. By configuring this header in my server's response, I instruct the browser to block any detected XSS attacks.

2. Broken Authentication:

Broken Authentication occurs when authentication mechanisms are improperly implemented, leading to vulnerabilities that can be exploited by attackers to gain unauthorized access to user accounts.

2.1 Strong Password policy:

Enforcing strong password policy is one technique to overcome Broken Authentication. We may greatly strengthen the authentication process by introducing password complexity requirements such as minimum length, the inclusion of alphanumeric characters, and the ban of commonly used or readily guessable passwords. Here's how I'll enforce a password policy during user registration:

```
this.registrationForm = this.formBuilder.group({
  username: ['', Validators.required],
  password: ['', [
    Validators.required,
    Validators.minLength(8),
    Validators.pattern(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]+$/),
  ]],
});
```

Figure 19 *Validators in angular forms*

In the code snippet above, the Validators.pattern validator is used to enforce a specific pattern for the password. The pattern ^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]+$ ensures that the password contains at least one lowercase letter, one uppercase letter, one digit, and one special character. By using this revised code, the registration form will validate that the password meets the specified criteria, ensuring the inclusion of at least one uppercase letter, one lowercase letter, one special character, and one number.

2.2 Secure Session Management:

To prevent unwanted access to user accounts, proper session management is essential. To guarantee that sessions are securely established and terminated, I will incorporate safeguards such as session timeouts, secure cookie processing, and single sign-on features. Following is an example of how to configure a session timeout in the server:

```
app.use(session({
  secret: createdSecret,
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: true,
    maxAge: 60 * 60 * 1000, // expiration after one hour
  },
}));
```

Figure 20 *Secure Session Management code example*

2.3 Protection Against Brute Force Attacks:

Account lockouts and rate limiting techniques will be implemented to avoid brute force attacks, in which attackers attempt to acquire unauthorized access by repeatedly guessing passwords. After a set number of failed login attempts, the user's account is temporarily or

permanently locked, reducing the possibility of successful brute force attacks. Below is an example of how I'll put a basic account lockout mechanism in place:

```
const MAX_LOGIN_ATTEMPTS = 3;// 3 Log in attempts
const ACCOUNT_LOCK_DURATION = 60 * 60 * 1000; // Account locked for 60 minutes

if (loginAttempts >= MAX_LOGIN_ATTEMPTS) {
  if (Date.now() - lastFailedLoginTime <= ACCOUNT_LOCK_DURATION) {···
  } else {···
  }
}
```

Figure 21 *Max log in attempts, angular code example*

2.4 Regular Security Audits:

I will conduct frequent security audits to regularly monitor the efficiency of our security measures and uncover any weaknesses. These audits will include code reviews, penetration testing, and vulnerability scanning to assess the security of our authentication mechanisms. We can maintain a robust and resilient security system for our online library by remaining watchful and proactively correcting any gaps.

3. Sensitive Data Exposure:

. Sensitive Data Exposure happens when sensitive information, such as user credentials or personal information, is not effectively protected and slips into the hands of the wrong people. To mitigate this danger and assure the security of our online library system, I will implement several solutions and present some code samples.

3.1 Encryption of Data in Transit:

I will mandate the use of secure communication protocols, such as HTTPS, which encrypts data exchanged between the user's browser and the server, to protect sensitive data during transmission. I can assure that all data transmitted is encrypted by configuring the server to use SSL/TLS certificates. Here's an example of how I'll make HTTPS mandatory in the server configuration:

```
const options = {
  key: fs.readFileSync('private.key'),
  cert: fs.readFileSync('public.crt'),
};

https.createServer(options, app).listen(443);
```

Figure 22 Data encryption code example

3.2 Secure Storage of User Credentials:

To secure user credentials, I will not keep passwords in plaintext. Instead, I'll use safe hashing algorithms like 'bcrypt' to securely store and compare passwords. Even if the stored data is hacked, attackers will find it incredibly difficult to recover the original passwords by utilizing a salted hash.

3.3 Proper Error Handling:

To avoid exposing sensitive data via error messages, I will guarantee that error messages do not reveal sensitive information. Users will see generic error messages, but specific error messages including sensitive data will be safely logged for debugging purposes.

3.4 Regular Vulnerability Scanning and Penetration Testing

4. XML External Entities (XXE):

XML External Entities (XXE) attacks exploit vulnerable XML parsers by injecting malicious external entities, which can lead to sensitive data disclosure, server-side request forgery, and denial of service.

4.1 Disabling External Entity Resolution:

To prevent XXE attacks, I will disable external entity resolution within the XML parser. We may successfully avoid this vulnerability by configuring the XML parser to prevent the processing of foreign elements.

```
export class XmlParserComponent {
  constructor() {
    const parser = new DOMParser({
      // Disable external entity resolution
      entityResolver: () => null,
    });
  }
```

Figure 23 Xml parser code example

By setting the entityResolver option to a function that returns null, we effectively disable the resolution of external entities in the XML parser.

4.2 Input Validation and Sanitization:

As described in the previous section I will implement strict input validation and sanitization techniques. This involves validating user input against expected data formats and sanitizing any XML input to remove potentially malicious content

5. Broken Access Control:

Broken Access Control occurs when the system fails to properly enforce restrictions on what resources a user can access or manipulate.

5.1 Role-Based Access Control (RBAC):

Role-Based Access Control (RBAC) will be implemented in our Angular application to enforce correct access control. RBAC is a popular method for assigning responsibilities to users and restricting their access depending on those roles. Those roles in the web variant of the app will be the normal user and Content creator someone that can create books.

```
enum UserRole {
    Basic = 'user',
    Creator = 'creator'
}

const user = new User(UserRole.Basic);
const creator = new Creator(UserRole.Creator)
// Check access based on user role
if (user.role === UserRole.Basic) {···
} else if (user.role === UserRole.Creator) {···
}  else {
  // Handle unauthorized access
}
```

Figure 24 Role-Based Access Control code example

5.2 Access Control List (ACL):

I will implement an Access Control List (ACL) method in addition to RBAC to fine-tune access control on a per-resource basis. We can use ACL to set explicit permissions for individual resources and grant or revoke access as needed.

```
// Define resource permissions
enum ResourcePermission {
    Download = 'download',
    PictureUpload = 'picture_upload',
    CreateBook = 'create_book',
  }

  // Resource class with ACL property
  class Resource {
    constructor(public acl: Map<UserRole, ResourcePermission[]>) {}
  }

  // Usage example
  const resource = new Resource(
    new Map([
      [UserRole.Basic, [ResourcePermission.Download, ResourcePermission.PictureUpload]],
      [UserRole.Creator, [ResourcePermission.Download, ResourcePermission.PictureUpload, ResourcePermission.CreateBook]],
    ])
  );

  // Check access based on user role and resource permission
  const userRole = user.role;
  if (resource.acl.has(userRole) && resource.acl.get(userRole)?.includes(ResourcePermission.Download)) {
    // Grant read access
  } else {
    // Handle unauthorized access
  }
```

Figure 25 Access Control List code example

6. Security Misconfigurations:

By implementing secure configuration management, secure HTTP headers, error handling and logging, secure deployment practices, regular security audits, and RBAC, we can effectively protect our web Angular site for the online library against security misconfigurations. These measures ensure that our application is properly configured, follows best practices, and reduces the risk of potential vulnerabilities and unauthorized access.

6.1 Secure Configuration Management:

I will create a secure configuration management method for our Angular application to prevent security misconfigurations. This entails keeping track of configuration settings such as database credentials, API keys, and server configurations and guaranteeing their secure storage and protection.

6.2 Secure HTTP Headers:

I will configure secure HTTP headers to improve the security of our web application. These headers give further protection by preventing specific sorts of attacks, such as cross-site scripting (XSS) and clickjacking.

7. Cross-Site Scripting (XSS):

One of the key areas I need to address is protecting against Cross-Site Scripting (XSS) attacks. XSS attacks occur when malicious scripts are injected into web pages and executed in the context of unsuspecting users, potentially leading to unauthorized access, data theft, or other malicious activities.

7.1 Input Validation and Sanitization:

To prevent XSS attacks, I will build comprehensive input validation and sanitization techniques throughout our Angular application. This includes verifying and sanitizing all user inputs, including form inputs, query parameters, and URL routes. Let's lookt at an example of how I will implement input validation and sanitization using Angular's built-in 'DomSanitizer', declared in the constructor as 'sanitizer':

```
// Sanitize user input
this.safeHTML = this.sanitizer.bypassSecurityTrustHtml(this.userInput);
```

Figure 26 *Input sanetizer*

In the code snippet above, the 'DomSanitizer' is used to sanitize the user input before rendering it in the template. This helps prevent the execution of any potentially malicious scripts.

7.2 Implementing a Content Security Policy (CSP):

Is another excellent way to prevent XSS attacks. CSP enables the website administrator to design a set of rules that describe which content sources can be trusted. The following is an example of how I'll set up a Content Security Policy in our Angular application:

```
<head>
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' 'unsafe-inline';">
</head>
```

Figure 27 *Content Security Policy code example*

7.3 Escaping User-Generated Content:

When displaying user-generated content, it is crucial to properly escape the content to prevent XSS attacks. Angular provides built-in mechanisms for escaping user-generated content, such as using the '{{}}' interpolation syntax or the '[innerHTML]' property binding with the 'DomSanitizer'. It's essential to ensure that user-generated content is escaped before rendering it in the DOM.

7.4 HTTP Headers and Secure Cookies:

Setting appropriate HTTP headers can help prevent XSS attacks. For instance, the 'X-XSS-Protection' header can enable the browser's XSS protection mechanisms. Additionally, ensuring the proper configuration of secure cookies with the HttpOnly flag can mitigate the risk of XSS attacks targeting sensitive session data.

8. Insecure Deserialization

To defend against Insecure Deserialization, I implemented several protective measures. Firstly, I adopted a robust input validation strategy to ensure that only trusted data was deserialized. This involved validating the integrity and authenticity of serialized objects before deserialization. Secondly, I utilized whitelisting to restrict the types of objects that could be deserialized, minimizing the risk of executing malicious payloads. Lastly, I closely monitored the deserialization process, implementing strict access controls and limiting the privileges of deserialized objects.

9. Using Components with Known Vulnerabilities:

To address the risk of utilizing components with known vulnerabilities, I adopted a proactive approach. I maintained an updated inventory of all third-party components used in the online library application, including libraries, frameworks, and plugins. Regularly checking for security advisories and updates, I promptly applied patches or upgraded to newer versions to eliminate known vulnerabilities. Additionally, I subscribed to vulnerability alert services to receive real-time notifications about any newly discovered vulnerabilities in the components I utilized. This allowed me to take immediate action and protect the application from potential exploits.

10. Insufficient Logging and Monitoring:

By implementing comprehensive logging mechanisms. By logging critical events and activities, I created a comprehensive record for security auditing and monitoring purposes. Regularly reviewing these logs allowed me to identify any suspicious or malicious activities promptly. To enhance my vigilance, I set up alerts and notifications to receive real-time notifications of potential security breaches.

Conclusion

In closing, as a working on my dissertation on developing a security system for an online library, I have investigated many vulnerabilities and security techniques to protect web and mobile applications. We have centered our discussions on the OWASP Top Ten vulnerabilities and their importance in the context of our online library security system.

We began by recognizing the significance of vulnerabilities such as Injection, Broken Authentication, Sensitive Data Exposure, XML External Entities, Broken Access Control, Security Misconfigurations, Cross-Site Scripting, Insecure Deserialization, Insecure Direct Object References, and Insufficient Logging and Monitoring. We went into detail about each vulnerability and offered code examples for both web and mobile environments.

We uncovered how to protect against these vulnerabilities by using various security procedures on the web Angular site. Input validation and sanitization, content security policy, escaping user-generated content, setting secure HTTP headers, implementing secure authentication and authorization mechanisms, using encryption and secure storage for sensitive data, and conducting regular security testing and code reviews were all discussed. These techniques work together to build a strong security system for our online library.

Similarly, in the Android mobile app, we talked about how to protect against vulnerabilities like improper platform usage, insecure data storage, insecure communication, insecure authentication, insufficient cryptography, insecure authorization, poor code quality, code tampering, reverse engineering, and extraneous functionality. To strengthen the security of our mobile app, we investigated strategies such as permission handling, secure storage, SSL/TLS encryption, secure authentication protocols, correct cryptographic algorithms, access control mechanisms, code obfuscation, integrity checks, and audits.

Finally, developing a security system for an online library necessitates a thorough grasp of the OWASP Top Ten vulnerabilities, as well as the rigorous execution of secure coding practices and ongoing security assessments. We can establish an online library that not only provides excellent resources but also protects user data and preserves user confidence by incorporating robust security measures and taking a proactive approach to security.

Table of Figures

Figure 1 use case diagram 5
Figure 2 Log in Activity Diagram 5
Figure 3 Download book Activity Diagram 6
Figure 4 Deployment Diagram 7
Figure 5 Communication Diagram 7
Figure 6 Class diagram 8
Figure 7 Database Diagram 9
Figure 8 Suit of Firebase services 14
Figure 9 OWASP top 10 mobile violations [21] 17
Figure 10 Prevalence of OWASP top 10 detected by SonarQube divided by web programing languages [22] 20
Figure 11 Android XML permissions 24
Figure 12 Improper Platform Usage Code segments 24
Figure 13 Implement Certificate Pinning code examples 27
Figure 14 SonarLint code problems highlight 30
Figure 15 Secure Network Communication code example 30
Figure 16 Android ProGuard 31
Figure 17 Code Integrity Checks code example 32
Figure 18 Input Validation and Sanitization code example 35
Figure 19 Validators in angular forms 36
Figure 20 Secure Session Management code example 36
Figure 21 Max log in atempts, angular code example 37
Figure 22 Data encryption code example 38
Figure 23 Xml parser code example 39
Figure 24 Role-Based Access Control code example 39
Figure 25 Access Control List code example 40
Figure 26 Input sanetizer 41
Figure 27 Content Security Policy code example 42

Bibliography

[1] F. T. F. Stan Sebastian-Daniel, *NetTheca,* Bucharest: Faculty of Cybernetics, Statistics and Economic Informatics, 2021.

[2] A. Goel, S. Gautam, N. Tyagi, N. Sharma and M. Sagayam, Securing Biometric Framework with Cryptanalysis, 2021.

[3] G. Deepa and P. S. Thilagam, Securing Web Applications from Injection, 2016.

[4] R. Mayrhofer, An architecture for secure mobile devices, 2014.

[5] Z. Bezhovski, "The Future of the Mobile Payment as Electronic Payment System," *European Journal of Business and Management,* vol. 8, no. 8, pp. 127-132, 2016.

[6] S. Fatonah, A. Yulandari and F. W. Wibowo, "A Review of E-Payment System in E-Commerce," *Journal of Physics: Conference Series,* vol. 1140, 2018.

[7] F. Wang, G. B. Shan, Y. Chen, X. Zheng, H. Wang, S. Mingwei and L. Haihua, "Identity Authentication Security Management in Mobile Payment Systems," *Journal of Global Information Management,* vol. 28, no. 1, pp. 189-203, 2020.

[8] A. Malikl, A. Burney and F. Ahmed, "A Comparative Study of Unstructured Data with SQL and NO-SQL Database Management Systems," *Journal of Computer and Communications,* vol. 8, pp. 59-71, 2020.

[9] V. Sachdeva and S. Gupta, "BASIC NOSQL INJECTION ANALYSIS AND," in *International Conference on Advanced Computation and Telecommunication*, Bhopal, India, 2018.

[10] M. Hussain, A. F. Siddiqui, P. Haswani and F. Majid, "Comprehensive Analysis on SQL and No SQL," *ASIAN JOURNAL OF ENGINEERING, SCIENCES & TECHNOLOGY,* vol. 11, no. 1, pp. 15-24, 2021.

[11] V. Sachdeva and S. Gupta, "Basic NOSQL Injection Analysis And Detection On MongoDB," in *International Conference on Advanced Computation and Telecommunication*, Bhopal, India, 2018.

[12] SUFATRIO, D. J. J. TAN and T.-W. CHUA, "Securing Android: A Survey, Taxonomy, and Challenges," in *ACM Computing Survey*, 2015, pp. 58-113.

[13] A. Bartel, J. Klein, Y. L. Traon and M. Monperrus, "Android, Automatically securing permission-based software by reducing the attack surface: an application to," in *International Conference on Automated Software Engine*, Essen, Germany, 2012.

[14] C. K. Behera and D. L. Bhaskari, "Different Obfuscation Techniques for Code Protection," in *International Conference on Eco-friendly Computing and Communication Systems* , Visakhapatnam, India, 2015.

[15] M. Ceccato1, M. D. Penta, J. Nagra and P. Falcarin, "The effectiveness of source code obfuscation: An experimental assessment," in *International Conference on Program Comprehension*, Vancouver, BC, Canada, 2009.

[16] W. Kankanamge, A. N. Dias and P. Siriwardena, Microservices Security in Action, Simon and Schuster, 2020.

[17] R. Chandramouli and Z. Butcher, Building Secure Microservices-based, NIST Special Publication, 2020.

[18] "OWASP Application Security Verification Standard," The OWASP Foundation Inc., [Online]. Available: https://owasp.org/www-project-application-security-verification-standard/. [Accessed 29 3 2023].

[19] A. Nehme, K. Mahbub, V. Jesus and A. Abdallah, "Securing Microservices," *IT Professional, 21,* vol. 21, no. 1, pp. 42-49, 2019.

[20] R. Chandramouli, Security Strategies for Microservices-based Application Systems, NIST Special Publication, 2019.

[21] T. Oh, B. Stackpole, E. Cummins, C. Gonzalez and R. Ramachandran, "Best Security Practices for Android, BlackBerry, and iOS," in *Workshop on Enabling Technologies for Smartphone and Internet of Things*, 2012, pp. 42-47.

[22] N. Agarwal, *Understanding OWASP Mobile Top 10 Risks with Real-world Cases,* Appinventiv, 2020.

[23] Sonar, "SonarSource," Sonar, [Online]. Available: https://www.sonarsource.com/solutions/security/owasp/?gads_campaign=SQ-Class01-Generic&gads_ad_group=DSA&gads_keyword=&gclid=Cj0KCQjwj_ajBhCqARIsAA37s0zJCTSmaGcy7Pz5atteeXEC3yGjXCuw4kCKKqE13by8pk5tZVdtoT0aAsJsEALw_wcB. [Accessed 1 6 2023].