DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Information Systems

Reducing Web Application Vulnerabilities through the Informed Choice of Webframeworks, Libraries and Automated Tools

Moritz Hüther

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Information Systems

Reducing Web Application Vulnerabilities through the Informed Choice of Webframeworks, Libraries and Automated Tools

Reduzierung der Schwachstellen von Webapplikationen durch die bewusste Wahl von Webframeworks, Bibliotheken und automatisierten Werkzeugen

Author:	Moritz Hüther
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Sascha Nägele, M.Sc.
Submission Date:	15.06.2021

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.06.2021 Moritz Hüther

Acknowledgments

I would like to thank my advisors Sascha Nägele and Richard Hudson for their support throughout the thesis as well as Professor Matthes and the Technical University of Munich for this opportunity.

Moreover, I would also like to thank the msg systems ag and in particular msg Research for all of their guidance, knowledge, and supporting words.

Finally, I thank my family and friends for all their ongoing support and encouragement during this time.

Abstract

Web-based applications are becoming a common choice for companies to deliver their services to their customers. To achieve this, they store data relating to their customers, making them a target for cybercriminals. These criminals make use of a variety of different errors and the vulnerabilities they cause to take control of the system, steal data or make the system unavailable. The costs generated through such attacks are rising annually, thus the security of web applications should be of a high priority. Therefore, the selection of an appropriate technology stack should be carried out with security in mind.
This thesis proposes an evaluation of used web application frameworks, libraries, and automated tools which are commonly used nowadays. The evaluation highlights different levels of security that can be achieved through the selection of common solutions containing web framework-native functionalities in combination with third-party libraries. Finally, automated tools such as Static Application Security Testing tools will be evaluated based on the level of support they give to make the previously defined solutions more secure. The result of these evaluations will give an overview of the coverage of vulnerabilities through these solutions and tools.

Kurzfassung

Webanwendungen werden für Unternehmen zu einer gängigen Wahl, um ihren Kunden ihre Dienste bereitzustellen. Zu diesem Zweck speichern die Webanwendungen Daten über ihre Kunden und machen sie dadurch zu einem Ziel von Cyberkriminellen. Diese Kriminellen nutzen Codierungsfehler und die Schwachstellen, die sie verursachen, um das System auszunutzen, Daten zu stehlen oder das System unverfügbar zu machen. Die durch solche Angriffe verursachten Kosten steigen jährlich, daher sollte die Sicherheit von Webanwendungen eine hohe Priorität haben. Die Auswahl eines geeigneten Technologie-Stacks sollte deshalb unter Berücksichtigung von Sicherheitsaspekten erfolgen.

Diese Arbeit schlägt eine Evaluierung der heute häufig verwendeten Webanwendungs-Frameworks, Bibliotheken und automatisierten Tools vor. Diese Evaluierung zeigt verschiedene Sicherheitsstufen auf, die durch die Auswahl von Lösungen bestehend aus nativen Webframework-Funktionen in Kombination mit Bibliotheken von Drittanbietern erreicht werden können. Schließlich werden automatisierte Werkzeuge wie Static Application Security Testing Tools anhand des Grads der Unterstützung bewertet, den sie bieten, um die zuvor definierten Lösungen sicherer zu machen. Das Ergebnis dieser Bewertungen ergibt einen Überblick über die Abdeckung von Schwachstellen durch diese Lösungen und Werkzeuge.

Contents

1 Introduction

Nowadays, web-based applications have become a very common choice for companies to provide their services to their customers [1]. Data relating to customers, which is often the target of cybercriminals, needs to be stored and maintained by these web applications. By exploiting vulnerabilities, which are security weaknesses or flaws [2], attackers can access this data or cause system downtimes that lead to a high monetary loss for the company. Cybercrime has produced more than one trillion USD of cost [3]. Especially during the COVID-19 pandemic, the amount of cybercrime incidents that were recorded in the USA increased dramatically [4]. Hence, security should be a high priority for web applications, especially in the selection of components, such as frameworks or libraries, since these can have an influence on the security of web applications [5].
Traditionally, web applications were built statically in HTML and enhanced with JavaScript to perform a certain logic. The majority of repetitive tasks required to build a web application are now abstracted by web application frameworks in order to reduce the complexity during development [5]. These tasks can contain, for example, session management or input sanitization. The most commonly used client-side web frameworks are React, Vue.js and Angular [6] which produce HTML5 Single-Page Applications (SPA) that change their rendered content dynamically. In addition, different server-side web frameworks exist such as Express or Spring Boot that often present a Representational State Transfer (REST) Application Programming Interface (API) [7] to access resources from a database management system. Using these frameworks, the automation of security-related functions can be achieved, which however can also lead to security issues, if the web framework implements these incorrectly [5]. In addition to the usage of frameworks, developers often make use of third-party libraries to quicken the development process by using them to implement further recurring tasks [8]. The number of these libraries is increasing daily [9] making it harder for developers to choose an appropriate solution [8]. Similar to the web frameworks, the security of the functionalities provided by the libraries is not guaranteed and can therefore lead to security flaws. Another approach to support the security of a web application is the usage of automated tools [10]. In particular, Static Application Security Testing (SAST) tools, which recognize patterns in source code based on static analysis can be embedded into the development process to reduce vulnerabilities [10]. Finally, the inclusion

of security-related processes, as for example displayed in the Security Development Lifecycle of Microsoft [11], can further enhance the security of web applications.
A large number of vulnerabilities need to be considered when developing a web application. Different sources like the Open Web Application Security Project (OWASP) Foundation [12] or the Common Weakness Enumeration (CWE) [13] propose lists of relevant security-related considerations. Even so, a majority of the web applications are still open to attacks [14] through the vulnerabilities whose prevention mechanisms are known. Thus, a solution that helps developers write secure code by default, meaning without having security as a major concern during development, needs to be proposed. However, the solutions that can be found in the literature to this problem are not sufficient. They either evaluate web framework-specific solutions solely for the Cross-Site Scripting vulnerability and within none of the above-mentioned widely used web frameworks [5] or evaluate libraries mainly based on non security-related characteristics [8], [9], [15], [16]. Moreover, comparisons of SAST tools do exist but do not include any evaluation of the constraints they provide [17].
Therefore, this thesis is providing in-depth insights into framework-native or library-provided functions that help to deal with common vulnerabilities of web applications [12], [13], [18]. The above-mentioned frameworks were chosen to be investigated because of their high popularity among developers [6]. They will be evaluated based on the security they provide according to recommendations by sources like the OWASP Foundation, the National Institute of Standards and Technology (NIST) and further literature. Secure solutions containing web framework-native functionalities and libraries will be presented and further enhanced through the evaluation of the constraints commonly used SAST tools provide. Additionally, an overview of the vulnerabilities and how well they are covered through the proposed solutions will be given which will display which tasks are currently, and most likely also in the future, not coverable through frameworks or libraries.
These results can, on the one hand, be used by software developers or architects to inform themselves about vulnerabilities, their prevention mechanisms, and which solutions exist for them. On the other hand, this thesis is enhancing the current research by reviewing the ability of modern web frameworks to deal with a broad spectrum of vulnerabilities and proposing important security-related characteristics of libraries.

1.1 Research Objectives

The main object of this master's thesis is to evaluate existing solutions for web application vulnerabilities that consist out of framework-native functionalities, libraries, and automated tools. Hence, the following research questions were formulated to guide this process.

Research Question 0.1: *Which vulnerabilities are relevant for web applications?*
The first sub-research question is necessary to identify which vulnerabilities have the highest impact on web applications. The resulting list of vulnerabilities will be used as a baseline for the following research questions.

Research Question 1: *Which vulnerabilities are covered by framework-native functionalities?*
Within this research question, the first part of a mapping of solutions to vulnerabilities will be created. In order to answer this question, existing solutions within the above named frameworks for the previously defined vulnerabilities will be researched and evaluated.

Research Question 2: *How can libraries support the coverage of the vulnerabilities?*
Since the solutions proposed in the first research question can either be missing completely or do not cover the vulnerability holistically, further libraries are researched that support a web framework's native solution or generate their own solutions. Hence, the most popular libraries for certain tasks will be researched and evaluated based on the recommendations within the literature

Research Question 2.1: *How can similar libraries be compared with one other?*
As described above, a vast assortment of libraries and therefore an overlapping of functionalities exist. To deal with similar libraries, a metric will be defined on which these libraries will be evaluated to support research question 2.

Research Question 3: *Which automated tools and processes can be used to reduce vulnerabilities?*
Automated tools and processes to make solutions more secure are to be researched within this research question. To achieve this, the constraints SAST tools provide will be evaluated on how they help to cover problems of the solutions defined in research questions 1 and 2.

1.2 Research Approach

The research was conducted based on the design science research approach. This approach is proposed by Hevner and Chatterjee [19] and was chosen because it focuses on the creation and evaluation of innovative artifacts within a problem domain. The artifacts that this thesis aims to provide are solutions that can reduce web application vulnerabilities. This thesis will follow the guidelines Hevner and Chatterjee [19] propose. To establish a knowledge base, multivocal literature was researched, and to evaluate these findings expert interviews were conducted. A detailed description of the used methodology can be found in chapter 4.

1.3 Outline

The thesis is structured as follows. In chapter two, the theoretical fundamentals of vulnerabilities and the corresponding prevention techniques are described. Related work can be found in chapter three. In the fourth chapter, the research methodology will be explained including information about the literature research, the evaluation, the expert interviews, and the data collection. Different relevant vulnerabilities and their covered aspects are defined in chapter five. Chapters six and seven propose solutions through the use of frameworks, libraries, and automated tools. Chapter eight will discuss the findings and list limitations. A summary of the answers to the research questions and an outlook for further research will be given in chapter nine. Finally, the appendix contains tables about the reviewed frameworks and libraries and also the questionnaires.

2 Fundamentals

The theoretical backgrounds of the vulnerabilities, that are further evaluated within the thesis, are established in this chapter. It is divided into three groups of vulnerabilities, and tools and processes. An overview of all vulnerabilities can be found in chapter 5.

2.1 Input-based Vulnerabilities

First of all, vulnerabilities that are related to user input are presented. These include Injection, XML External Entities, Cross-Site Scripting, Insecure Deserialization, Denial of Service, Input Validation and Open Redirects and Forwards.

2.1.1 Injection

Injection attacks are possible if certain commands are created, depending on the input the system receives. Without proper handling of these inputs, an attacker can inject code into commands and thus change the intended behavior of a query or command. The most common types of such attacks are SQL injections. [20]
The following will cover existing literature concerned with SQL and NoSQL injection attacks.

2.1.1.1 SQL Injection

If a web application is vulnerable to SQL injection attacks, an attacker can manipulate the queries that are sent to the connected database [21]. By injecting malicious input into a SQL query, the query can be tampered with to create, read, update or delete data. Halfond, Viegas, Orso, *et al.* [22] listed the following mechanisms to inject data into a SQL query: through user input, cookies, server variables, or second-order injection, which describe the storage of user input that can lead to a SQL injection at a later point in time. Using these mechanisms, attackers can try to inject commands into a SQL query. If the inclusion of attacker-provided inputs is successful, different types of SQL injection attacks can be executed. These attacks include tautologies, union queries, illegal or logically incorrect queries, and more [22]. Listing 2.1 display an example of a SQL injection using a tautology. The input of a statement that always evaluates to true,

the tautology, will result in all users being returned by the query. The impact of these attacks does not only pertain to the database, but also to evading authentication [22] or accessing the underlying operating system [23].
To prevent this vulnerability, multiple countermeasures can be defined. First of all, validating input that is received can be used as a countermeasure [22]. However, as stated in [24] in order to prevent a SQL injection a developer should "prevent user-supplied input which contains malicious SQL from affecting the logic of the executed query". Therefore, only relying on input validation should not be considered sufficient and only used as a secondary level of defense. Furthermore, both sources propose the use of encoding or escaping of the input that is supplied by the user to filter out certain characters that can be interpreted as SQL commands (for example the character '). Also, they both describe similar approaches for guaranteeing that only secure values enter the query. Halfond, Viegas, Orso, *et al.* [22] proposes the positive pattern matching which can be compared to whitelisting input, while the OWASP Foundation [24] proposes an allow-list that contains the values that are allowed and under which circumstances these are chosen. The most important approaches presented by the OWASP Foundation [24] are prepared statements and stored procedures. In the former, code and data are separated from each other. The developer first prepares a query with placeholders and then later binds the data to the query. Hence, a SQL interpreter can differentiate between the predefined code and what is entered by the user. The data then is, depending on the type that is bound to the prepared statement, interpreted as a number or a text. Thus, injected code can no longer alter the query. Stored procedures on the other hand need to be used with caution because stored procedures can be created with dynamic content [22], [24]. Furthermore, many other techniques have been researched. Kumar and Pateriya [21] surveyed existing countermeasures within the literature.

```
var input = "' OR 1=1";
connection.query("SELECT * from user WHERE name = '" + input,
    function(err, rows, fields) {
    //...
  });
```

Listing 2.1: Tautology SQL Injection Attack Using MySQL in Java

2.1.1.2 NoSQL Injection

NoSQL injection attacks work similarly to SQL injection attacks. The injection of certain carefully crafted inputs can lead to the manipulation of the data query and therefore result in leakage or manipulation of data. Different types of NoSQL injection attacks were defined by Ron, Shulman-Peleg, and Puzanov [25]: Tautologies, using

the `$ne` operators. Union queries, in which the `$or` operator is passed with the input. These attacks allow the attacker to add an alternative constraint, and therefore change the behavior of the query. Furthermore, the injection of JavaScript code through for example the `$where` or `$mapReduce` operators is possible since NoSQL databases can execute JavaScript code [26]. Finally, piggy-backed queries, in which a second query is added to the intended query, can be executed. In addition to the attacks proposed by the authors, [27] proposes a vulnerability through the `$where` operator in which classical SQL injections are made possible by defining a `WHERE` clause.
Most of these attacks make use of the operators which are defined by NoSQL databases such as MongoDB. These operators include helpers for comparisons like `$gt` for greater than or `$ne` for not equals, as well as operators to help evaluating the query like `$where` (see [28] for more operators). The first set of operators can also be used as a payload to achieve a tautology or a union-query attack. If we consider a PUT REST endpoint as displayed in Listing 2.3, an attacker could set the payload in such a way, that it would cause the query to return all data within the collection. The payload, which is sent in the body of the request, must contain a further JSON object. This object contains an operator as key and specific data as value. The data must be chosen in such a way, that it almost always evaluates to true in combination with the operator. Hence as shown in the example, the operator `$ne` as key, with `" "` as value evaluates to true if the title is not equal to `" "`. Consequently, every document in the collection, whose title is not empty, will be returned by the `find` method. Furthermore, the `$where` operator, can be used to pass JavaScript code to MongoDB to change the query results. It gives the possibility to write code such as found in the second example of Listing 2.3. Since this input accepts JavaScript as input, an attacker can also manipulate the query to receive all documents of the collection. To do so, the attacker needs to make a specific request, which has similarities with SQL injections. The request can for example include a tautology as seen in the example. Thus, the query results to true for every document. For further examples please refer to [25].
Ron, Shulman-Peleg, and Puzanov [25] propose the usage of well-validated sanitization libraries and also scanning for injection vulnerabilities with SAST or DAST tools is recommended. Spiegel [29] proposed the use of typecasting and dynamic code analysis. In addition, to deal with JavaScript-based injections, the execution of such scripts can be disabled in for example MongoDB.

```
collection.find({'a': 3}).toArray(function(err, docs) {
    //docs = All documents with attribute a = 3
});
```

Listing 2.2: Usage of *mongodb*'s `find` Method in Express

```
//Query operators:
// PUT Request with body = {"title" : "{"$ne": " "}"}
app.post('/operatorInjectionUnsanitized', function (req, res) {
  let title = req.body.title
  collection.find({title: title}).toArray(function(error,docs) {
    //Business Logic - Find all elements with the title and change a field
    //Return all edited docs
    res.send(docs)
  });
});

//$where operator:
//PUT Request with body = {"title" : " ' || 'a'=='a"}
app.put('/whereInjectionUnsanitized', function (req, res) {
  let title = req.body.title
  let unsanitizedQuery = { $where: `this.title == '${title}'` }
  mytestcoll.find(unsanitizedQuery).toArray(function(erree,docs) {
    //Business Logic - Find all elements with the title and change a field
    //Return all edited docs
    res.send(docs)
  });
});
```

Listing 2.3: Operator Injection in MongoDB Using Express

2.1.2 XML External Entities (XXE)

XXE attacks are possible if an XML parser allows references to external entities that for example come from user input. An external entity is defined within an XML document type definition with for example the following code statement: `<!ENTITY err "An error occurred">`. After the external entity has been defined, it can be referenced within the document by starting with a `&` character and ending with the `;` character. In the example above, the entity is referenced by adding `&err;` to the document and the text will be displayed. The `SYSTEM` keyword offers the possibility to refer to an URL that defined the location where the value of the entity can be found. A possible attack could now include certain locations, such as secret files, that can be loaded through this entity. This could lead to the leakage of sensitive information such as passwords or configurations. Furthermore, the entities can be referenced within the declaration of

another entity which can lead to a popular XXE attack that is called the 'Billion Laughs' attack. The recursive referencing of entities results in the expansion of used storage and CPU usage and therefore denying the service for other users. The payload of such an attack can be seen in Listing 2.4. [30]
The detection and prevention mechanisms proposed by Hogue [30] include the usage of tools such as Burp Suite to test the application for the XXE vulnerability; the usage of intrusion detection systems that recognize the usage of `<!DOCTYPE>` within user input and reducing the damage of a possible attack by reducing the CPU power a parser can use and limiting its access to the actual application. However, the OWASP Foundation [31] states that the "safest way to prevent XXE is always to disable DTDs (External Entities) completely". In case this is not possible, external entities must be disabled, which is often depending on the underlying parser [31].

```
<!DOCTYPE root [
<!ELEMENT root ANY>
<!ENTITY LOL "lol">
<!ENTITY LOL1 "&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;">
<!ENTITY LOL2 "&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;
    &LOL1;">
...
<!ENTITY LOL9 "&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;
    &LOL8;">
]>
<root>&LOL9;</root>
```

Listing 2.4: Billion Laughs Attack

2.1.3 Cross-Site Scripting

The embedding of malicious scripts into the code rendered and executed by the browser is called Cross-Site Scripting (XSS) attack. Attackers use the lack of validation on the client- and/or the server-side to embed code on the client-side of other users. This code can for example if they are not secured in other ways, access cookies, or other personal information. The injected input is not limited to JavaScript and therefore opens an enormous attack surface and can also lay the foundation for further attacks [32]. Additionally, another possibility to execute injected code opens up if the web application uses the input of users inside of URLs. If for example the attribute `href` of `<a>` (for further examples, see [33]) takes input from users, and attackers can enter URLs using the `javascript:` scheme. This scheme makes it possible to write code within the URL [34] and consequently be used to perform an XSS attack. Another

possibility to execute code is enabled if user input is inserted into style sheets. For example, the usage of the `url` CSS function can lead to XSS attacks through the URLs described above.
Furthermore, different types of XSS attacks exist [35]. The stored XSS attack is an attack in which input that was previously injected by an attacker is received from the server by a user. This attack can take place if the server does not validate input before saving it to its database. The reflected XSS attack, on the other hand, does not need the server to store the input in the database but relies on the server to return the input directly, like through an error message which is containing the input. These types of attacks can also be mitigated through server-side prevention techniques [36]. The third type differs from the others because it does not need interaction with the server. It is called DOM-based XSS attack [37] (DOM standing for Document Object Model). This attack relies on the website to write data on the DOM without securing it. This type is especially important for the HTML5 Single Page Applications that are created through the reviewed frameworks since they only consist of one page that changes the elements in the DOM in case a new page is rendered.
The OWASP Foundation [33] proposes a variety of different rules to prevent XSS attacks. In general, no untrusted data should be put into places that can execute JavaScript code. Therefore, data that is put inside of an HTML document, into HTML attributes, between script tags, inside of CSS, or into a URL should be encoded in a way to make it secure. Moreover, HTML can be sanitized, which removes potentially malicious content, and URLs that contain the `:javascript` scheme should be avoided out of the reasons seen above. In addition, another assortment of rules can be found solely for DOM-based XSS attacks [38]. To conclude, data that is added to the DOM, via for example the `innerHTML` method, should be made trustworthy.

2.1.4 Insecure Deserialization

In, for example, Java, insecure deserialization is possible, because within Java bespoke serialization and deserialization functionalities can be defined. To do so a developer has first of all to implement the `Serializable` interface. This interface offers the functionality to implement the `readObject` and `writeObject` methods, that can be used to customize the serialization and deserialization process. This process can be triggered through the usage of `ObjectInputStream` or `ObjectOutputStream`. In this case, an object is converted to byte streams and converted back. Objects, that were serialized through Java, can be easily recognized through their first few characters. In case the serialized Java object is in the hexadecimal format it starts with "aced" and if it is Base64 encoded it starts with "rO0" [39]. Furthermore, the configuration `application/x-java-serialized-object` of the `Content-type` header exists which

also indicates serialized Java objects. Since determining whether a serialized object is being used by the application is very straightforward, an attacker can make use of this. When untrusted data is deserialized multiple vulnerabilities arise. First of all, a serialized object that is not enhanced with any integrity checks can be tampered with. Thus, attackers can for example add further access rights to their accounts. Furthermore, the deserialization process is executed before any type casting is applied. This means that in the case of this code `(myObject)ois.readObject();`, the `ObjectInputStream` (here only "ios") will read and deserialize the object before it is casted to a specific type. Therefore, an attacker could create any Java object that is contained in the classpath and in case the execution of methods is bound to the construction of the object, these methods will be called. Such an example is shown in Listing 2.5 in which a class is defined that can be used to exploit the said deserialization process of Java. If this class is deserialized, the `readObject` method is called, and therefore the command the object was serialized with is executed. As described above, the type casting is done afterward, and in this case, resulting in an error. Even so, the code will already have been executed. Notably, the attack could only succeed because the `DeserializationExploit` class was within the classpath of the Java project. Different CVEs describe current real-world examples of this type of vulnerability. An example can for example be seen in CVE-2021-3160. Additionally, the OWASP Foundation [40] displays an example, in which the deserialization of a specific object can also lead to a denial of service attack.
In order to prevent such an attack, the OWASP Foundation [12] proposes the following requirements:

1. Objects that are serialized should either be encrypted or use other integrity checks.
2. Strict type constraints should be considered before creating the object. This should not be used as the only layer of defense.
3. The deserialization code should be run on an environment that only has minimal privileges.
4. The network requests going out from a unit that does deserialization should be monitored.
5. The deserialization process should be monitored.

```java
public class DeserializationExploit implements Serializable {

    private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException{
        in.defaultReadObject();
        Runtime.getRuntime().exec(command);
    }
}
```

Listing 2.5: Exploiting Java Deserialization

2.1.5 Denial Of Service (DoS)

A denial of service attack describes the malicious act of an attacker to hinder a service from working correctly, denying the service to legitimate users. The attack itself can include the flooding of the service with network traffic or the saturation of resources such as CPU or RAM. For example, an attacker can establish a large number of TCP requests, in which the handshake is not finished by the attacker, blocking all ports accessible by the service and thus preventing legitimate users from connecting to the server. In addition, in case an attacker uses multiple clients to attack a service, the attack is called distributed denial-of-service attack. Through this, certain rate-limiting functionalities can be avoided by attackers. [41]
Mainly three different types of DoS attacks can be named [42]. First, the volume-based attacks in which a high volume of content is sent to a service. It is measured in bits per second. Protocol attacks in which attackers misuse protocols such as the above described attack called SYN flood (also called layer three and four attacks). Here, the magnitude is defined by the packets that are sent per second. Finally, application-layer attacks (also called layer 7 attacks) exist that make use of HTTP by either attacking it with slow HTTP attacks, flooding the service with GET or POST requests, or crafting certain requests to exploit vulnerabilities of the service. The requests that are sent per second define the magnitude of the attack.
In order to protect a service from a DoS attack, the literature proposes the usage of filer mechanisms [43], [44]. Different techniques such as ingress/egress, hop-count, and history-based filtering can be used to filter malicious from legitimate traffic. The majority of these techniques have in common that they form a protective layer over the services so that the malicious content cannot enter them in the first place. Furthermore, the OWASP Foundation [45] proposes further prevention techniques. These include input validation (see subsection 6.1.6), access control (subsection 6.2.2), rate limiting, and the usage of commercial cloud filter services.

2.1.6 Input Validation

Input validation is defining certain restrictions for a variable and testing the user-provided input against these restrictions. The restrictions can for example contain the data type, its size, a valid range and the content of the data [46]. Shar and Tan [47] propose additional input validation methods such as `Null`, `Containment`, `Match` and `Regex-match`. In [48] two strategies for input validation are defined. The syntactic validation enforces that the data has the correct syntax like a price in Euro being in the format `123.456,78 €`. The second one is the semantic validation strategy, which is concerned with the values of the data being correct given the context of the application. This means that for the use case of an online auction website, users cannot bid negative values. In order to be compliant with these strategies, some of the restrictions above should be used.
Preventing this vulnerability will add a second layer of security to the prevention of different other vulnerabilities like Injection, XSS, or XXE. However, the validation of input should not be used as a single line of defense. [24]

2.1.7 Open Redirects And Forwards

When the web application opens a new external URL within its user's browsers, it is redirecting the users. In contrast, if the URL is internal, meaning within the own web application, it is forwarding the users. These technically identical processes are declared open when they are done through user manipulable parameters. If the web application does not do any kind of validation of the untrusted data, an attacker could make use of this and perform phishing attacks. An example for such an open redirect would be the following URL: `http://myApp.com/?redirect=http://evilApp.com`. In case the data that is passed to the `redirect` parameter is provided with malicious intent, legitimate users could be redirected to the given website while thinking they were opening an URL to the original website. Hence, this vulnerability is offering a baseline for phishing attacks. The heuristic defined by Shue, Kalafut, and Gupta [49] to find a redirect within an URL consists of searching for a second usage of `http://` or `https://`. Wang and Wu [50] extend this heuristic by adding four further types of redirects. These firstly include a redirect without an additional HTTP prefix like `http://myApp.de/?redirect=evilApp.de`. Further, redirects that are connected to the authentication or login process of the web application are considered. After the successful login, a web application often redirects the user back to the point where the authentication became necessary. Finally, the usage of other delimiters such as _ and the inclusion of multiple HTTP prefixes within one URL are also included.
Furthermore, different mitigation strategies were proposed by Wang and Wu [50] and

the OWASP Foundation [51]. First of all, the authors agree on avoiding open redirects and forwards or at least not using user input within them as a simple countermeasure. Moreover, both sources recommend the implementation of an interrupting page, which tells the users that they will leave the website. Furthermore, the usage of validation should be considered. Both propose the usage of whitelisting, Wang and Wu [50] also make use of hashing in their approach, in order to only validate a redirect if this page is intended to be redirected to. Another approach described by Wang and Wu [50] makes use of the `Referrer` header (see subsection 2.3.2) by only accepting redirects where the header is present and referring to the own web application. Finally, in the case of forwarding, it should be checked whether the user is authorized or not to do this request [51].

2.2 Permission- and Access-based Vulnerabilities

This section establishes a baseline of different authentication- and authorization-related vulnerabilities. The fundamentals of the vulnerabilities Broken Authentication, Broken Access Control and Cross-Site Request Forgery are listed.

2.2.1 Broken Authentication

The following will deal with the needed basics for the subcategories of this vulnerability. The fundamentals of password policies, password hashing, general authentication, and session management will be set out.

2.2.1.1 Password Policy

A password policy defines different limitations for passwords and therefore has an impact on their creation process. Limitations, such as only accepting passwords with a minimum length of six characters, can be chosen to enhance the security of an organization. [52]
Password policies are defined by different sources within the literature [53]–[56]. Since most of the proposed properties highly overlap, the definition of the OWASP Foundation [57] will be chosen as point of reference since it is based on [55] and [56]. Table 2.1 enumerates the different properties that a secure password policy must have, as stated by the author. In addition to the individual properties, a column was added that displays which of these properties are either realizable through a library ('Library' in the table) or require further engineering ('Engineering'), like for example that users can change their passwords. In addition, 'Library/Visualization' describes that a task can be accomplished through a library but it needs to be embedded into the user interface.

#	Requirement	Realization through
1	Password length at least 12 and no longer than 128 characters	Library
2	No password truncation is performed	Library
3	All printable Unicode characters should be permitted	Library
4	Users are able to set a different password	Engineering
5	Setting another password requires both new and old password	Engineering
6	Password is checked against a list of common passwords, including common words or sequential characters	Library
7	Password strength meter is provided as support	Library/Visualization
8	No password composition rules such as special characters or numbers	Library
9	Rotation of passwords and requirements based on password history are not present	Engineering
10	Browsers should allow to paste content into password fields	Configuration
11	Users can either unmask a password temporarily or view the character that was typed lastly	Visualization

Table 2.1: Password Policy Requirements Based on the OWASP Foundation [57]

2.2.1.2 Password Hashing

Grassi, Fenton, Newton, *et al.* [56] emphasize that, in order to store passwords, these must not be vulnerable to offline attacks. Therefore, the authors recommend the usage of a key derivation function that creates a hash of the password by passing the password, a salt, and a cost factor to the function. Salting is the usage of an additional randomly generated string that is concatenated to the password. This can be applied as prevention from attacks that make use of pre-computed hashes or rainbow tables. The work factor has an impact on the computation time of the hashing function. It influences the number of hashing iterations. In addition, the OWASP Foundation [58] suggests the usage of an additional pepper as further defense, which is a hard-coded salt on the client-side. Grassi, Fenton, Newton, *et al.* [56] recommend the usage of Password-based Key Derivation Function 2 (PBKDF2) with a secure one-way function such as SHA256. However, the OWASP Foundation [58] suggests the usage of Argon2id, a variant of the algorithm Argon2, which is the winner of the Password Hashing Competition which was held to establish a new standard in password hashing. Further alternatives

mentioned in [58] are bcrypt, as a direct alternative for Argon2id, or scrypt, which is recommended for legacy systems.
A further point with high relevance is the appropriate configuration of these algorithms, whose recommendations can be found in the following. The algorithms Argon2 and bcrypt create their own salt, while both PBKDF2 and scrypt require a salt. According to [56], the length should be at least 32 bits (4 bytes). During the research, a length of 16 bytes was found to be very common.

- **Argon2id:** minimum memory size = 15/37 MiB, number of iterations = 2/1 and degree of parallelism = 1 [58]
- **bcrypt:** work factor = 10 [58]
- **PBKDF2:** Iterations > 10.000 [56]; SHA1 = 720.000, SHA256 = 310.000, SHA512 = 120.000 Iterations [58]
- **scrypt:** cost = 64 MiB, blocksize = 8, degree of parallelism = 1 [58] (More configurations can be found in the source)

2.2.1.3 Authentication

In the following subsection, security concerns of the authentication methods JSON Web Token (JWT) and session cookies will be evaluated. JWTs are a standard, defined in RFC 7519. A JWT consists of the three components header, payload, and signature. The token itself is made out of three Base64 encoded strings, which represent the three components, concatenated with dots. To ensure the integrity of the token, the third string (the signature) contains the hash of the first two strings, which are the Base64 encoded header and payload. The header includes the used algorithm for the signing process and the payload contains the actual data of the JWT. Session cookies, on the other hand, make use of the HTTP Cookie headers (RFC6265) that make it possible to store data such as sessions. Therefore, session identifiers, which are managed on the server-side and have an association to a specific user, are commonly stored within cookies. These cookies are when sent with every request and used for authentication.

JWT

If JWTs are used for authentication multiple security concerns could arise. Firstly, the JWT standard does not exclude the 'none' algorithm. Thus, an attacker can forge a JWT which carries the data `alg:"none"` within its header. In case the validation process of a library is influenced by the `alg` key, that is present in the header of the token, the attacker can now dodge the verification mechanism of this specific library. Furthermore,

exploiting the same security issue, it is possible to forge a valid request if asymmetric encryption like RS256 is used. In this case, the verification method of the JWT requires the token that is to be verified, as well as the public key to decrypt the token. An attacker can now forge a JWT that changes the `alg` key from RS256 to HS256 and sign it with the public key used as secret. Accordingly, the verification method, if its behavior is changed through the passed `alg` key, will now treat the second parameter, which is the public key, as the secret to verify the token. Since the attacker signed it with the public key, which is in this case misused as a secret, the verification will succeed. To prevent these issues, the used algorithm should be defined and tokens that do not use this algorithm should be omitted. [59]
Secondly, the JWT standard does not implement a revocation mechanism. Hence, once issued, a token will be valid until its expiration date was exceeded. The OWASP Foundation [60] proposed the approach of introducing a block list, which includes a JWT once a user has logged out. However, this comes with the responsibility of managing this block list.
Thirdly, an attacker can make use of a JWT, that was gained through intercepting or even stealing it. The attacker can then make requests to the system being identified as the user the token belongs to. This is especially concerning because JWTs cannot be revoked, as seen in the second concern. Therefore, the JWT should contain a fingerprint of the user which is also included in a secure cookie as a prevention technique. Only requests which contain a cookie that stores this fingerprint of the user, additionally to the JWT, should be validated successfully. [60]
Fourthly, storing the JWT securely is important. Different concerns are bound to the three storage approaches and therefore no uniform opinion was found within the literature. If the JWT is stored within the browser's storage, such as either the local or the session storage, the web application needs JavaScript to not be disabled to read the contents of these storages. Therefore, XSS attacks are theoretically made possible. The approach of using the session storage is recommended by the OWASP Foundation [60]. Furthermore, storing the JWT inside of a cookie is also a possibility recommended by grey literature [61], [62]. The web application then becomes vulnerable to CSRF attacks, because the cookie, and therefore the token, is sent with every request. In conclusion, both approaches need further security mechanisms, such as XSS or CSRF protection, to build a secure solution.
Finally, both Sebastian Peyrott [59] and the OWASP Foundation [60] emphasize that choosing a secure HMAC secret is necessary. Therefore, RFC 7518 describes that the key should have at least the same length as the hash created by the used algorithm. This means that the SHA256 algorithm should use a 256 bits strong key since it produces a hash with 256 bits of length.

Session Cookies

This paragraph will deal with the security of using cookies to store session identifiers. Session management will be discussed in subsubsection 2.2.1.4. If session cookies are used, the OWASP Foundation [63] proposes various security features to harden a session cookie. Firstly, the attributes `HttpOnly`, `Secure` and `SameSite` should be set. The reason for this is to avoid XSS attacks, leakage of information or CSRF attacks. The OWASP Foundation [63] also recommends setting `Domain` and `Path` as restrictive as possible. Finally, restricting the storage duration of the cookie is necessary, as stated by the author. Therefore, the author recommends to set either `Max-Age` or `Expires`.

2.2.1.4 Session Management

Improper session management can lead to broken authentication. The OWASP Foundation [63] proposes guidance to implement session management securely. The proposed properties are the name of the identifier, the length of the ID, its entropy, and the value the identifier has. The name of the identifier should be renamed to something generic to avoid exposing used frameworks. Furthermore, in order to secure the unpredictability of the session identifier, the OWASP Foundation [63] suggests an entropy of at least 64 bits and as stated in the source "this value is estimated to be half the length of the session ID". Therefore, the session ID should have a length of at least 128 bits. Finally, the value that the identifier carries must not include any personal information.
A common attack on the session management of a web application is session fixation. This attack is possible if the session management accepts session IDs that are generated by a user. If this is done, an attacker could set up a so-called fixed session in which the session ID victims use to generate their session is provided by an attacker. Hence, after the user is authenticated with this ID, the attacker can use it and make requests with the identity of the victim. Disallowing user-provided session IDs, securing already existing session IDs (as seen above), and the implementation of a session lifecycle is recommended. [64]

2.2.2 Broken Access Control

This subsection contains information about the subcategories contained in Broken Access Control. These are general authorization and Cross-Origin Resource Sharing.

2.2.2.1 Authorization

Authorization in general describes the need to have certain permissions in order to carry out a specific action. These access controls are used by web applications to restrict

functionalities to users with the correct privileges. These restrictions can be defined through different methodologies like role-based access control (RBAC), attribute-based access control (ABAC), and many more. Since the first two were found often in existing solutions they will be evaluated further in the following discussion. [65]
A commonly used authorization mechanism is RBAC, in which every user has at least one role that is used for granting permissions. Therefore, instead of defining the permissions separately for every user, a user is given a role with the required permissions. These roles can also be ordered hierarchically. [66]
Moreover, in contrast to RBAC, the restrictions of ABAC apply to a user depending on the different attributes the user provides. Thus, a user's permissions are not evaluated based on a predefined role, but a set of rules consisting of a Boolean combination of different attributes. [67]

2.2.2.2 Cross-Origin Resource Sharing (CORS)

The CORS standard defines a set of HTTP headers that can be sent by the server to allow cross-origin requests. Through the `Access-Control-Allow-Origin` header a server can define which origins can access certain resources. The default, which is achieved by not setting the header, is that only requests coming from their own origin will be accepted (based on the same-origin policy of the browsers). This means that in general the implementation of CORS should only be considered if it is necessary. To improve the security, Mozilla [68] proposes that "they should be locked down to as few origins and resources as is needed for proper function".

2.2.3 Cross-Site Request Forgery

In order to protect a web page against Cross-Site Request Forgery (CSRF or XSRF) various approaches are recommended in [68]–[71].
The first approach makes use of certain HTTP headers to determine the source of the request. This can be done via the `Referer` or the `Origin` header, which both contain the URL where the request is coming from. While the former header can contain the complete URL, depending on the privacy settings on the webpage the request is originated from (see subsubsection 2.3.2.3), the latter will only contain scheme, hostname, and port. If these headers are contained within the request, a developer can validate the request's source and handle it if the source is valid. If these headers are not contained, blocking is recommended by OWASP. However, this has the downside of blocking 1-2% of traffic as described in [69] and also disallowing users from accessing the page if they make a request from a website with a restricted `Referer` header.
Furthermore, forcing the user to interact with the website before making a request

can be used to mitigate CSRF attacks [69], [70]. The website can for example force re-authentication or solving a CAPTCHA. If this is done, requests which are forged by attackers are not accepted, because no user interaction is included. It should be mentioned that this approach influences the usability of the website by interrupting the user. Therefore, it is recommended to use this only for critical functions.
As written by Mozilla [68]: "the most common and transparent method of CSRF mitigation is through the use of anti-CSRF tokens" which is the third approach. It offers different types of implementation which are the Synchronizer Token Pattern, the Encryption Based Token Pattern, the HMAC Based Token Pattern and the Double Submit Cookie method [69]. The first pattern, the Synchronizer Token Pattern, stores tokens similar to session IDs. These tokens should be unpredictable and therefore creating them with a secure method is recommended. This token is added to a cookie, read on the client-side, and added to the HTTP headers of further requests or a hidden field of a form. On the server, the sent token is then validated and the request is permitted if the tokens within the header and the storage of the server match. The Encryption Based Token Pattern, on the other hand, uses encryption. A unique key is used to generate a token with the user's session ID and also a timestamp. This token is passed back to the server in the same manner as before, and then it is validated by decrypting it with the same key. Hence, no storage of the token is needed but the secret key needs to be managed. Furthermore, the HMAC Based Token Pattern can be chosen. This pattern uses HMAC with hashing algorithms like SHA-256 to encrypt data. Firstly, a token is created by inserting the session ID and the current timestamp into the HMAC. A secret key is used to create the hash. The client then includes this token, as well as the request's time, in the request via the above-mentioned methods. To validate these tokens, the timestamp from the token of the request, as well as the session ID, are used to re-generate the token. If the token matches and the timestamp, that is sent with the request does not exceed the time of expiration, the request is permitted. Finally, if no state is to be stored on the server, the Double Submit Cookie method can be used. Which, instead of sending one cookie, sends two cookies one containing session information and the other containing the CSRF token. The client then also adds the CSRF token to a hidden field or header and sends this data to the server. The server then checks if the value from the header and the cookie match, without saving the token in a database.
Finally, the web application should make use of different cookie attributes to enhance its security. First, setting the cookie to `Secure` prevents sending cookies in non HTTPS requests. Furthermore, the `SameSite` attribute helps to enhance security. This attribute determines whether the cookie of a website should be sent to in all contexts (none), only within a first-party context (strict), or only when doing top-level navigation with secure HTTP methods (lax). This feature is enabled in most major browsers, except Internet

Explorer, and also defaults to `lax` in Chrome, Edge, and Opera. If this attribute is set to `strict`, for example changing from the URL `subdomain1.myApp.com` to the URL `subdomain1.myApp.com` will result in no cookies being sent with the request. If `lax` is chosen on the other hand, the cookies will be send with the request if the request is a safe HTTP method such as `GET`, `HEAD`, `OPTIONS` or `TRACE`. However, different methods of exploitation exist if `lax` is used. First, an attacker can open new windows (`iframe`) or use the top-level navigation to send the cookies with the request. Secondly, if the web application uses prerendering, this can be used to create requests that will count as same-site and therefore contain the cookie. In conclusion, these attributes help to make cookies more robust, but should not be used as the sole protection method. The implementation of the above-mentioned tokens is highly recommended.
Furthermore, it should be mentioned that all of the security mechanisms above can be bypassed, if an attacker can successfully apply an XSS attack. Since most of the patterns described above read data from the cookies, a successful XSS attack could do the same. Consequently, attackers could forge requests containing the cookie's data and therefore execute CSRF attacks.

2.3 Configuration-based Vulnerabilities

The third group of vulnerabilities covers vulnerabilities concerned with the configuration of web applications. This includes Sensitive Data Exposure, Security Misconfiguration, Using Components with Known Vulnerabilities and Insufficient Logging and Monitoring.

2.3.1 Sensitive Data Exposure

This subsection will cover the protection of sensitive information. This includes protecting data and also the management of cryptographic keys.

2.3.1.1 Protection of Data

Data that is transported between, for example, the client and the server needs to be secured to make it confidential. If this is not the case, man in the middle attacks can lead to the leakage of data. Therefore, as proposed in [12], the header `Strict-Transport-Security` should be used to enforce communication over HTTPS (see subsubsection 2.3.2.3 for more information).
Once the data is at rest, sensitive or personal data should be encrypted or in certain cases also has to be encrypted depending on the applicable data regulation such as the General Data Protection Regulation. According to the OWASP Foundation [72],

the algorithm AES with a key length of 128 bits, or if the system allows it 256 bits, should be used for symmetric encryption. The difference in the cipher key length defines the number of cycles that are done by the algorithm, resulting in more security if 256 bits is chosen over 128 bits [73]. The block size of AES is however always 128 bits [73]. In addition, the choice of a secure cipher mode, which determines how the blocks of a block cipher are put together, is also emphasized to be important by [72]. The recommended cipher modes, by [72] and [74], are GCM and CCM since these are authenticated modes. Also, an initialization vector is needed depending on the cipher mode. This is randomly generated data that is created for every en- and decryption process [75]. It ensures that, two equal inputs will not generate the same output. Its length should be the same as the block length of the cipher (for AES 128 bits) since it is, again depending on the chosen cipher mode, added during the en- or decryption using an XOR operation. Finally, when using asymmetric encryption, the OWASP Foundation [72] recommends the usage of Curve25519 or RSA with a key length of 2048 bits.

2.3.1.2 Key Management

Different keys exist within a web application. These keys can either be secret keys, used for symmetric encryption or HMAC, or public and private key pairs that are used for asymmetric cryptography [76]. They have to be generated, distributed, stored and rotated [72]. When generating a key, the key strength depends on the used cryptographic algorithm. See [77] for guidance on how to select a secure key for a specific algorithm. The distribution should only be done through secure channels [72]. Once generated, the key should be stored securely. The OWASP Foundation [72] recommends storing a key inside of a physical or virtual Security Module or in key vaults (for example Azure Key Vault). In case a key leaked, reached its cryptoperiod (see [77]) or was used to encrypt a certain amount of data (depending on the length of the key) it has to be rotated [72]. Old keys should be encrypted with newer ones and stored in order to access data that was encrypted by the old key.

2.3.2 Security Misconfiguration

In the following fundamentals that are concerned with the vulnerability Security Misconfiguration will be disclosed.

2.3.2.1 Security Hardening

Mourad, Laverdiere, and Debbabi [78] define security hardening as: "any process, methodology, product or combination thereof that is used to add security functionalities and/or remove vulnerabilities or prevent their exploitation in existing software".

Therefore, a broad spectrum of tasks is included in the activity of security hardening. The OWASP Foundation [12] describes certain tasks such as the build process, the testing for unnecessary features, the patching of necessary features, and the segmentation between separate components. The build process should be automated in order to enable repeatable secure deploys of an application throughout the software development lifecycle [57]. This includes the creation of configurations that are recommended for the technology that is used, the setting of flags to prevent buffer overflows, and having administrators that guarantee the integrity of said configurations. Furthermore, all components should be up to date, unnecessary components should be removed and necessary elements should come from a trusted source and should be inventoried. Lastly, error messages should not lead to the exposure of security issues, the debug mode should be disabled and the setting of secure HTTP headers should be considered.

2.3.2.2 Error Handling

An error is caused by a fault and leads to the failure of the system if the error has an influence on the service or reaches its surface [79]. Therefore, the system needs error handling, which tries to remove errors from the current state of the system [79]. If no error handling is done, the failures of the system can give attackers information about the underlying code. If for example stack traces are displayed, an attacker receives not only the information on why this error happened but also where in the system. To mitigate this problem, the OWASP Foundation [80] proposes to use a global error handler that catches every error that happened unexpectedly. Furthermore, Hsieh, Le My, Ho, *et al.* [81] propose different code smells in JavaScript error handling, which are also applicable for Java, that should be addressed. First of all, ignoring the exception by leaving the `catch` block empty should be refactored to make the system error reporting. Secondly, the errors of asynchronous callbacks should be handled within these callbacks and global error handlers should be used for the reasons described above. Thirdly, error objects should be thrown, containing more information, rather than only string, containing the error message. Finally, exception handlers should not only log the information but also introduce fallback mechanisms.

2.3.2.3 HTTP Header

The headers that HTTP provides have an impact on the security of the system. The headers described in the following are based on the OWASP Secure Headers Project [82], the first round of expert interviews and sources found in the literature [83]–[86].

First, the HTTP Strict Transport Security (HSTS) header, which forces the browser to use encrypted HTTPS for further requests, is considered. In case a user tries to access

the web application via HTTP, the browser will change this request to use HTTPS. An attacker cannot then downgrade the HTTP connection, and therefore data in transmit stays encrypted and cannot be accessed through man in the middle attacks. In order to configure it securely, the directive `max-age` should be set to an adequately high value (63072000 is recommended and 15768000 seen as minimum by Mozilla [68]) and `includeSubDomains` should be enabled to also enforce HTTPS on subdomains. Furthermore, the `preload` directive exists. This is executed based on a service of Google that maintains a list of websites for that HSTS should be preloaded. If this directive is set, the browser will check the website against a hardcoded list (which can differ depending on the browser) and hence will not use HTTP as an initial request if the website is contained in the list. In addition, as described in [87], setting the `preload` directive will be treated as a request to be included in the preload list. For further requirements to be accepted into the list, please refer to the source.
Second, the `X-Frame-Options` will be evaluated which is used to protect the web application from clickjacking by restricting the page from rendering in `<frame>`, `<iframe>`, `<embed>` or `<object>` elements. Rendering can either be completely denied, only accepted from the same origin, or a specific domain. However, because of the Content-Security-Policy (CSP), which will be covered later, and its `frame-ancestors` directive, this header is becoming obsolete. If both headers use their implementation, `X-Frame-Options` will be overwritten by the CSP. Nevertheless, the usage of this header is important because certain browsers such as Internet Explorer 11 do not yet support the CSP's `frame-ancestor` directive. The values `sameorigin` or `deny` are recommended as defenses against clickjacking.
Third, the CSP, which helps to protect the website against XSS or data injection attacks will be elaborated. This header proposes different directives, which all define a policy for a certain type of resource. For example, `script-src 'self'` defines that only the same origin is a valid source for JavaScript code. In specific cases, this CSP can disallow certain functionalities of web frameworks and thus is recommended to be set at the start of development. As stated by Expert 2, when implementing the CSP the developer should choose a whitelist approach. Therefore, setting `default-src 'none'` to generally disallow any resources and adding more policies during the development to make exceptions as needed is recommended by [68]. In addition, the `require-trusted-types-for` directive is also recommended. It tells browsers to monitor the data that is passed to functions such as `innerHTML`. Hence, a type error is thrown if no trusted type, but for example, a string is passed to such functions. By using the `createPolicy` of `trustedTypes` a new type with for example input sanitization using *DOMPurify* can be defined (see [88] for a full example). Furthermore, external sources can be used to validate the CSP. For example, the *CSP Evaluator* [89] can be used to check the configuration and receive support on how to configure it securely.

Fourth, is the `Referrer-Policy` header which is concerned with the information that is sent with the `Referer` (misspelling of the word 'referrer') header. Depending on the amount of privacy that is wanted for the `Referer` header, the following directives should be considered based on [68] (see [90] for the full list): `no-referrer` for no information, `same-origin` for referrer information only in requests to the same origin, `strict-origin` for only the URL without path (for example https://myApp.com/) or `strict-origin-when-cross-origin` for sending the full referrer within the same origin but only the URL without path to different origins. As stated by Dolnák [86], it is recommended to choose either `strict-origin` or `strict-origin-when-cross-origin`.
Fifth, a header that is supposed to help to deal with XSS attacks called `X-XSS-Protection` is examined. This header tells browsers to stop loading if an XSS attack is recognized. However, this feature was either removed (Chrome and Edge) or not implemented in the first place (Firefox) because as stated in [91]: "The XSS Auditor can introduce cross-site information leaks and mechanisms to bypass the Auditor are widely known". Consequently, setting this value to 0 ([68] does recommend setting it to 1 because the guidelines were created before the feature was removed by major browsers) is recommended, so that older browser versions do not introduce the issues described above. Furthermore, similarly to the `X-Frame-Options` header, the CSP can be used as an alternative since it can disable inline JavaScript.
The final group of headers is concerned with caching, which was mentioned to be important by Expert 2 and 3. All of the following headers will be used to reduce to possibility of information leakage through cached information. For `Cache-Control` enabling the directives `no-cache`, `no-store` and `must-revalidate` in order to validate caches with the server before using them in case caches are present, not storing caches at all, and validating caches if the resource is stale is recommended. In addition, `max-age=0` can be used to force caches to revalidate themselves after zero seconds. The next header is `Expires` which implements the same functionality as the `max-age` directive but takes an expiration date instead of seconds. Since the former is taking priority over the latter [92], setting `max-age` is sufficient. The last header is the `Pragma` header, which used for HTTP/1.0 in which `Cache-Control` is not available. Its `no-cache` directive should be set.
Further headers, that exceed the scope of the thesis, that can also be used to enhance the security of a web application, are for example: `X-Permitted-Cross-Domain-Policies`, `X-Content-Type-Options`, `Clear-Site-Data` and `Permission-Policy`.

2.3.3 Using Components With Known Vulnerabilities

The usage of components that have known vulnerabilities can lead to the system being attacked using these vulnerabilities. Therefore, scanners exist, which compare

an application's dependencies to a database of vulnerabilities. First of all, it is to be emphasized that the quality of these scanners depends heavily on the database they use. If the database is not updated once a vulnerability is found, scanners cannot detect them.
The most commonly used database for vulnerabilities of software is the Common Vulnerabilities and Exposures (CVE) list. As mentioned on their website their goal is to [93]: "identify, define, and catalog publicly disclosed cybersecurity vulnerabilities". The database contains over 150.000 records that are provided with a unique identifier (for example CVE-2021-1234) containing the characters 'CVE', the year in which the vulnerability was issued, and a sequence number containing four or more digits. In addition, these records also contain a description of the issue and references. Each record deals with a specific instance of a product or system and the issues leading to a vulnerability. The focus of this database is, as mentioned above, only to catalog these vulnerabilities. Further evaluation is done through the National Vulnerability Database (NVD) which is built upon CVE's database. The data coming from CVE is evaluated by NVD based on the existing information regarding the vulnerability and the Common Vulnerability Scoring System (CVSS) v3.1 and v2.0. The result is a CVE record enhanced with a severity score and further information.

2.3.4 Insufficient Logging And Monitoring

A file that stores data about the events that take place within a system is called a log. Logs consist of entries. An entry displays all relevant information for one certain event that occurred in the system. Nowadays, logs are used to track the user's activity or to recognize attacks. Different sources can create logs, such as routers, firewalls, databases, or applications. [94]
Kent and Souppaya [94] propose different types of information which should be logged within applications. First of all, the requests and responses sent between client and server. As stated by the authors, the information can be used to understand the actions and their sequence done by an authenticated user. Furthermore, information regarding user accounts is declared to be important. This includes authentication processes, regardless of failure or success, the usage of certain privileges, or changes within the account. This information can help to prevent automated attacks against the authentication of an application. Next is the information about the usage of the application. The authors include data such as the number of transactions that are done in a certain amount of time and also their size. Finally, actions with a significant impact on operation should be logged. These are starting or shutting down the application, crashes, or changes within its configuration. In addition, further events are defined by the OWASP Foundation [95]. Those are errors in the validation of input or output,

authorization errors, failures within session management, errors of the application such as runtime errors, or the usage of functions with a higher level of risk such as access to administrative functions. Further literature supports the findings described above [96], [97].
Furthermore, the data that is included in an entry is discussed. The OWASP Foundation [95] proposes that the data contained within one entry should answer the questions when, where, who, and what. However, Marty [97] further adds the why question and Chuvakin and Peterson [96] goes even further by adding a how question. To answer the when question all authors agree on recommending a timestamp with the time zone. The where question defines in which part or component of the application the log originates from. Especially for a web application, this can include URL entry points and the used HTTP method [95]. In the who questions the statements of the authors do not align since Chuvakin and Peterson [96] proposes the usage of session identifiers which is emphasized not to be used by the OWASP Foundation [95]. A user identifier should be chosen like the username or other distinctive attributes. The what is also defined differently by the authors. Chavan and Meshram [98] define it as the object, meaning for example the resource, that is causing the log, while the others define its type of event with a given severity (defined as an additional priority attribute by [98]) such as 'warn', 'error', 'crit' or similar. Both Chuvakin and Peterson [96] and [97] answer the why question by simply giving a reason why this log was created. The OWASP Foundation [95] proposes this in a list of additional data that could be considered. Finally, the how question is answered with the action [98], such as the reason for this request, which is also included in the list of additional data in [95].
More important aspects of logging are the following: no sensitive data such as passwords or source code should be logged [95], [96], a logging syntax should be defined to have standardized logs [97], the logs should be protected when they are at rest or in transit [94], [95] and the logs should be monitored in order to respond to certain incidents.
Kent and Souppaya [94] present guidelines on how logs should be analyzed. At the start, an understanding of the logs should be established. The author suggests doing regular reviews of logs to gather a better understanding which can then lead to automation of the analyzing process. Next, the logs should be prioritized based on characteristics such as the entry type, the source of the log, or the frequency of the entry. Finally, log entries that are of interest should be responded to by initiating a corresponding process in the organization.

2.4 Automated Tools and Processes

This section deals with the fundamentals of automated tools and processes that are used to enhance the security of web applications.

2.4.1 Automated Tools

Many tools exist to test the security of a system. A survey that was conducted by Felderer, Büchler, Johns, *et al.* [10] breaks these tools down into multiple categories which are model-based testing, code-based testing and static analysis, penetration testing and dynamic analysis, and regression testing. In order to not exceed the scope of this thesis, the focus is put on static and dynamic analysis and in particular, tools that can execute these tasks in an automated manner. A third category of tools, so called Interactive Application Security Testing [99] tools, which combine SAST and DAST approaches are not considered in the thesis.

The static analysis of code describes the process of source code evaluation without executing it, and therefore a white-box testing technique [10], to search for mistakes. Such mistakes can include unreachable code and undeclared variables but also potential vulnerabilities leaving the system open to attacks through for example injection or XSS. These errors are found through the search of predefined patterns that could lead to the said errors. [100]
When the analyzers described above are used to find vulnerabilities and evaluate the security of an application, it is called Static Application Security Testing (SAST) [101]. These tools are often used early, during the development of software [11], [101] and hence can help to find security issues at the start of the software development lifecycle [10]. After comparing the incoming source or byte code against a predefined set of rules [100], the result of such an analysis with a SAST tool is a list of code blocks that could lead to possible vulnerabilities [10]. This list can include false positives or false warnings, which are pieces of code that were classified as vulnerabilities from the SAST tools but cannot be attacked [100]. Finally, since the set of rules determines the results, missing rules can therefore lead to false negatives [10]. A generally positive attitude towards SAST tools from developers is discernible in the literature [17].

Dynamic analysis, on the other hand, requires the application to be running and does not access the source code. Thus, it is a black-box testing technique [10]. Through the usage of for example HTTP requests, so-called Dynamic Application Security Testing (DAST) tools try to find vulnerabilities within a web application [102]. These tools contain vulnerability scanners [103], that send predefined payloads to the interfaces of the system, dynamic taint analyzers, which search for insecure handling of data, and fuzzing tools, that input random data into the system [10]. Since these tools are used to verify the absence of vulnerabilities they are commonly used after the development of a system [11], [102]. The testing is done directly on a running instance of the application and therefore successful exploitations are directly viewable resulting in a low amount of false positives.

2.4.2 Processes

In order to reduce possible vulnerabilities in web applications even further, various guidelines for secure software development exist. The Security Development Lifecycle (SDL) [11], the Building Security in Maturity Model (BSIMM) [104], and the OWASP Software Assurance Maturity Model (SAMM) [105] can be chosen as guidelines. All of these guidelines include training in security, the establishment of security requirements, the clarification of possible threats, verification through security testing and code reviews or penetration testing, and also the management of secure deployment. Developers should consider introducing one of these guidelines in their software development lifecycle.

3 Related Work

Web Application Vulnerabilities: Various sources set out web application vulnerabilities and how to deal with them [2], [12], [13], [18], [98], [106], [107]. Their proposed countermeasures do however often not include specific guidance for the secure implementation of these countermeasures. Thus, software engineers or architects do not have concrete guidance regarding the used frameworks and libraries that is proposed within this thesis.

Web Framework Evaluation: An evaluation of web framework-native functionalities for the prevention of XSS attacks was proposed by Weinberger, Saxena, Akhawe, *et al.* [5], [108]. They analyzed frameworks based on their built-in XSS sanitization functionality. The results showed that during that time, only half of the reviewed frameworks even implemented an automatic sanitization progress. Moreover, the very recent paper from Peguero [109] evaluates the security of client- and server-side JavaScript frameworks. However, only the XSS and CSRF vulnerabilities were considered again. Their research does not, or only partially, include the frameworks covered in this thesis. In addition, they only gathered different web applications and tested how many of them use common solutions for these vulnerabilities. Thus, no evaluation of the solutions themselves was conducted. Finally, further literature regarding the comparison of web frameworks exists but is only concerned with the performance and quality of web frameworks [110]–[112]. Only Delcev and Draskovic [112] considered security but only to a low extent and again only evaluating aspects that are related to the XSS vulnerability and for none of the frameworks evaluated in this thesis.

Library Evaluation: Different literature can be found that deals with characteristics of libraries [8], [9], [15], [16]. Since Larios Vargas, Aniche, Treude, *et al.* [8] mostly include the other sources, it can be seen as the most comprehensive source. Through the execution of both expert interviews and a survey, the characteristics they propose can be seen as highly validated. However, the security of a library is only covered shallowly and therefore further security-related characteristics are proposed in this thesis.

4 Methodology

This chapter will disclose the methods used to answer the research questions within this thesis. To achieve this, the following sections will cover the different methods used to collect information.

4.1 Literature

To support the solutions, that are to be defined, with theory, literature was researched. This research includes literature from sources such as Google Scholar, IEEE Explore, and Springer. Also, the inclusion of blogs, framework and library documentations, and other grey literature was necessary to receive a greater understanding of the solutions frameworks and libraries provide and to gather security issues connected to these solutions. Therefore, also the general Google search engine was added to search for grey literature. Furthermore, publications of the OWASP Foundation as well as NIST were used as knowledge sources because of their concrete and pragmatic security-related guidance. The guidelines of Garousi, Felderer, and Mäntylä [113] were used to conduct the research.
The keywords used as search strings varied depending on the context. For the research of vulnerabilities the keywords "Web Application" and "Vulnerability" were used in combination with one of the following keywords: "Assessment", "Classification", "List" or "Statistic". The keywords "Software Library" in combination with "Metric" or "Selection" were used to gather information about relevant library characteristics. To find solutions for a given vulnerability, the vulnerability was searched together with the keywords "Detection", "Prevention", "Attacks", and the names of the frameworks. For the evaluation of SAST tools, the keyword "SAST" was for example searched in combination with "Evaluation".

4.2 Evaluation

The solutions that are provided by libraries and frameworks were evaluated based on the information that was found within the literature (see chapter 2) and provided by the experts (see the corresponding sections in chapter 6). An overview of the evaluated

frameworks and also the libraries, that were covered more in detail can be found in Appendix A. Furthermore, to develop a greater understanding of the proposed solutions, code examples were created. These code examples are used to test certain functionalities like for example escaping mechanisms, error handling, logging, and more. They offer a simple example of the functionality that is to be tested. Where these examples contain relevant information about the functionality within chapter 6, this is mentioned within the evaluation. They can be found under the following URL `https://github.com/moritzhuether/mastersthesis`.
Moreover, a similar approach was chosen to evaluate the tools. Literature was researched and the tools were also tested with the above-mentioned code examples. This means that examples, in which vulnerabilities were intentionally placed, were used to evaluate whether the tools find these vulnerabilities or not. In both evaluated tools, Sonarqube and LGTM, the built-in constraints were evaluated.

4.3 Expert Interviews

To support the findings of the literature, data was collected through semi-structured expert interviews. These interviews were planned, conducted, and evaluated according to Gläser and Laudel [114]. The interviews were divided into two groups. The first group was concerned with research questions 0.1, 1, 2, and 2.1. The questions of the second group were focused on research question 3.
For the questionnaires please see Appendix B. The demographic questionnaire was used in both rounds of interviews.

4.3.1 First Interview Group

The first questionnaire was planned using the concurrent embedded research design defined by Creswell and Creswell [115]. Therefore, not only qualitative data was gathered, but also quantitative while having a higher focus on the qualitative questions. The quantitative questions acted as filter questions, which were asked at the beginning of each subject. The experts had to choose on a Likert scale, going from one to five, how familiar they are with firstly the vulnerabilities that exist for web applications and secondly with the OWASP top 10 web application security risks [12] in detail. This was done to evaluate the quality of the results from a specific expert, in order to prioritize answers of experts with higher stated knowledge in case of conflicting answers. However, throughout the interviews, the pattern was recognized that experts, who gave answers of higher quality, tended to also declare their knowledge in a specific vulnerability to be lower than experts that gave answers of lower quality. In

addition, conflicting answers could only be recorded for a few answers, therefore this measurement was omitted for the detailed questions about the OWASP top ten [12]. The interviews were conducted to gather currently relevant vulnerabilities, important characteristics of libraries, and solutions, consisting of framework-native or library-provided functionalities, to deal with the vulnerabilities proposed by OWASP [12]. To reduce the time frame of the interviews, the detailed questions about vulnerabilities were limited to the OWASP list [12].
The selected interviewees were all employees within msg systems AG. Employees that have points of contact with security were considered for the interview. Since rather specific knowledge was asked within the questionnaire, the contacted individuals received the three main questions that were to be asked during the interview beforehand. Only employees that stated that they could answer these questions were questioned. In total four experts were interviewed, two working as security advisors and penetration testers, and two working as developers and architects. For further information regarding the experts, please refer to Table 4.1. The interview time was initially set to 60 minutes. However, this was not sufficient, since the first interview was not finished. Therefore, the next interviews were scheduled for 90 minutes. One expert (Expert 1) did unfortunately not respond to further requests to finish the interview. Consequently, only the questions until the Cross-Site Scripting vulnerability were taken into consideration.

4.3.2 Second Interview Group

This group was not conducted using a mixed-method but by doing only qualitative research for the reasons described above. Therefore, no additional quantitative questions were asked. The main goal of this round of interviews was to generate knowledge about tools or processes that are used in the software development lifecycle. Thus, experts were questioned that experienced the usage of such tools or processes when working in a software project. The questionnaire was divided into two thematic blocks. The first block covered the tools with questions about what tools are used, where they are located (for example in Git), who maintains them, and what problems they solve. The second block, on the other hand, focused on different activities that are done to enhance security during the development. Again, questions regarding the processes in general, when they are executed, and who is responsible for them, and what they solve were asked.
Again, employees of the msg systems AG were questioned. In contrast to the first group, the experts were selected after they stated that they have worked in projects in which said tools or processes were applied. Therefore, direct knowledge in the area of security was not necessarily required. Detailed information about the experts can be found in Table 4.1. The experts of the second interview round are marked accordingly.

Expert #	**Interview Round**	**Profession**	**Years of experience**	**Points of contact with security**
Expert 1	1	Lead IT Consultant	Senior IT Consultant	Penetration Tester
Expert 2	1/2	IT Consultant	2 Penetration Testing/ 5 Software Engineering	Penetration Tester
Expert 3	1	Lead IT Architect	16 Architect	Development of Web Apps
Expert 4	1/2	Lead IT Consultant	5 Architecture/ 20 Software Engineering	Development of Web Apps
Expert 5	2	Senior IT Consultant	8 Architecture/Planning	Planning of Security
Expert 6	2	Senior IT Consultant	5 Architecture	General Development
Expert 7	2	Principal IT Consultant	7 IT Consultant	Application Security
Expert 8	2	Lead IT Consultant	25 Software Development	Front-End Development/ Privacy Committee

Table 4.1: Demographics of the Interviewed Experts

The time frame for this interview group was set to 30 minutes since fewer areas were covered by the second questionnaire than in the first questionnaire.

4.4 Data Collection

In order to compare libraries with similar functionalities, data about them was collected. This section will therefore deal with the different approaches that were elaborated to generate a metric to compare similar libraries. The following will first evaluate the results of the expert interviews, and further name the defined characteristics and how they are gathered and used in the thesis.

4.4.1 Expert Interviews

To find out which criteria influence the process of selecting libraries, the interviewed experts were asked multiple questions.
In the first question, Expert 1 to 4 stated that popularity is important to them. However, Expert 2 and 3 also stated that popularity is not the most important criterion for them. A further important criterion, that was mentioned by the experts, is how vulnerabilities or issues are handled by the maintainers. This includes the reactions to issues and the time till they are resolved (Expert 4), current vulnerabilities and how the library maintainers handle them (Expert 2), and the general history of issues (Expert 1). The last release and the update rate of the library are valued highly by Expert 4. In addition, Expert 2 is interested in the initial release of the library. Both Expert 1 and 2 favor libraries that are open source so that source code reviews can be conducted. Expert 3

states that the main criterion is that the library needs to fit the purpose. To validate the security of the library the experts proposed different options. First, Expert 1, 2, and 3 communicated that they would review the source code of a library if this is necessary. As Expert 4 stated, this process is often skipped to be more cost-efficient at the start of the project. Furthermore, if the library is more popular, Experts 1 and 2 both hope that someone else already checked the security of the library. On the other hand, Expert 4 only relies on a penetration test of a proof of concept of applications to validate their security.
In the second question, the proposed security criteria were to be validated. The first security criteria, 'Security by Default', was emphasized to be a good indicator by all experts. In contrast, 'Customizability' did mostly receive criticism. Experts 1 to 3 stated that the customizability of a security-related library can also open up room for failure by misconfiguration. Therefore, as stated by Expert 1, the library should rather be safe by default than offering a high level of customizability. 'Communication' was also assessed to be a good criterion. 'State of the Art' was however again treated more carefully. Expert 2 said that the newer approaches do not generally offer a better solution and that rather "battle-proven" solutions should be chosen. Expert 4 supports this statement by mentioning that already existing solutions should be chosen.

4.4.2 Library Characteristics

In this subsection, characteristics are defined and approaches to gather them will be evaluated.

4.4.2.1 General Characteristics

In order to evaluate a library further, literature was researched [8], [9], [15], [16] and experts were questioned as seen above. A mapping was created to gather the most important characteristics. The ones that were stated the most and which properties of them are important can be seen in Table 4.2. During the research, different approaches were pursued to generate a metric to evaluate libraries with these characteristics. These approaches can be seen in the following discussion.

Existing Solutions

Existing solutions were researched that evaluate the characteristics of a library to receive a comparable score. Possible candidates were either npmjs' library evaluation or the evaluation of npms.io. The built-in evaluation of npmjs was originally based on npms.io (as is still written in the npmjs documentation) but as stated in [116] npmjs no longer makes use of npms.io. The scoring is divided into popularity, quality, and maintenance,

Popularity	Downloads
	Stars
	Used by other Projects
Active Maintenance	Issue Response Rate
	Issue Response Time
	Issue Closing Time
	Issue Coverage
	Last Update
	Contributor
	Recent Commit
Maturity and Stability	First Release
	Release Frequency
	Issues per Release
Community Activeness	StackOverflow Questions
	Google Trends

Table 4.2: Library Characteristics

but the concrete implementation of the calculations and the used data could not be found. Furthermore, during testing, it was noticed that npmjs scores the maintenance of almost every library found at a value of 33%. Accordingly, this measurement was not used. npms.io, on the other hand, is an open source project whose source code is available. It produces scores regarding the popularity, maintenance, and quality of a library. A total score calculated out of the mean of the three scores is also available. The popularity evaluation contains stars, forks, downloads, contributors, dependents, subscribers, and the download acceleration. For the maintenance, the coverage of issues, the closing time of issues, the recent commit, and the commit frequency are considered. These values do already cover most of the characteristics defined in Table 4.2. Nevertheless, the quality score is also available and considers the availability of a README, a stable version, outdated dependencies, badges, and more. The Github project behind npm.io, called *npms-analyzer*, starts with gathering data through the usage of so-called `Observers`. They add new libraries to the database and update older ones with more recent data. Using this data the scoring process is started. Within its aggregation state, the minimum, maximum, and mean values are calculated for all libraries that are in the database. Using this aggregation, a Bezier Curve is created which is then used to create the scores of the libraries. Unfortunately, the library is not free from issues. Similar to npmjs, the maintenance score does not work consistently. The score is for the majority of libraries close to or even 100%. Furthermore, the popularity score is also influenced by the stars, forks, contributors, and subscribers a library has. This can at a first glance seem like an inconsistency because, for example, the library *react* with currently 10,3M weekly downloads has a popularity score of 94% while *escape-html* with 17M weekly downloads only has a score of 64%. However, the reason for this is a value called `communityInterest` which is defined through the attributes above. The big gap in the popularity score is explained through *react* having a `communityInterest` value of 209.423, while *escape-html*'s community interest value is only at 435. No problems could be identified with the quality score.

Prototype

In an attempt to overcome the problems described above a prototype was developed which gathers data related to the maintenance-related characteristics. Hence, a JavaScript program was written which queries data from the APIs of npmjs, Github, npms.io, and StackOverflow. The gathered data represents the characteristics that were found. The results of this script can be seen in Figure 4.1.
Different problems arose with this implementation. First of all, the gathering of the different properties of Community Activeness did not generate any useful results. Generic names such as of the library *xss* led to a large number of results from Google and StackOverflow. However, most of these results did not deal with the library but the XSS vulnerability in general, falsifying the result. Moreover, data about the issue response time was infeasible to gather since not only maintainers can respond to issues. Therefore, it was not clear whether a person working on the library answered the issue or, for example, a follow-up question was asked by another user. In addition, it was not possible to determine the number of issues per release since the issues are not necessarily connected to a specific release. Finally, a general problem is that one data set of a library is hard to compare with the data set of another library. The reason for this is because individual differences are difficult to evaluate since no holistic score is present as seen in the existing solutions above. The source code can also be found on Github[1].

Conclusion

To define a fitting solution, some of the mentioned approaches are used together. Since npmjs' code base could not be located, it is not used. On the other hand, npms.io's scores of a library's popularity and quality are used. The data that is considered within these scores is sufficient based on the characteristics defined above. Additionally, since no functioning maintenance score exists, the prototype is used to gather the missing

[1]`https://github.com/moritzhuether/mastersthesis/tree/main/DataCollection`

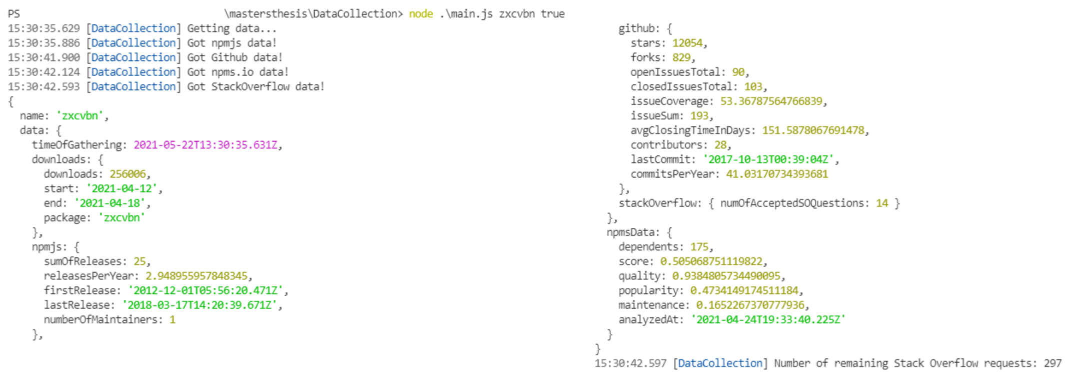
Library Name	Down.	First Rel.	Last Rel.	Rel. Freq.	Com. Freq.	Iss. Cov.	Avg Iss. c. T.	Contr.	Dep.	Qual.	Pop.
zxcvbn	256k	2012-12-01	2018-03-17	2.96 p.Y.	41.18 p.Y.	53%	151 d.p.Iss.	28	175	94%	47%

Table 4.3: Example Table of the Library Metric

data. Different properties, like the release frequency, general release data, the handling of issues and contributors were used.
Since Java often offers fewer libraries and also fewer interchangeable ones, these characteristics will only be applied to JavaScript-based libraries. To reduce the amount of data that is to be collected, only the most popular libraries will be evaluated further through the metric.
The final gathering process consists of multiple steps. First of all, the user of the script passes the library name as a first and an optional Boolean as a second parameter. Using this library name, data is gathered from npmjs.com. Additionally to the characteristics described above, the Github repository is received through the `bugs` attribute of npmjs' data. In case the data exists (some libraries do not have a Github connected), the repository is extracted and used for further requests that are directed to the Github API. After the data about stars, issues and contributors was collected or if no Github information was found, the data is requested from npms.io. Finally, if the optionally passed Boolean parameter is true, data from StackOverflow is requested. Here, questions with accepted answers regarding a certain library, by querying for its name, are gathered. However, as mentioned above, this is not used further in the thesis and therefore, if no second parameter is passed, the data will not be gathered. Figure 4.2 displays this flow as a state diagram.
The results of this evaluation are displayed in a table. An example of such a table, containing the data of the *zxcvbn* library, can be seen in Table 4.3. The displayed values, from left to right, are the name of the library, its weekly downloads from 12th of April to 18th of April (one specific time frame was chosen for comparability), the first and last release date, the releases per year ('Release Frequency'), the commits per year ('Commit Frequency'), the coverage of issues (closes issues divided by all issues), the average closing time of the issues in days per issue, the number of contributors and dependents, and finally, npms.io's quality and popularity score.

4.4.2.2 Security Characteristics

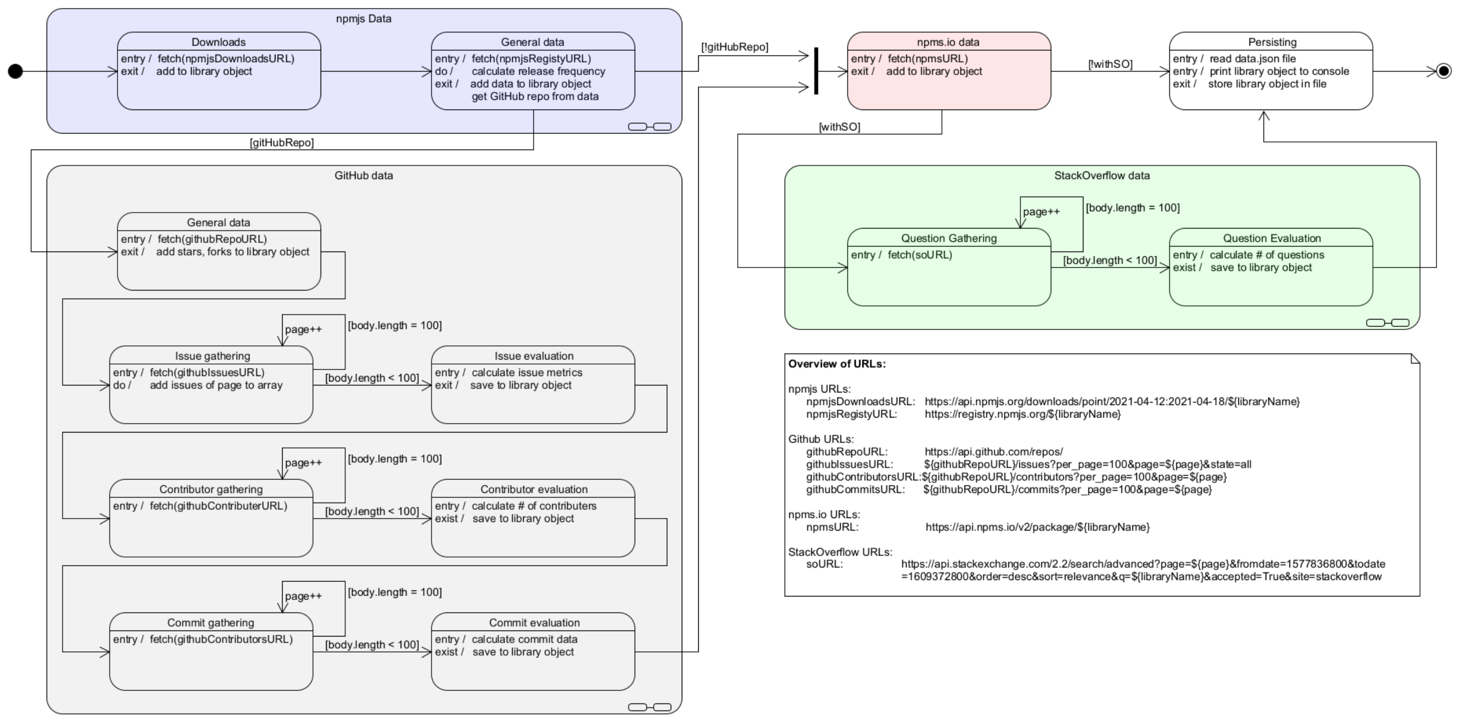
Since the security of the libraries was only evaluated rather one-sidedly in the literature [8], further security-related characteristics of libraries that implement security-related functionalities are proposed in the thesis. These are Security by Default, Customizability, Communication, and State of the Art. The first describes that the standard configuration of a library is already configured securely. The second describes that



```
PS                              \mastersthesis\DataCollection> node .\main.js zxcvbn true
15:30:35.629 [DataCollection] Getting data...
15:30:35.886 [DataCollection] Got npmjs data!
15:30:41.900 [DataCollection] Got Github data!
15:30:42.124 [DataCollection] Got npms.io data!
15:30:42.593 [DataCollection] Got StackOverflow data!
{
  name: 'zxcvbn',
  data: {
    timeOfGathering: 2021-05-22T13:30:35.631Z,
    downloads: {
      downloads: 256006,
      start: '2021-04-12',
      end: '2021-04-18',
      package: 'zxcvbn'
    },
    npmjs: {
      sumOfReleases: 25,
      releasesPerYear: 2.948955957848345,
      firstRelease: '2012-12-01T05:56:20.471Z',
      lastRelease: '2018-03-17T14:20:39.671Z',
      numberOfMaintainers: 1
    },
    github: {
      stars: 12054,
      forks: 829,
      openIssuesTotal: 90,
      closedIssuesTotal: 103,
      issueCoverage: 53.36787564766839,
      issueSum: 193,
      avgClosingTimeInDays: 151.5878067691478,
      contributors: 28,
      lastCommit: '2017-10-13T00:39:04Z',
      commitsPerYear: 41.03170734393681
    },
    stackOverflow: { numOfAcceptedSOQuestions: 14 }
  },
  npmsData: {
    dependents: 175,
    score: 0.505068751119822,
    quality: 0.9384805734490095,
    popularity: 0.4734149174511184,
    maintenance: 0.1652267370777936,
    analyzedAt: '2021-04-24T19:33:40.225Z'
  }
}
15:30:42.597 [DataCollection] Number of remaining Stack Overflow requests: 297
```

Figure 4.1: Data Collection Script - Result

a library's functionalities offer to apply different configurations to enhance security. Communication is related to the documentation and how it communicates the security mechanisms or security concerns that come with this library. State of the Art means that the library's security mechanisms are up to date with what is proposed in the literature. The results of the expert interviews (see subsection 4.4.1) resulted in refactoring of the characteristics. The characteristic that received the most praise is Security by Default. Therefore, if the libraries offer certain configuration possibilities, it will be evaluated whether these are secure by default. This evaluation, on the other hand, depends on the last characteristic, State of the Art. Here literature is researched to determine a secure configuration (this can be found in chapter 2). However, Customizability received the most criticism and is therefore omitted. Communication will be described if the documentation is referring to security-related issues or concerns that arise when using this library. Since these characteristics are often highly interconnected, they will not be stated explicitly in chapter 6 but covered within the library evaluations.


npmjs Data
Downloads
entry / fetch(npmjsDownloadsURL)
exit / add to library object
General data
entry / fetch(npmjsRegistyURL)
do / calculate release frequency
exit / add data to library object
get GitHub repo from data
[!gitHubRepo]
[gitHubRepo]
npms.io data
entry / fetch(npmsURL)
exit / add to library object
[!withSO]
[withSO]
Persisting
entry / read data.json file
entry / print library object to console
exit / store library object in file
GitHub data
General data
entry / fetch(githubRepoURL)
exit / add stars, forks to library object
page++
[body.length = 100]
Issue gathering
entry / fetch(githubIssuesURL)
do / add issues of page to array
[body.length < 100]
Issue evaluation
entry / calculate issue metrics
exit / save to library object
page++
[body.length = 100]
Contributor gathering
entry / fetch(githubContributerURL)
[body.length < 100]
Contributor evaluation
entry / calculate # of contributers
exist / save to library object
page++
[body.length = 100]
Commit gathering
entry / fetch(githubContributorsURL)
[body.length < 100]
Commit evaluation
entry / calculate commit data
exist / save to library object
StackOverflow data
page++
[body.length = 100]
Question Gathering
entry / fetch(soURL)
[body.length < 100]
Question Evaluation
entry / calculate # of questions
exist / save to library object

Overview of URLs:

```
npmjs URLs:
    npmjsDownloadsURL:   https://api.npmjs.org/downloads/point/2021-04-12:2021-04-18/${libraryName}
    npmjsRegistyURL:     https://registry.npmjs.org/${libraryName}

Github URLs:
    githubRepoURL:          https://api.github.com/repos/
    githubIssuesURL:        ${githubRepoURL}/issues?per_page=100&page=${page}&state=all
    githubContributorsURL:${githubRepoURL}/contributors?per_page=100&page=${page}
    githubCommitsURL:       ${githubRepoURL}/commits?per_page=100&page=${page}

npms.io URLs:
    npmsURL:                https://api.npms.io/v2/package/${libraryName}

StackOverflow URLs:
    soURL:                  https://api.stackexchange.com/2.2/search/advanced?page=${page}&fromdate=1577836800&todate
                            =1609372800&order=desc&sort=relevance&q=${libraryName}&accepted=True&site=stackoverflow
```

Figure 4.2: Data Collection Script - State Diagram

5 Vulnerabilities

In the following, vulnerabilities that were found to be relevant and their subcategories are elaborated. In addition, the results of the expert interviews are displayed and the aspects of the vulnerabilities covered in this thesis are defined.

5.1 Vulnerability Mapping

Vulnerabilities were researched and evaluated to gather a list of relevant vulnerabilities. This process included literature research and expert interviews as described in chapter 4. The sources that were considered are listed in the following discussion. The OWASP top ten list of the most critical web application security risks [12] was used as a baseline. Moreover, OWASP's more recent top ten list of the most critical API security risks [117] was also taken into account. Additionally, the results found by Al-Khurafi and Al-Ahmad [118] were used to also include the top 25 most dangerous software weaknesses according to the Common Weakness Enumeration (CWE) [13] and the threat classification list of the Web Application Security Consortium (WASC) [18]. Since the version of the former list used in [118] is no longer up to date (CWE list from 2011), these results had to be adapted to the current version (CWE list from 2020). Moreover, the statistic from Positive Technologies [14] was also given consideration. Finally, the vulnerabilities mentioned by Chavan and Meshram [98] and Lepofsky [106] were also taken into account.
To evaluate important vulnerabilities, the ones found in the sources listed above were mapped against the web application security risk list of OWASP [12]. The reasons for this are that this list often appears in the literature [2], [118]–[120] and also the very positive resonance of Expert 1 to 4 regarding the relevance of the list in 2021. To generate further relevant vulnerabilities over and above the OWASP list, experts were questioned. Additionally, vulnerabilities found in the literature, that were not already covered by the OWASP list [12], were also considered. In order to limit the scope of the thesis, only vulnerabilities that were found three times within the sources above were added to the list. These vulnerabilities are Cross-Site Request Forgery, Denial of Service, Input Validation, and Open Redirects and Forwards.

The mapping of the vulnerabilities covered in this thesis to sources in which they make an appearance can be seen in Table 5.1. This table displays the 14 vulnerabilities and also which source proposed a comparable vulnerability. In case the thematic fitting was only partial, the vulnerability was put into brackets.
Furthermore, the categories 'Not considered' and 'Not relevant' exist. Within the former category, vulnerabilities are included that did not reach the threshold of three sources mentioning them. Vulnerabilities such as 'Unrestricted Upload of File with Dangerous Type' or 'Abuse of Functionality' only appeared twice and are therefore out of scope. The latter category includes vulnerabilities that are not, or only in specific cases, exploitable within web applications. These include the usage of SOAP and vulnerabilities which deal with the memory of a program (for example out-of-bounds write) and are therefore mainly exploitable in programming languages such as C. However, the web frameworks that are considered within the thesis use the programming languages JavaScript and Java and therefore these vulnerabilities do not apply. The majority of the vulnerabilities within this category originate from the Common Weakness Enumeration list since this list is not specific to web applications but to software in general.

5.2 Expert Interviews

In the beginning, all experts stated that they are familiar with the vulnerabilities of web applications. Expert 1 and 4 both replied with a 5 on the Likert scale which represents "Very familiar", while Experts 2 and 3 answered 4, which is only "Familiar".
All of the experts stated in the first question that the OWASP list [12] is still relevant in 2021. However, Expert 2 and 4 would change the order of the vulnerabilities. Expert 2 would define two groups of vulnerabilities that are injection and configuration and Expert 4 suggested moving authentication and authorization to the top of the list.
In question three it was asked to name the most relevant vulnerability out of the OWASP list [12]. Expert 1 answered with Insecure Deserialization and Security Management being the most relevant, while Injection and Cross-Site Scripting are less relevant because of the good coverage of frameworks and libraries for these vulnerabilities. Expert 3 stated that the relevance of a vulnerability depends on the use case of the applications and that therefore certain applications do not have to consider some vulnerabilities. Furthermore, as stated by Expert 4, XML External Entities is nowadays less relevant than the rest of the list.
In question four, further relevant vulnerabilities were tried to be gathered. Expert 1 mentioned cache poisoning and after being questioned about vulnerabilities such as CSRF, he declared those to not be very relevant. Moreover, Expert 2 answered with session management, which is however already covered in Broken Authentication,

law regulations regarding data protection and handling data in such a way that only necessary information is saved, which are already covered by Sensitive Data Exposure. Expert 3 did not contribute any further vulnerabilities. Vulnerabilities, mentioned by Expert 4, were the usage of components that were maliciously altered, which is already covered in Security Misconfiguration, the general complexity of modern web applications, and social engineering.

The findings show that the OWASP list of the top ten most critical web application vulnerabilities [12] is still relevant in general. However, the vulnerability XML External Entities is considered less relevant than the rest and the order of the list is not completely clear among the experts. Furthermore, only a small amount of vulnerabilities were mentioned additionally to the list proposed by the OWASP Foundation [12].

5.3 Scope Definition

The following paragraphs will further elaborate on the vulnerabilities and their subtopics that will be covered within the thesis. In case no further details are given, the vulnerability will be covered completely. Since the first ten vulnerability categories are based on OWASP's top ten list [12], this list is used as source throughout the paragraphs.

Systems are vulnerable to **Injection** if they enable an attacker to influence the execution of commands with their input. The vulnerability considers the injection into LDAP, XQuery, the Operating System, SQL, NoSQL, HTML, and many more [121]. Since the possibilities this vulnerability offers are immense, this thesis will solely focus on the aspects of SQL and NoSQL injection.
Broken Authentication deals with issues concerning the authentication of users. Firstly, proper handling of user passwords is considered. This includes implementing password policies, hashing, and storing passwords. Session management is examined in this vulnerability. Finally, authentication mechanisms such as JSON Web Token (JWT) and session cookies will be evaluated. The thesis will not evaluate automated attacks, multi-factor authentication, or further authentication methods.
Websites with the **Sensitive Data Exposure** vulnerability, do not handle sensitive information carefully. It generally deals with any data that could be considered sensitive. This includes first of all to only handle data that is necessary, the encryption of this data in transmit, the encryption of this data at rest, the management of cryptographic keys, and cryptographic algorithms in general.
In **XML External Entities** attackers exploit insecurely configured XML parsers, therefore applications which use XML can be vulnerable. This can lead to the leakage of

data or denial of service attacks.
Broken Access Control is concerned with the authorization of users. The subcategories of this vulnerability are general authentication since this is the reason for most of the named problems by the OWASP Foundation and Cross-Origin Resource Sharing.
Security Misconfiguration describes different approaches to set different configurations to secure values. The thesis will deal with the handling of errors, setting of HTTP headers as well as security hardening.
The embedding of malicious scripts into websites is called **Cross-Site Scripting (XSS)** attack. An attacker can run JavaScript code within the browsers of victims in case the web application is vulnerable. Therefore, data such as tokens or cookies can be leaked.
A website can be vulnerable to **Insecure Deserialization** if untrusted user input is serialized. The vulnerability can lead to remote code execution or denial of service attacks.
Using Components with Known Vulnerabilities is a vulnerability that is concerned with the vulnerabilities of components used in web applications through for example dependencies.
If a web application is doing **Insufficient Logging and Monitoring**, attacks cannot be recognized through the evaluation of logs. Therefore, the creation of adequate logs and their evaluation will be considered.
Cross-Site Request Forgery attacks can occur when authentication with cookies is implemented insecurely. Since cookies are sent with every request to the server, an attacker can forge a request and trick a victim on executing this request. Consequently, the attacker controlled request is authorized through the cookies of the victim.
A **Denial of Service** attack happens if an attacker successfully overloads the system so that it can no longer provide its service. The thesis will focus on application layer denial of service attacks and their prevention.
The **Input Validation** vulnerability is not concerned with filtering out malicious content from input, but with the input having the correct properties that are required by the business logic. Validating that user input is a positive number is for example be a requirement for the quantity of ordered items in an online shop. The thesis will look into different input validators and evaluate common patterns in input validation.
If a website has the **Open Redirects and Forwards** vulnerability, an attacker can make use of unsafe or not validated redirects of the website. URL parameters can be used to redirect from the attacked website to another website, which is often used for phishing attacks.

Vulnerability	**OWASP Web App.[12]**	**OWASP API [117]**	**CWE [13]**	**WASC [18]**	**Lepofsky [106]**	**Chavan [98]**	**[14] PT**
Injection	A1:2017	API8:2019	6, 10, 17	6, 12, 19, 23, 28, 29, 30, 31, 36, 39, 46	injection flaws	Content Spoofing, Injecting OS Command, SQL Injection	PT 29%
Broken Authentication	A2:2017 (API4:2019)	API2:2019	14, 18, 24	1, (11), 18, (21), 37, 47, 49	authentication, session management, access control	(Brute Force), Insufficient Authentication, Credential/Session Protection, Insufficient Session Expiration, Session Fixation	PT 45%
Sensitive Data Exposure	A3:2017	(API3:2019)	7, (20)	4, 13	(related security issues)	Information Leakage	PT 13%
XML External Entities	A4:2017		19	43, 44			PT 5%
Broken Access Control	A5:2017	API1:2019 API5:2019	16, 12, 22, 25	2, 16, 17, 33, 34, 48	unauthorized view of data	Broken Access Control, Path Traversal	PT 37%
Security Misconfiguration	A6:2017		(13)	14, 15, (45)	error handling, security misconfigurations	Improper Error Handling, Application Misconfiguration,	PT 85%
Cross-Site Scripting	A7:2017		1	8	cross-site scripting	Cross-Site Scripting	PT 53%
Insecure Deserialization	A8:2017		21				
Using Components with Known Vulnerabilities	A9:2017				(related security issues)		PT 13%
Insufficient Logging and Monitoring	A10:2017	API10:2019					
Cross-Site Request Forgery			9	9		Cross-Site Request Forgery	PT 34%
Denial of Service		(API4:2019)	23	10	denial of service	Denial of Service	
Input Validation			3	20	input validation		
Open Redirects and Forwards				38	redirects and forwards		PT 16% Open Redirect
Not considered[a]		API6:2019 API9:2019	11, 15	3, 5, (22), 24, 25, 26, 27, 40, 41, 42		Abuse of Functionality, Buffer Overflow	PT 34 % User Interface Misrepresentation PT 16% Unrestricted File Upload PT 11% SSRF
Not relevant			(2, 4, 5, 8)[b]	7 [b] , (32, 35)[c]			

Table 5.1: Vulnerability Mapping

[a]Did not appear at least three times within the sources and is therefore to specific and out of scope for this thesis
[b]Not applicable for Java or JavaScript
[c]SOAP is nowadays only used rarely in web applications

6 Mapping of Vulnerabilities to Solutions

In this chapter, different solutions consisting of framework-native functionalities and libraries are evaluated. The chapter is again structured according to the vulnerability groups and deals first with the results of the expert interviews. Further, it covers the solutions native to frameworks and the ones of libraries separately for every subtopic in the corresponding vulnerability. The results of the expert interviews are only available for the vulnerabilities taken from the OWASP top ten list [12]. Furthermore, whenever a library is mentioned its name ist be written in italic. In the case of a library for Node.js, the name of `npmjs.com` is used and for Java, the name found on `mvnrepository.com` is used.

6.1 Input-based Vulnerabilities

This section sets out solutions for the input-based vulnerabilities.

6.1.1 Injection

In this section, solutions of frameworks and libraries for the protection from injection attacks on different database management systems (DBMS) are disclosed. To cover the most common use cases, two of the most popular SQL and NoSQL DBMSs were chosen. Based on the statistics of [122], [123], these DBMSs are MySQL and MongoDB. MySQL is a relational DBMS that was released in 1995. MongoDB is a NoSQL database management program that uses document store. Its initial release was in 2009.
Since the database is in most cases connected to the server, only the server-side frameworks are considered. Example implementations were created that show the query possibilities for Express[1] and Spring Boot[2].

[1] `https://github.com/moritzhuether/mastersthesis/tree/main/Express/01-Injection`
[2] `https://github.com/moritzhuether/mastersthesis/blob/main/SpringBoot/demo/src/main/java/main/demo/Injection.java`

6.1.1.1 Expert Interviews

Within the first group of interviews, multiple questions concerning the SQL and NoSQL injection vulnerability were asked.
The first question did not result in a clear result, since Expert 1 and 2 responded that there is no difference in security when choosing direct APIs, object relational mapper (ORM), or object document mapper (ODM) and query builders. Expert 3, on the other hand, stated to use ORMs or ODMs. Expert 4 emphasizes that especially in an environment with lower experienced developers higher-level libraries like ORM/ODM or query builder should be chosen to "reduce the risks that other developers possibly disregard".
In question two, Expert 4 (also mentioned by Expert 2 in "Further Aspects") proposed the development of an interface that can be used by inexperienced developers. The interface denies direct access to the functionalities of the library and only offers self-implemented functions that use only secure methods.
The answers of all experts of question three align with each other. They recommend the usage of tools in general and in particular static code analysis tools (Expert 1 and 4), linter (Expert 4), architecture metric tools (Expert 3), and coding policies (Expert 4).
General recommendations from the experts regarding injection vulnerabilities were to test the application and check where certain characters can be misinterpreted.

6.1.1.2 Frameworks

Express does not offer any native implementations to connect to the two DBMSs. Hence, it will not be considered in the following.

MySQL

In Spring Boot data can be queried via different APIs. The APIs covered are Java Database Connectivity (JDBC) and Java Persistence API (JPA). Since these APIs are also used when communicating with other DBMS such as Oracle, the following findings are also applicable for those DBMS.
First of all, it is also possible to connect to a MySQL database through the *mysql-connector-java* dependency using a `Connection` object. This makes it possible for a general Java project to connect to a database using JDBC. This object allows to send unvalidated strings as SQL query via the `executeQuery` method to the database and is therefore not safe from injection attacks. However, it also supports the usage of Java's `PreparedStatements` that should be used to prevent SQL injection. Calling the `prepareStatement` method of the `Connection` object will create a new `PreparedStatement` object which can contain placeholders within its SQL query. These

placeholders can then be set by calling the `set'Datatype'` method, where 'Datatype' defines the datatype the driver will convert the input to. In case of the method `setString` the given string value will be converted to SQL's `VARCHAR` or `LONGVARCHAR`. This secures the query process from possible injection attacks.
Moreover, it is also possible to use JPA without any Spring specific dependencies. If this is done, an `EntityManager` object has to be defined which is queried through the `createQuery` method. The same security issues as described above arise if a string containing user-provided data is passed to the method. However, there is also the option for `PreparedStatements`.
When using Spring's JDBC the *spring-boot-starter-data-jdbc* dependency is required. After connecting to the database, a query can be executed using the `JdbcTemplate` class. It implements methods that can be used to query data from the database such as `query`, `queryForObject` or `queryForList`. The latter requires a string to be passed to them which can contain unvalidated user input. Hence, SQL injections are possible. Furthermore, the `execute` method can also be used, which executes SQL queries without returning something, making them useful for the creation or deletion of tables. Similar to the methods before, the string can, if user input is inserted into it, cause unexpected behavior. In addition to this insecure `JdbcTemplate` class, Spring is providing the `NamedParameterJdbcTemplate` which can be used to execute named parameters. Thereby, the parameter is passed in a second instance causing it to not be treated as code, making the inclusion of user-provided data into the query safe.
JPA, on the other hand, functions as ORM that maps Java objects to tables within the database. To use this functionality, the *spring-boot-starter-data-jpa* dependency is required. Since this is an ORM, a corresponding `Entity` class has to be created. A class is declared as such through the `@Entity` annotation. This class defines attributes that correspond to columns that a table within the database has. Through this, queried data is already cast to the data type that is defined within the class. Different querying methods can then be defined in a custom repository class that is extending `CrudRepository`. Using the `@Query` annotation, a query can be defined. This method does not allow the manipulation of the query.

MongoDB

MongoDB is usable natively if Spring Boot is chosen as the web framework for the server and the *spring-data-mongodb* dependency is added to the project. Using a `MongoTemplate` or a `MongoRepository`, similarly as described above with JDBC and JPA, data can be queried or created. After researching for possible injections in MongoDB under Spring Boot, no security concerns were found. The research contained the review of the Spring Boot and MongoDB documentation, and Google and StackOverflow search with the search query "Spring MongoDB Injection". However, the results did not indicate any possible vulnerabilities. In addition, the in-depth testing of injections in both `MongoTemplate` and `MongoRepository` that are possible when using MongoDB under Express did also not lead to any successful attacks. Only one StackOverflow question mentioned that NoSQL injections are possible [124], but the attacks which were described in the answer were not reproducible. Thus, a further security analysis of this subject is needed which would exceed the scope of this thesis, to provide a concrete assessment of its security.

6.1.1.3 Libraries

Since Spring Boot is already providing native support for querying data with MongoDB or MySQL no further libraries could be found that support the native functionality or implement similar functionality.

MySQL

In order to connect an Express Server to a MySQL database, the *mysql* library can be used. After connecting to the database, a developer can query data using the `query` function. This method offers three different approaches to query data. The first one is a simple SQL string as the first and a callback function as the second argument. This string will be executed by the SQL interpreter and therefore, if used incorrectly, can open up the system to injection attacks. Incorrect use, in this case, means the concatenation of user input to the SQL string. The next approach makes use of placeholder values (also called prepared statements), which are escaped before they are added to the query. The third approach can be used, if advanced query options are needed. It does not add more security. The documentation of the library emphasizes that [125]: "In order to avoid SQL Injection attacks, you should always escape any user provided data before using it inside a SQL query". Therefore, developers should be aware of potential security threats when using the first approach without further escaping user input. Moreover, the library also gives examples of how certain value types will behave when escaping them. In addition, the documentation also underlines, that the `NO_BACKSLASHES_ESCAPES` mode needs to be disabled. Enabling it would cause the backslash character to not be usable as an escape character anymore.
Moreover, access via ORMs is also possible. The most downloaded ORM that could be found is *sequelize* which works for MySQL, Postgres, SQLite, and more. After connecting to the MySQL database, models have to be defined. These models do, as within most ORMs, represent the database tables that are planned to be queried. Therefore, for example, a user variable needs to be created using the `define` method. Here, the data types of the columns are defined, making adding user input to a `WHERE`

clause secure. Therefore, the general procedure of model definition and model querying is secure. Nevertheless, vulnerabilities were already found that allowed a SQL injection anyway (CVE-2019-10748). Currently, no vulnerability is present. Another concern is however the existence of the so-called raw queries. With the `query` method, direct SQL queries can be executed which can contain user-supplied data and are thus vulnerable to SQL injection. These raw queries also can be used with parameter binding, which is an equivalent of stored procedures.

MongoDB

When using MongoDB in combination with Express multiple libraries are offered to the developer. These libraries are the official MongoDB Node.js library *mongodb* and an ODM called *mongoose*. While *mongoose* offers the possibility to define schemas and therefore provides more convenience when working with data than *mongodb*, both access data through the `find` method. Hence, the following security concerns are applicable for both libraries. After connecting to the database and selecting a collection, data can be queried from MongoDB. To achieve this, the `find` method needs to be called (see Listing 2.2). If user input is embedded into the query, the web application can be vulnerable to injection attacks. These attacks make use of the operators, which MongoDB implements, that were further described in the fundamentals. The first set of operators can also be used as a payload to manipulate the query. If we consider a PUT REST endpoint as displayed in Listing 2.3, an attacker could set the payload in such a way, that it would cause the query to return all data within the collection. The payload, which is sent in the body of the request, must contain a further JSON object. This object contains an operator as key and specific data as value. As described in the literature and also shown in the example, the operator `$ne` as key, with `" "` as data evaluates to true if no title is equal to `" "`. Consequently, every document in the collection, whose title is not empty, will be returned by the `find` method.
Furthermore, the `$where` operator, can be used to hand JavaScript code to MongoDB to change the query results. It gives the possibility to write code such as found in the second example of Listing 2.3. Since this function accepts JavaScript as input (the method `mapReduce` and the operators `$accumulator` and `$function` also allow JavaScript input), an attacker can manipulate the query to receive all documents of the collection. To achieve this, the attacker needs to make a specific request, which has similarities with SQL injections. The request has to include a tautology as seen in the example. Thus, the query results to true for every document.
These vulnerabilities can be dealt with, if a sanitization library is chosen. Libraries, that help with this concern, are *mongo-sanitize* and *express-mongo-sanitize* (see Table 6.1). More libraries exist, that implement similar functionality, but are not considered due to

Library Name	Down.	First Rel.	Last Rel.	Rel. Freq.	Com. Freq.	Iss. Cov.	Avg Iss. c. T.	Contr.	Dep.	Qual.	Pop.
express-mongo-sanitize	19,6k	2015-11-11	2021-01-07	1.63 p.Y.	8.36 p.Y.	88%	153.07 d.p.Iss.	4	18	93%	19%
mongo-sanitize	13,3k	2014-09-04	2020-03-02	0.44 p.Y.	2.38 p.Y.	50%	222.39 d.p.Iss.	3	15	51%	2%

Gathered: 2021-05-10

Table 6.1: MongoDB Sanitization Libraries

their generally low popularity and are only listed below. The above-mentioned popular libraries function interchangeably, by removing any key that starts with $, and therefore removing all operators that are passed to the `find` method. Nevertheless, *express-mongo-sanitize* does offer more comfort, because it can be added to the middleware of Express and thus removes the obligation to use it at every endpoint. In addition, another group of interchangeable libraries exists. These are libraries that provide a plugin for a MongoDB document schema. A plugin defines code that will be executed with every request for this specific schema. Hence, libraries define sanitization logic that can be used via the `plugin` method of a document schema. The libraries are by name: *mongoose-sanitizer-plugin*, *mongoose-sanitizer*, *mongoose-sanitize* and *mongoose-sanitize-json*. However, all of these libraries have rather low popularity with less than 120 downloads per week (2021-05-10). Consequently, the first group of libraries is recommended due to their larger community.
Further sanitization libraries, with a popularity of under 500 weekly downloads (2021-05-10) are: *mongodb-sanitize*, *mongo-escape*, *mongo-query-sanitize*, *mongo-query-filter*, and *mongoose-sanitize*.

6.1.2 XML External Entities

Solutions for the XML External Entities vulnerability can be found in the following. Code examples were created for both JavaScript[3] and Java[4]

6.1.2.1 Expert Interviews

The experts all had the same opinion about the XML External Entity vulnerability. All of them answered with either disabling the processing of external entities or not using XML in general. Expert 4 stated that other alternatives like JSON should be used, because XML is "too complex, too faulty with the meaning of being misinterpreted in the application". In addition, Expert 3 recommends completely denying the processing of XML in case it is not used mainly as data format.

[3]`https://github.com/moritzhuether/mastersthesis/blob/main/GeneralJS/xxe.js`
[4]`https://github.com/moritzhuether/mastersthesis/tree/main/GeneralJava/04-XXE/main/src`

6.1.2.2 Frameworks

The frameworks that are based on Node.js, namely React, Angular, Vue.js, and Express, do not natively support XML. However, if a developer persists in the usage of XML, an XML parser library is necessary.

Java on the other hand, and therefore Spring Boot, offers many XML parsers internally and through libraries. As described in the literature and the expert interviews, the disabling of DTDs is recommended. Depending on the internal implementation of Java that is used to parse XML data, different configurations have to be considered. The OWASP Foundation [31] collected the needed configurations for multiple implementations.

6.1.2.3 Libraries

The library *sax* is a parser that can be used for XML or HTML in Node.js. This library was chosen for evaluation because of its immense download count and because it is also used as a dependency in other XML parser libraries called *xml2js* and *xmldoc*. It implements a strict mode, which is not enabled by default and therefore needs further configuration. Setting the mode to true will force the parser to ignore any entities that are not contained in the predefined set of XML entities. The entities within this set are: `&amp;`, `&apos;`, `&gt;`, `&lt;`, and `&quot;`. Therefore, any other entities, like externally defined ones, will cause an error. A similar strict mode is also used by *xml2js*. *xmldoc* on the other hand does not implement its own strict mode but uses *sax* with its default configuration, also requiring further configuration. Many other XML parsers exist, however, the general approach should be to enable a secure mode if it is not by default.

Java is also offering different libraries to parse XML data. As described above, please refer to the OWASP Foundation [31] to receive guidance on how to disable external entities within the used parsing library.

6.1.3 Cross-Site Scripting

This subsection deals with possibilities to reduce XSS vulnerabilities in the named web frameworks. For all three client-side frameworks, React[5], Vue.js[6] and Angular[7], code examples were created to show framework-native security mechanisms. Within the

[5]https://github.com/moritzhuether/mastersthesis/blob/main/React/main/src/XSS.js
[6]https://github.com/moritzhuether/mastersthesis/blob/main/Vue/main/src/components/HelloWorld.vue
[7]https://github.com/moritzhuether/mastersthesis/tree/main/Angular/main/src/app

React example, the in the following described escaping and sanitization libraries are also displayed.

6.1.3.1 Expert Interviews

In question one, the experts all agreed that they use framework-native functions. However, Expert 2 emphasized to not use these security mechanisms on their own and to also use an additional CSP. The experts, in contrast to the literature, also stated that they do not use additional libraries to deal with XSS vulnerabilities.
As further aspects the experts mentioned always think about where user input can be at (Expert 2) and to do server-side validation (Expert 3).

6.1.3.2 Frameworks

XSS is a vulnerability that has already received a lot of attention from React, Vue.js, and Angular. These web frameworks all offer approaches to protect web applications against XSS attacks, which will be discussed in the following paragraphs. However, firstly not directly framework-specific but HTML5-specific security mechanisms will be discussed since all three client-side frameworks use HTML5.

HTML5-Specific Solutions

This paragraph will elaborate the security concerns arising through `innerHTML`. Directly binding the content to HTML by either using `innerHTML` or other methods with the same outcome, can lead to the injection of HTML and therefore also JavaScript. Fortunately, if this method is used, the `<script>` element will not be executed [126]. Hence, the injection of strings such as `<script> alert('XSS') </script>` will not result in the display of the alert. However, other inputs can be used to execute code. These inputs make use of event handler such as `onError` or `onFocus`. One possible input is an image, which has an invalid source and an error handler in which code can be executed. The input looks like this: `<img src="..." onerror="alert('XSS')">` [127].

Framework-Specific Solutions

In React, JSX is commonly used to render content. If the developer wants to include the values of an variable to the content, this can be done by using curly braces: `<h1> Welcome, {username} </h1>`. Including data in that way results in React transforming the input to a string, before rendering it. Hence, as stated in the documentation of React [128]: "It is safe to embed user input in JSX". The data that is embedded is escaped before it is rendered. Characters such as ", ', &, < and > are escaped by replacing them with their corresponding HTML entity. This is of course an effective measurement against XSS but also results in limitations. Escaping the characters < and > is not possible in certain web applications if for example rich-text support is needed. In this case, HTML elements such as `<bold>` or `<h1>` will no longer work. Nevertheless, React is also offering other methods to display data. React's `dangerouslySetInnerHTML` method, which is a replacement for the `innerHTML` method, can be used. This methods make it possible to directly change the HTML content of a `<div>` element and therefore it is not safe of injection attacks. To deal with this security concern, React firstly changed the name of the method to directly alarm developers of the security concerns connected with it. Secondly, using this method was made artificially harder by React. It does not simply use a string as a parameter but requires an object with a `__html` key that contains the actual content [129]. In consequence, using this function without being aware of the security concerns is very hard. No protection of URL or CSS injections was found.

Vue.js follows a similar approach as React. When binding data in Vue.js using a template, the content is escaped: `<h1> Welcome, {{username}}</h1>` . Just like React, it also offers a possibility to render unescaped HTML content by passing `v-html="name"` to a `<div>` element. In contrast to React, no alarming name was chosen, and also the usage of this unsafe approach was not made harder. Finally, Vue.js is also not implementing any URL sanitizer to deal with the injection of `:javascript` into URLs.

Angular, on the other hand, offers even more protection. First of all, Angular implements the security contexts of `HTML`, `Style`, `URL` and `ResourceURL`. These contexts are used if a given value is supposed to be interpreted in one of the contexts. Therefore, Angular uses the HTML security context if the developer binds data to `innerHTML` (only if this binding is done via `[innerHTML]=` inside of a HTML tag) or wants to add variables to the DOM (called interpolation in Angular: <h1> Welcome, {{username}}</h1>) [130]. This makes the use of `innerHTML` with Angular safe, while React and Vue.js suffer from the issues explained above. This functionality is implemented through the `DomSanitizer` which is provided with the context and the string. In order to evade the automatic sanitization of Angular, a developer can make use of the `bypassSecurityTrustHTML` method (methods for URL or CSS can be used analogously). This tells Angular, to trust the data coming from this source. Similar to React's `dangerouslySetInnerHTML` method, the method's name was chosen to be alarming and therefore stop developers from using it without knowing of its security concerns. Additionally, using `ElementRef` with `nativeElement` gives a developer the possibility to directly access the DOM. If this is done, Angular's sanitizers do not sanitize what is passed to the DOM. The documentation of Angular declares this feature as a security risk and also refers to its security guide [131].

Express does not offer security mechanisms that are native to the framework. Since server-side rendering is recommended by OWASP to help reduce the possibility of reflected and stored XSS attacks [36], the usage of libraries is necessary.

Spring Boot on the other hand has a native implementation of an HTML escaper within the `HtmlUtils` class. The `htmlEscape` method can be used to escape all special characters to their corresponding HTML entity.

6.1.3.3 Libraries

For the Node.js environment, different groups of libraries could be found. The most popular representatives of their group can be seen in Table 6.2. First, there is a group of libraries, that do HTML escaping. These libraries, with some exceptions, focus on the task of removing certain characters. Most commonly ", ', &, < and > are escaped. The libraries do not offer any additional functionality for the client-side frameworks, since these already provide either native support for this functionality or even sanitization. However, the libraries can be used for Express, to deal with stored or reflected XSS attacks. All three *escape-html*, *html-escaper* and *escape-goat* have a high popularity. Even so, the library *html-escaper* receives more maintenance by taking less time to resolve an issue and also receives the best quality score out of the three. Since these libraries work interchangeably, *html-escaper* should be chosen based on the better performance in the library metric. In addition, also other libraries such as *stringify-entities*, *htmlescape* or *@wordpress/escape-html* are possible choices for this group because all implement the same functionality.
The second group is on the other hand not concerned with completely disallowing HTML but with filtering out potentially malicious contents. All of the three most popular sanitizers use a whitelist approach which they also lay open within their documentation. This means they define a list of allowed HTML elements and their attributes. The previously mentioned example of the `<image>` element with a non-existing source and a corresponding malicious `onError` attribute would be harmless if a sanitization library is used. The reason for this is that within all three whitelists, the `onError` attribute is not contained and therefore removed from the string. The libraries deal also with the problem of the `:javascript` scheme injection, by either implementing a sanitizer for the `href` attribute (*xss* and *dompuritfy*), or by completely removing it (*sanitize-html*). Also, the injection of CSS is not possible since the whitelists also do not contain the `style` attribute. Tests, which contained classic XSS attacks as they are described in subsection 2.1.3, did not lead to a noticeable difference between the three libraries. Even so, since *dompurify* was explicitly mentioned by the OWASP Foundation

[33], often highlighted in other sources [132], [133] and also because it is maintained well, it is considered the preferred choice between these libraries. However, conducting more in-depth security testing of these sanitizers is necessary to give a statement about their security.
The third group, which is only concerned with URL sanitization, is needed to make potentially unsafe URLs safe to use. *@braintree/sanitize-url* is the name of the only library found in this group. It makes use of regular expressions in order to find the `:javascript` or `:data` scheme inside of URLs.
Finally, the fourth group is countering XSS vulnerabilities through the configuration of HTTP headers. See subsubsection 6.3.2.2 for further information regarding these libraries.
The usage of these libraries does of course not remove the possibility that an XSS attack against a web application implementing them is successful. These libraries can contain vulnerabilities themselves. This means that for example malicious input is not filtered correctly or benign input is removed. In addition, although the libraries have often only a minimal interface with one method that does the sanitization or escaping, it can still be misused. First of all, they must be applied to all inputs that are considered untrusted, especially inputs provided by users. If only one input is not made trustworthy, attackers can still exploit the system. Furthermore, inputs that were made trustworthy, should not be enhanced with user-provided data. In case this is done, the input becomes untrustworthy again and needs to be resanitized or reescaped. Based on these arguments, the libraries are adding an additional layer of security to the web application, but security can still not be guaranteed. Therefore, the usage of further tools is recommended to test the correct implementation of the framework's security mechanisms and used libraries.

Similar libraries do also exist when using Spring Boot. First of all, for encoding, there is the *owasp-java-encoder* library from OWASP. This library is also encoding HTML by calling the `forHTML` method of the `Encode` object. This approach is happening based on the encoding steps described in [36]. Another possible solution is Apache Commons `StringEscapeUtils` class which offers different escaping methods also including HTML. The `escapeHtml4` method should be used to include all HTML 4.0 entities.
Furthermore, the library *jsoup* exists. It offers the `clean` method, which compares the HTML against a whitelist and therefore sanitizes it. The library offers different whitelists, or `Safelists` as it is called in the library, that can be used to filter malicious content. These `Safelists` have different levels of strictness and can be used depending on the context. The developer needs to choose a list since no default option exists for the `clean` method.

Library Name	Down.	First Rel.	Last Rel.	Rel. Freq.	Com. Freq.	Iss. Cov.	Avg Iss. c. T.	Contr.	Dep.	Qual.	Pop.
escape-html	17,1M	2012-08-20	2018-03-07	0.57 p.Y.	4.12 p.Y.	100%	25 d.p.Iss.	8	976	68%	64%
html-escaper	10,7M	2015-04-08	2021-02-18	1.47 p.Y.	4.59 p.Y.	100%	0.67 d.p.Iss.	1	82	97%	51%
escape-goat	4,6M	2017-05-27	2021-04-16	1.47 p.Y.	2.52 p.Y.	100%	7.07 d.p.Iss.	8	88	83%	51%
xss	1,4M	2012-09-19	2021-05-06	7.75 p.Y.	47.07 p.Y.	75%	81.84 d.p.Iss.	29	356	97%	56%
dompurify	1,4M	2014-05-21	2021-04-28	10.32 p.Y.	198.92 p.Y.	100%	8.96 d.p.Iss.	70	453	93%	58%
sanitize-html	1,1M	2013-09-10	2021-03-19	12.25 p.Y.	65.20 p.Y.	96.55%	309.94 d.p.Iss.	60	590	96%	53%
@braintree/sanitize-url	382k	2017-08-08	2021-04-29	3.46 p.Y.	18.82 p.Y.	100%	14.18 d.p.Iss.	12	53	83%	31%

Gathered: 2021-05-10

Table 6.2: XSS-Related Libraries

6.1.4 Insecure Deserialization

To understand the solutions that exist for Insecure Deserialization, they are listed and evaluated in the following. Also, code examples were created that demonstrate an insecure deserialization attack for both JavaScript[8] and Java[9].

6.1.4.1 Expert Interviews

In the first question, none of the experts was able to name a library. Expert 2 replied with generally avoiding more complex serializations and using JSON if possible. Expert 4 said that he relies on his own implementation.
Question two also resulted in no usable answers since all experts do not have any experience with monitoring serialization errors.
A further aspect mentioned by Expert 2 was that this vulnerability has a very high impact but is hard to be exploited.

6.1.4.2 Frameworks

JavaScript mainly uses JSON as data format and offers the methods `JSON.stringify` and `JSON.parse` to deal with serialization and deserialization. In contrast to objects serialized in Java, data serialized in JSON is not in a binary format. Further, this language-agnostic format of JSON helps it to reduce the Insecure Deserialization vulnerability. The deserialization of a JSON string does not trigger the execution of any embedded methods. However, the tampering of data is also possible if a JSON string is not protected as described in the literature.
Choosing JSON over other data formats enhances the security of a web application. Nevertheless, the encryption of, for example, authorization tokens (like JWT) is still

[8]https://github.com/moritzhuether/mastersthesis/blob/main/Express/08-InsecureDeserialization/serialisation.js
[9]https://github.com/moritzhuether/mastersthesis/tree/main/GeneralJava/08-InsecureDeserialization/main/src/main

necessary. Additionally, untrusted data should preferably be omitted.

Java's deserialization implementation does not offer sufficient protection. In case no further prevention techniques are used, serialized objects should be encrypted to establish integrity of the objects. Otherwise, no data, coming from an untrusted source or that contain any input provided by users, should be deserialized. In addition, a developer can extend the deserialization process by creating a custom class that extends `ObjectInputStream`. Through this, it is possible to change the deserialization behavior to make use of the 'look ahead' serialization approach in which the class, the serialized data represents, is validated before the deserialization. Thus, only classes that are intended to be deserialized are allowed to enter the system. [134]

6.1.4.3 Libraries

The Node.js-based frameworks can make use of the *node-serialize* library. This library adds to the existing implementation of JSON the possibility to also serialize functions. However, as displayed by Chaudhary [135] it is possible to exploit this functionality. Therefore using this library is not recommended.

Spring Boot offers the possibility to use the *SerialKiller* library. This library gives a concrete implementation of the previously described 'look ahead' approach. This functions by replacing the default `ObjectInputStream` with *SerialKiller*'s implementation of it. Through the use of a configuration file, different white and blacklists can be defined. To configure the library securely, only the classes that are planned to be deserialized within the application should be put into the whitelist. In case a non-whitelisted class is tried to be deserialized the library will throw an `InvalidClassException`. However, this library can not offer any default configurations for the black- or whitelist and therefore the security is still up to the developer configuring the library. Unfortunately, it also has not received an update since 2016, thus the following library should be considered.
The library *Apache Commons IO* and its `ValidatingObjectInputStream` class can be used. This class needs to be created with a set of different classes, only allowing the deserialization of these predefined allowed classes. In contrast to *SerialKiller*, it still receives updates and does not require a complex configuration file. Therefore, its usage is recommended.

6.1.5 Denial Of Service

The solutions that could be used on an application level to prevent DoS attacks are evaluated in the following.

6.1.5.1 Frameworks

In order to deal with DoS attacks, Express offers different supported libraries. First of all, its *body-parser* library per default limits the size of the request body to 100 kilobytes. Furthermore, it offers the *connect-timeout* library which timeouts request after a certain amount of time. However, as described in the library's documentation, Node.js will still terminate the process which means that it cannot be used to prevent Dos attacks.

Using Spring Cloud Gateway, a rate limiter can be implemented. After adding it to the dependencies, the gateway can be configured to filter requests. The currently existing implementation for the Redis database implements the Token Bucket Algorithm.

6.1.5.2 Libraries

For Express, different limitation libraries can be used to support the DoS attack protection. The libraries *ddos*, *bottleneck*, *limiter*, *ratelimiter*, *express-brute*, *express-limiter*, *express-rate-limit* and *rate-limiter-flexible* all offer a similar approach by limiting the amount of requests a user can make in a given time. Furthermore, the library *safe-regex* is recommended by Express to deal with DoS attacks. It sets the amount of allowed repetitions per default to 25 and therefore disallowing to spend to much resources on the regex algorithm.

The library *Bucket4j* can also be used in Spring Boot to implement rate-limiting again using the Token Bucket Algorithm. The library defines a `Bucket` object, that uses a `Bandwidth` object to define its limits. At every request, the developer has to call the `tryConsume` method to remove tokens from the bucket. Once the bucket is empty, the method will return false and the developer has to act accordingly.

6.1.6 Input Validation

Different framework-native functionalities and libraries are evaluated based on how well they support a developer to validate user input.

6.1.6.1 Frameworks

HTML5- and framework- specific solutions for input validation are evaluated in the following.

HTML5-Specific Solutions

User input is often received via web forms. HTML5 added new types for semantic validation of data coming from an `<input>` element inside of a form. This element defines a broad assortment of types, which can be used to validate input. Listing 6.1 shows an example for an input field of an email. If this type is used, the entered text is validated based on a regular expression as seen in the listing [136]. However, this regular expression also accepts emails that do not have a top-level domain like for example `testemail@gmail`. Therefore, the `pattern` attribute can be used to validate by a different regular expression. Furthermore, attributes for the maximal length of a text (`maxlength`), for the range of a number (`min` and `max`), and determining whether the value is not null (`required`) are present. A direct implementation of the `Containment` method is not existing, but the `patterns` attribute with the correct regular expression can be used instead. In addition to the input types, a developer can also use JavaScript to validate input. The input can be validated when submitted or when changed, using restrictions implemented by the developer in JavaScript.

```
<input type="email" name="email" required>
Validates based on the following regular expression:
/^[a-zA-Z0-9.!#$%&’*+/=?^_‘{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$/
```
Listing 6.1: <input> Element of `type` Email

Framework-Specific Solutions

In React input validation can be indirectly achieved through its *prop-types* library. This library checks whether props, data that is passed between components, are of a specific type. The library, which is developed by Facebook and therefore counted as framework-native, offers the functionality to enforce certain primitive JavaScript types like string or number for the props. No further native restrictions except `required` exist.

Vue.js has no implementations other than the standard HTML5 forms and the use of self-implemented logic with JavaScript.

Angular, on the other hand, offers two different types of forms. These types are the template-driven forms and the reactive forms. In template-driven forms, it is possible to use the HTML5 types for input validation as described above. By passing `minLength="4"` to the `<input>` element, Angular uses the `MinLengthValidator` directive and therefore apply the minimum length validation. If a type is passed, it is validated in the same manner. The reactive forms, on the other hand, need the implementation of a `formControl` model. Inside this model, different validators can be assigned from the `Validator` class. These built-in validators can be used to validate the minimum length, the range of a number, and more (see [137] for the full list). In addition, both forms offer the possibility to define custom validators.

The *express-validator* is a middleware offered by Express to validate input. This library uses sanitizers and validators of the *validator* library. For more information about the implemented functions of *validator* please refer to subsubsection 6.1.6.2. These functions can be passed as parameters to for example Express' `post` method. If this is done the defined validators are used before the method is entered. The `custom` method also gives the possibility to add a custom sanitizer or validator. Also, a schema can be defined, that offers the possibility to define a set of validators and error messages for expected variables.

Spring Boot makes use of the *JSR-303 Bean Validation* specification, which defines different validators. Similar to the SLF4J (see subsubsection 6.3.4.2), this specification is offering an API so that different reference implementations can be used. As described by [138], the *Hibernate Validator* is the "de-facto standard" reference implementation for validation. Using the *javax.validation* dependency, different constraints can be added to an `Entity` class. In order to validate these constraints, a developer can add the `@Valid` annotation in front of the parameters that are passed to an endpoint of a `RestController`, what is possible through the *spring-boot-starter-validation* dependency. In that way, the constraints are validated with the validation methods defined in the *Hibernate Validator*.

6.1.6.2 Libraries

Different groups of libraries exist to validate data for Node.js-based frameworks (see Table 6.3). The first group helps with the general validation of single values. Thus, the libraries of this group define different methods to check whether a certain string follows certain restrictions that are defined in the method. The by far most popular library of this group is called *validator*. It implements a large amount validators such as `isIBAN`, `isEmail`, and many more. In addition, it also implements further sanitizers that can be used to normalize emails, black or whitelist data, or trim whitespaces. Based on the high popularity, dependents, and maintenance of this library, one can assume

that it is the standard for input validation in Node.js. Different libraries exist, which offer similar functionalities but implement fewer validators. These libraries are for example *validatorjs*, *node-input-validator* or *better-validator*. Since these validators are in comparison not popular, they are not considered. Furthermore, different libraries exist, which only implement single validators such as *is-negative-zero* or *email-validator*. Since these do not offer a holistic approach, they are also not further evaluated.
Another, even larger group of libraries, are JavaScript or JSON schema validators. These libraries define a schema in JSON containing the different restrictions for the data that is to be validated. Libraries like *ajv*, *joi* and *yup* offer such schema builders containing different validators. Since these libraries offer highly similar functionality, are all maintained well, and also popular, no clear recommendation between them can be given. The choice comes down to personal preference.

In React, a developer can choose the library *formik* which improves the convenience when working with forms in React. The validation needs to be self-defined through a *yup* JSON schema. Furthermore, the library *react-hook-form* works in the same way but offers support for different schema builder libraries. In addition to React's native *prop-types* library, Airbnb is offering the *airbnb-prop-types* library, extending the functionality of React's native solution. This gives the possibility to add ranges or predicates as validators. Finally, the library *formsy* exists which offers native validators. However, the amount of validators available is less than what can be found in the general Node.js library *validator*. Therefore, only Airbnb's prop type library could be considered, but as stated above, this does not directly affect input handling.

Vue.js offers the possibility to use *vuelidate* or *vee-validate*. Both offer many validators, to restrict the range of a number, check whether the string is an email or an URL, and define whether the attribute is required or not. Furthermore, custom validators can be implemented. Out of a security perspective, the libraries can be used interchangeably. Both libraries do offer high support for managing the presentation model and are presumably chosen for this reason.

For Angular additionally to its native solution libraries exist. These libraries are *ng2-validation*, *ngx-validate*, *angular-validation* and *ngx-validator*. They are generally rather unpopular with a maximum of 16k downloads (2021-05-31) while other, non-Angular specific solutions have more than one million downloads. Hence, these are not further evaluated.

Library Name	Down.	First Rel.	Last Rel.	Rel. Freq.	Com. Freq.	Iss. Cov.	Avg Iss. c. T.	Contr.	Dep.	Qual.	Pop.
validator	5M	2011-01-04	2021-04-20	19.99 p.Y.	186.11 p.Y.	87%	59.06 d.p.Iss.	314	3271	100%	77%
ajv	46.1M	2015-05-29	2021-05-09	56.27 p.Y.	418.91 p.Y.	91%	40.87 d.p.Iss.	133	4119	99%	85%
joi	3.3M	2012-09-16	2021-04-09	24.27 p.Y.	259.82 p.Y.	97%	47.69 d.p.Iss.	181	3911	85%	76%
yup	2M	2014-10-18	2021-03-10	15.99 p.Y.	93.29 p.Y.	81%	66.00 d.p.Iss.	120	1109	76%	65%

Gathered: 2021-05-10

Table 6.3: Validation Libraries

6.1.7 Open Redirects And Forwards

As recommended in the literature, a developer should either not redirect to any external URLs at all or only when these URLs are contained in a whitelist that is stored on the server. The maintenance and correct use of such a whitelist is generally unspecific to the underlying programming language or used frameworks and libraries and therefore completely up to the developer. However, certain libraries that are reviewed in subsection 6.1.6 implement methods to check a string against a predefined whitelist. Such a library, called *validator*, can be seen in subsubsection 6.1.6.2. In case the redirection to a user-provided page is necessary, the *validator* library can also be used to validate the URL. It disallows per default the `:javascript` scheme and thus already protects against an XSS attack through the redirection. In Spring Boot, the *Hibernate Validator* is validating a URL with the `URL` class of *java.net* which also disallows this scheme. However, it does not protect from the literature described social engineering attacks, therefore the proposed additional page, telling the users that they are about to leave the page should be implemented.

6.2 Permission- and Access-based Vulnerabilities

The upcoming subsections evaluate solutions that deal with authentication and access control.

6.2.1 Broken Authentication

This subsection deals with solutions regarding password policies and password hashing as well as general authentication through JWTs and session cookies. In addition, code examples were created for the correct configuration of the below listed password hashing algorithms[10] [11].

[10] `https://github.com/moritzhuether/mastersthesis/blob/main/GeneralJS/pwHashing.js`

[11] `https://github.com/moritzhuether/mastersthesis/blob/main/SpringBoot/demo/src/main/java/main/demo/WebSecurityConfig.java`

6.2.1.1 Expert Interviews

In the first question, only Expert 2 could name a library, which is called *zxcvbn*. Expert 3 and 4 both referred to Keycloak as their primary resource for user management and therefore no password policies are considered. This tool is however not further evaluated in this thesis since it is not a framework-native functionality or a library.
In question two again Expert 2 was able to enumerate libraries, which are *argon2*, *brcypt* and *scrypt*. Expert 2 emphasizes choosing one of these three solutions and not implementing a solution yourself. Expert 1 rated the time a hashing function needs to be important if passwords are hashed, with slow hashing functions being preferred over fast ones. Experts 3 and 4 both have a default hashing function but were not able to name it. Peppering was not done by any of the experts and is therefore not further evaluated in the thesis.
Question three showed that multi-factor authentication should be considered if financial or personal data is accessed.
The fourth question gave insights into the different methods of authentication. Expert 1 stated that all methods displayed in the question (JWT, session cookies, OAuth) can be secure if implemented correctly. Expert 2 prefers the implementation of an existing session library or the usage of OAuth rather than JWT. The stated reason for this is missing features in JWTs such as invalidation. Both, Expert 3 and 4, emphasized to always use libraries or using external services like Active Directory or Keycloak.
Further aspects of this vulnerability are the way cookies and tokens are stored (Expert 1), rate limiting and automated attacks (Expert 2), and 'the human' as a very common point of failure through social engineering (Expert 4).

6.2.1.2 Frameworks

The following displays framework-native solutions for password policies, password hashing and authentication.

Password Policies

The frameworks React, Vue.js, Express, and Spring Boot do not offer any native password policies.

Angular offers the in subsubsection 6.1.6.1 described validators that can be used to also implement a simple password policy. The minimum and maximum of the password length can be validated. Furthermore to these native validators, a custom validator can be defined. However, no password-specific validators exist.

Password Hashing

In Express, the *crypto* module can be used to implement PBKDF2 or scrypt. The `pbkdf2` method does not offer any secure defaults for choosing a salt, the key length, or the iterations. However, it is not to neglect, that the documentation is referring to NIST SP 800-132 for guidance on how to configure the salt. The method uses SHA1 as default hashing function which is not favored [58]. Additionally, the example provided for the `pbkdf2` method in the documentation does not completely match with the recommendations provided by the OWASP Foundation [58], since only 100.000 instead of 120.000 iterations are used. On the other hand, the `scrypt` method is providing defaults for cost, block size, and parallelism. Even so, these values do also not align with the literature by having either a too low cost (16 MiB instead of 64 MiB) or a too low degree of parallelism (1 instead of 4).

Spring Boot offers the `PasswordEncoder` interface to hash passwords through Spring Security. This interface has concrete implementations for Argon2, bcrypt, PBKDF2, and scrypt. The implementation of Argon2 aligns with the literature. It uses 64 MiB of memory (which is more than recommended but does not have an impact on the security, only on the performance), two iterations, and parallelism of one. bcrypt uses a default salt length of 16 bytes, and ten salt rounds. PBKDF2's implementation does not align with the configurations found in the literature. Its default salt length is only eight bytes long, it uses SHA1 which requires 720.000 iterations [58] but is only doing 185.000 iterations. Finally, scrypt does use a CPU cost of only 16 MiB instead of 64 MiB, when considering the setting of the degree of freedom (one) that is set per default. The block size is configured correctly.

Authentication

Express does not offer a native functionality to implement authentication. In order to implement authentication, additional libraries are required.

Spring Security is offering native authentication for Spring Boot. In general, the authentication filter of Spring Security is intercepting the requests to URLs that are defined to be only accessible then authenticated. This filter then passes a `Authentication` object with the credentials to the `AuthenticationManager` that then accesses all available `AuthenticationProviders` and passes the `Authentication` object to them. After this, the providers use their `supports` method to determine whether they can authenticate this user. If they do, they call the `UserDetailsService` which is returning details of a user with a given name by calling the `loadUserByUserName` method. This method can for example use JPA to access any data stored in a database. The

`AuthenticationProvider` is then using its `authenticate` method to authorize the user based on the logic defined in the method. The result is again an `Authentication` object containing the `Principal` which represents the user and the authorities (in the `Authority` object), which define the user's rights.
If Spring Security is added to the dependencies, Spring Boot is automatically using basic authentication with a user whose password is set randomly and printed to the console when starting the application. Moreover, Spring Security is natively supporting LDAP authentication by adding a provider for that and also `UserDetailsServices` that receive data from within the memory or using JDBC. In addition, custom implementations of the `AuthenticationProvider` class can be used to shape the authentication process. A developer can make use of the `AuthenticationManagerBuilder` object that is passed to the `configure` method of the `WebSecurityConfigurerAdapter` class. Here, with the `authenticationProvider` method, a custom provider can be defined.

JWT:

No native JWT implementations were found for Express and Spring Boot.

Session Cookies:

Express offers one of its self-maintained middleware modules called *cookie-session*. First of all, the name of the cookie that is set through the library for the session identifier does not have a generic name. Its name defaults to `express:sess` and therefore gives away Express as used server framework. The name should be changed in the options. The hardening of the session cookie is considered. `Secure` defaults to true if HTTPS is used. `HttpOnly` is also true. No `domain` is passed and the default path is `/`. Session expiration can be achieved by setting `req.session` to null. The session IDs are signed by passing keys via the options to the library. These keys are passed further to the *cookies* library which cryptographically signs them based on SHA1 HMAC. To use more secure HMAC algorithms, a `Keygrip` object can be passed to *cookie-session* so that for example SHA256 is used for signing the session. However, the library does not contain any guidance regarding the length of the session identifier or its entropy. Therefore, using it without further knowledge does not result in a secure solution.

In Spring Boot, Spring Security enables session cookies by default. The name of the session cookie is `JSESSIONID` and therefore also not generic. The cookies are secured by setting `HttpOnly` to true. However, `Secure` defaults to false. The expiration of the session can be defined by passing a time to `server.serverlet.session.timeout`. The session can be stored through for example JDBC or in a redis database. The corresponding `SessionRepository` class will implement methods to handle a session.

The session ID is generated with the `randomUUID` method from *java.util.UUID*. This method generates a version four UUID (Universally Unique Identifier) based on the `SecureRandom` class which generates a "cryptographically strong random number" [139] as stated in the documentation. The general session management in Spring Boot is handled through the `SessionRepositoryFilter`. Finally, the session is migrated using the `migrateSession` method per default if a user authenticates successfully, protecting the server from session fixation attacks.

6.2.1.3 Libraries

Library-provided solutions for password policies, password hashing and authentication are discussed in the following paragraphs.

Password Policies

For Node.js-based frameworks different libraries exist that can be chosen to configure a password policy. The two libraries with the highest level of popularity are *password-sheriff* and *password-validator*. The former also functions through the definition of rules. These rules however are very minimal by only containing the length, the types of characters and how many of them that are contained in the password, and the number of identical characters that are allowed in a row. No default configuration is present. The library *password-validator* is working comparably by also defining rules for a schema. The amount of rules that are included in *password-validator* is higher, but they do not contain any of the recommendations that were defined in [57] except for the possibility to validate the length of the password. Further libraries like *common-password-rules*, *passwdqc* and *password-policy* offer a similar approach but are less popular and thus not further evaluated in detail.
These libraries do only offer the functionality to define a policy. Since this leaves room for mistakes and the literature also recommends the usage of strength meters, libraries that implement this functionality should be considered. In general, these libraries take the user-provided password and try to categorize it on a scale from bad to good. The most commonly used library is *zxcvbn* which is an open source project of Dropbox. As stated in its documentation it considers common passwords, common names, English words that are popular within Wikipedia and television, sequences of characters, and so-called 'l33t speak' (replacing characters with similar looking numbers). The library will return a score reaching from zero (too guessable) to 4 (very unguessable). The score is derived from the estimated amount of tries that are needed to crack the password. Also, suggestions to improve the password are returned that then can be displayed to the user. Further libraries exist such as *check-password-strength*, *hardpass*, *tai-password-strength* and many more. Since Carné de Carnavalet and Mannan [140]

and the OWASP Foundation [141] highlight *zxcvbn* as their preferred choice, further libraries are not evaluated.

The library *Passay* can be used to define a password policy in Spring Boot. A `PasswordValidator` object can be initialized with one or many `Rule` objects. These objects represent different constraints within the policy that have either positive or negative matching conditions. The positive rules define how the password should look like, by giving the possibility to pass a regular expression, define allowed or contained characters, and limit the length of the password. A set of rules, that help implementing in the literature recommended properties, is represented through the `DictionaryRule` and `DictionarySubstringRule` objects. These objects can be initialized with a list of common words (which has to be defined by the developer) and will lead to not accepting a password containing these words. The sequence rules can be used to disallow the usage of multiple identical characters sequentially. Furthermore, multiple constraints, that were not recommended in the literature but are often seen within web applications, can be implemented through this library. These constraints require a specific number of characters from a certain type of characters, disallow passwords that were used in the past, or do not allow the password to be related to the username.
As mentioned earlier, these constraints defining libraries do not ensure a secure password policy. Hence, zxcvbn, which also has a Java implementation through the library called *zxcvbn4j*, should be used instead.

Password Hashing

Express also offers the possibility to use libraries to hash passwords. First, the library *argon2* offers the functionality of the cryptographic hashing algorithm Argon2. Argon2i, Argon2d, and Argon2id, which are different types of Argon2 that have different resistances to side-channel or GPU-based attacks, are implemented in this library. It uses Argon2i by default, which is as stated in [142] also usable for password hashing. In general, its API is very easy to use and only requires to pass the to be hashed password to the `hash` method. A password can later be verified by passing the stored hash of it and the password provided by the user to the `verify` method. Its default values align mostly with the recommendations found in the literature. The salt length is 16 bytes, the amount of iterations is three, the default memory cost is 4 MiB ([58] recommends 15 MiB, but also with only two iterations) and the parallelism is set to one. Since the amount of required memory size decreases through increasing the number of iterations no clear statement about the security of these defaults can be given. Thus, the values proposed in the literature should be chosen. The salt is automatically generated using the `randomBytes` method of the *crypto* module and therefore cryptographically safe.

Library Name	Down.	First Rel.	Last Rel.	Rel. Freq.	Com. Freq.	Iss. Cov.	Avg Iss. c. T.	Contr.	Dep.	Qual.	Pop.
bcryptjs	942k	2013-05-01	2019-03-01	3.22 p.Y.	13.15 p.Y.	73%	106.06 d.p.Iss.	10	1356	95%	55%
bcrypt	685k	2011-02-21	2021-05-18	5.07 p.Y.	48.69 p.Y.	98%	34.79 d.p.Iss.	66	1737	97%	58%

Gathered: 2021-05-20

Table 6.4: bcrypt Libraries

Secondly, the password hashing algorithm bcrypt is considered. The libraries *bcryptjs* and *bcrypt* can be used to implement it. Both work interchangeably by offering a function to generate a salt with a given amount of salt rounds and a function that hashes the password with the given salt. Both use a default value of ten rounds. However, based on the metric as seen in Table 6.4, the library *bcrypt* seems to be better maintained and is also used more often as dependency making it the preferable choice.
The library *pbkdf2* is not considered, since this is the library used within the *crypto* module to provide the PBKDF2 functionality.

For Spring Boot the library *BouncyCastle* can be used. This library will be further evaluated in subsubsection 6.3.1.3.

Authentication

The following will display solutions from libraries to implement JWT or session cookies.

JWT:

Express can make use of the library *jsonwebtoken* to implement JWTs. This library was chosen because it is often used as dependency for other libraries such as *jwt*, *express-jwt* or *passport-jwt*. The library offers the methods `sign` and `verify`. These methods are used to sign and verify a JWT given a payload, a secret or private key, options, and a callback method. The options contain the algorithm which defaults to HS256, the period the token is valid, and more. In case the algorithms option is not passed, the library will search the `secretOrPublicKey` parameter which has to be passed to the `verify` method for the strings 'BEGIN PUBLIC KEY' or 'BEGIN RSA PUBLIC KEY'. When these are found, the library will verify the JWT with the corresponding public key algorithms (RS256, RS384, RS512, ES256, ES384, ES512) or RSA algorithms (RS256, RS384, RS512). If the parameter does not contain these strings, it defaults to HMAC algorithms (HS256, HS384, HS512). The library then searches the algorithm given through the `alg` key within the list. Hence, the attacks that function through forging a token with a manipulated algorithm as described in subsection 2.2.1, are not possible. None of the other concerns mentioned in the literature are solved through this library. However, again most of the concerns are related to the design of JWTs and therefore not solvable.

For Spring Boot, multiple libraries can be chosen. Firstly, the library *io.jsonwebtoken* implements JWTs for Java and Android. In order to deal with the `none` algorithm vulnerability, the library completely disallows the usage of unsigned JWTs by throwing an `UnsupportedJwtException` in that case. Furthermore, it is not possible to tamper with the algorithm stated in the `alg` key, since the library only verifies a JWT with a predefined key. The key defines the algorithm used when signing or verifying the token. In case the passed algorithm and the one defined within the code do not match, the library throws a `InvalidKeyException`. Furthermore, the library also helps with choosing a correct key length by throwing a `WeakKeyException` and referring to RFC 7518 within the error message. The other concerns that were mentioned in the literature are not covered.
Furthermore, a developer can choose to use auth0's implementation of JWTs with the *java-jwt* library. The library is also not vulnerable to attacks in which an attacker can manipulate the behavior by changing the `alg` key within the header of the JWT. When validating a JWT with this library, the developer has to define the algorithm used for the verification. Therefore, if a manipulated JWT with `none` as value for the `alg` key is tried to be verified using an HMAC algorithm, it will not succeed and result in a `AlgorithmMismatchException`. In contrast to the previous library, the length of an HMAC key does not need to comply with the standards defined in RFC7518. However, the documentation of the library emphasizes its importance. This library does also not cover the other concerns of the literature.

Session Cookies:

For Express, the library *express-session* offers an implementation of session cookies. The name that is used by default is `connect.sid` which is a better name than Express' native solution's default name, but not as generic as it should be. It is still possible for an attacker to identify the used library. Furthermore, the library sets `Path` to `/`, `HttpOnly` to true, `Secure` to false, and `maxAge` to null by default. In contrast to Express' native solution, the `Secure` attribute is not set dynamically whether HTTPS is used or not. However, the documentation emphasizes that setting `Secure` to true is recommended. The `maxAge` of the cookie makes the session to not be persisted throughout closing the browser, which is the most secure option. For the generation of the session identifier, the library *uid-safe* is used which creates secure UIDs. Its length is 24 bytes and therefore sufficient according to the literature. Additionally, the library also offers different methods for session management. A session can be regenerated (create new session id and session instance), destroyed, reloaded (reloaded from the session storage), saved (saves session into the storage), or touched (updates the `maxAge` attribute, which is also done automatically). Furthermore, this library offers the possibility to connect the session storage to a DBMS such as MySQL or MongoDB. Finally, no session fixation prevention is implemented per default. The `regenerate` method can be used for it, which however needs to be called once a user authenticates and therefore does not offer security by default.

No further libraries were found for Spring Boot.

6.2.2 Broken Access Control

Solutions for general authorization and CORS are described in the following.

6.2.2.1 Expert Interviews

In question one, Expert 1 to 3 all mentioned that authentication should be implemented through a library if one exists. Expert 4 pointed out that it is depending on the context whether he chooses a library or not. Expert 2 recommends using annotations when implementing authorization in Java applications.
Further aspects that Expert 1 mentioned were that manual testing is very important. In addition, Expert 4 referred again to 'the human' as a common point of failure.

6.2.2.2 Frameworks

In the following paragraphs, framework-based solutions for authorization and CORS are evaluated.

Authorization

In Express no native solution for authorization is implemented. The implementation of such a mechanism is therefore highly dependent on the engineering capabilities of the software developers. However, a possible implementation of authentication could make use of Express' middleware functionality. Using the middleware, the authorization can take place in one centralized location within the application and then be used at every endpoint. This could directly restrict access to an endpoint, in case the request could not be authorized. In conclusion, further libraries or in-depth testing of the authorization mechanisms should be considered.

Spring Boot offers an authorization mechanism natively with Spring Security. Through the usage of different built-in expressions, access to certain resources can be restricted.

These restrictions include the role or authority a user has. The roles can be self-defined and offer the possibility for a hierarchy of roles and the authorities define operations such as 'read'. These expressions can be used in different ways within Spring Security. One way is to make use of the annotations `@PreAuthorize` and `@PostAuthorize`, which were highlighted by Expert 2, that are placed before the to be restricted methods. The expressions mentioned above can then be passed to the annotations to implement the level of restriction that is needed. The annotation `@PreAuthorize("hasRole('USER')")` would restrict the underlying resource to everyone who is within the group `USER`. Furthermore, this functionality can also be implemented through the `configure` method within the `WebSecurityConfigurerAdapter` class. In contrast to the `configure` method seen in subsubsection 6.2.1.2 that is used to implement authentication, this one is passed a `HttpSecurity` object. For this object different `antMatchers` can be defined that corresponds to a certain URL. This URL definition can then be restricted through expressions. In order to restrict the access of a certain resource (URL) to clients with the USER role the following code can be used: `antMatchers("/books/**").hasRole("USER")`. Therefore, the resource books with all underlying URLs is restricted. [143]

Cross-Origin Resource Sharing (CORS)

Express offers the *cors* middleware library that is maintained by Express. It can be used by either adding it to the middleware, which makes the whole server accessible through the defined origins or only within a specific route.
As already described in the literature, the developers are responsible for securely defining CORS. Thus, this library is not adding any security but only giving the possibility to setting CORS conveniently.

Similar to Express, Spring Boot is natively providing a convenient implementation of CORS. The annotation `@CrossOrigin(origins = "http://myApp:8080")` makes cross-origin request from the defined origins possible. Furthermore, if Spring Security is used, the global usage of CORS can be defined within the `configure` method of the `WebSecurityConfigurerAdapter` class.
As written above, no security is provided through the usage of this functionality.

6.2.2.3 Libraries

Library-based solutions for authorization and CORS are described in the following discussion.

Authorization

In Express a developer can choose multiple libraries to handle authorization. However, if the popularity with these libraries is compared to other libraries, it seems like this kind of functionality is often implemented without usage of an additional library, in contrast to what was stated by the experts. The most popular authorization library is the library *accesscontrol*. It implements role-based access control in combination with attribute-based access control. This means a user can define a role that can create, read, update or delete a certain resource. Similarly, the library offers a method to check whether a role can access the resource in the way it wants to (for example read a resource). The library *rbac* works in the same manner as *accesscontrol*. Furthermore, *casbin* can be used to implement authentication. It functions by reading a model and a policy configuration file. Depending on how these are configured different authorization models can be defined such as role-based or attribute-based access control. Finally, the library *abac* exists that seems to be deprecated and not used, therefore it is not considered.
All of these libraries have in common to only support the developer in defining roles and permissions for users and resources. This can of course be helpful, however, it does not protect developers in any way from allowing unauthorized access. The defined roles can be imprecise and allow room for attackers to exploit the application. Thus, the web application should be evaluated for errors within the implementation of authorization.

Libraries that do authorization for Spring Boot could not be found while researching.

Cross-Origin Resource Sharing (CORS)

Further libraries exist for Express that help to deal with CORS. These libraries are *simple-cors*, *yeps-cors*, *cross-origin* and more. However, due to their very low popularity, they are not considered.

Spring Boot's implementation of CORS seems to be the standard when implementing the CORS functionality and therefore no further libraries could be found.

6.2.3 Cross-Site Request Forgery

Frameworks and libraries are evaluated in the following by reviewing the corresponding CSRF implementation.

6.2.3.1 Frameworks

Angular is the only client-side web framework that offers support for handling CSRF tokens. Its `HTTPClient` has a built-in CSRF protection by supporting the token patterns. The interceptor of Angular searches the cookies for an `XSRF-TOKEN` and automatically sets it to the `X-XSRF-TOKEN` header. In order to change these default names, one can pass options to the `HTTPClientXsrfModule.withOptions` method. [130]

In Express the official Node.js CSRF library *csurf* can be used which uses the library *csrf* as dependency. Since it is maintained by the Expressjs team, it is considered framework-native. *csurf* is using the secret and token generation functionality of *csrf*. The offered approach is using HMAC functions but does not fully implement the HMAC Based Token Pattern. Instead of hashing the session ID and a timestamp, the token consists of the following string: `salt + ’-’ + hash(salt + ’-’ + secret)`. The salt is created beforehand by the *rndm* library. As stated in its documentation, these salts are not created cryptographically safe. The hashing is done with the SHA1 function in combination with further configurations. When verifying the token, the salt is removed from the token and used to re-generate the token. Both tokens are then compared using the *tsscmp* library to avoid timing attacks. This implementation can either be used with the *express-session* library, if storing these tokens with the session is chosen, or with the Double Submit Cookie method resulting in a separate cookie for the CSRF token. This choice is made by passing the `cookie` attribute as options to the *csurf* library. Also, options to enhance the security of the cookies as mentioned in the literature are available, but not set to secure values by default. Both attributes `Secure` and `SameSite` default to false. In addition, to find the token, the library will search different locations such as the request body and query as well as under certain header names.
In conclusion, Express’ CSRF token creation and verification approach does not match completely with what is described in the literature since, instead of session information and a timestamp, only a salt and the secret are required to verify a token. Furthermore, none of the above-mentioned security enhancements are enabled by default such as `SameSite` or `Secure`. Therefore, a correct configuration of the cookies and a further evaluation of the security of the verification process is required.

Spring Boot has built-in support for CSRF tokens with Spring Security, by enabling the usage of CSRF tokens by default. This makes it very easy to use and adds security by default. The used pattern for token handling is the Synchronizer Token Pattern, which is implemented as described in the literature. The token is generated similar to the session ID using the `randomUUID` method. In addition, the cookie is enhanced with the `Secure` and the `Max-Age` attributes. `SameSite` is not set. The tokens are then compared using the `equalsConstantTime` method from the `CsrfFilter` class which provides a constant-time comparison to avoid timing attacks. If these two tokens match, the `FilterChain` is passed on and if not, the request stops. [144]

6.2.3.2 Libraries

For Express, there are many non official alternatives to *csurf*. The following libraries can also be used to implement CRSF tokens: *csurfer*, *hmac-csrf*, *csrf-simple-origin*, *express-csrf-double-submit-cookie* and *csrf-guard*. However, because of their generally low popularity, these libraries are not considered.

Spring Boot can make use of OWASP's *CSRFGuard* which is another implementation of CSRF tokens. However, since it seems outdated with its latest version 3.1.0 released in 2015, it was not considered in this thesis. Once version 4.0 releases, its release date is planned to be in 2021, it could be considered as a possible alternative.

6.3 Configuration-based Vulnerabilities

Solutions for the third group of vulnerabilities can be found in the following sections.

6.3.1 Sensitive Data Exposure

The following deals with cryptography and key management. No solution for the general management of data was found within frameworks or libraries. Code examples for the solutions regarding cryptography were created for general JavaScript[12] and Spring Boot[13].

6.3.1.1 Expert Interviews

In order to make data in transmit confidential (question one), the experts agree on using encryption through HTTPS. Furthermore, as mentioned by Expert 2, no sensitive information should be written into the URL. Expert 3 emphasizes that also the use of the correct architecture is important. However, since this is more fitting for the second question, this answer will be covered there. Expert 4 says to enhance the security even more a VPN can be used.

[12] `https://github.com/moritzhuether/mastersthesis/blob/main/GeneralJS/cryptography.js`

[13] `https://github.com/moritzhuether/mastersthesis/blob/main/SpringBoot/demo/src/main/java/main/demo/WebSecurityConfig.java`

In terms of where to encrypt data, Expert 2 and 3 both replied by emphasizing that the architecture is important. Expert 3 says that, starting at the design of the data model and also applying it to the rest of the application, a developer has to consider what data is needed, will be sent, and who can access it. Therefore, as also stated by Expert 2, it is important to minimize the required data in the design phase. Expert 2 also answered that passwords should be hashed, personal data should be encrypted and a possible library for Java is *BouncyCastle*. On the other hand, Expert 4 relies fully on the encryption done when transmitting data.
The chosen algorithm (question three) should be within a common library (Expert 2) and should also be secure (Expert 3). Expert 3 discouraged the usage of MD5 or Base64. In the last question regarding further aspects of this vulnerability, Expert 2 and 4 both mentioned again the minimization of data that is used within the system.

6.3.1.2 Frameworks

The following paragraphs highlight different solutions existing for cryptography and key management which are implemented through framework-native functionalities.

Cryptography

The Node.js *crypto* module, which makes use of OpenSSL, can be used to encrypt data. By calling the method `createCipheriv` of the module, a new cipher can be created. The cipher takes an algorithm that is used for the encryption, a key that is used by the algorithm, and the initialization vector. To decrypt the data again, a `Decipher` object has to be created with the same parameters for algorithm, key, and initialization vector using the `createDecyipheriv` method. Both objects provide an `update` and a `final` method that, if used in combination, will produce the en- or decrypted data. For the creation of `Cipher` or `Decipher` objects no defaults are set. Not providing the above described parameters will result in an error. Thus, the security of cryptography is highly depending on the choice a developer makes. The example that is shown in the documentation is however aligning partially with the literature. It uses AES-192 with the CBC mode (which should be changed to GCM or CCM) and generates both the key and the IV randomly with cryptographically secure functions.

In Spring Boot a developer can use classes provided by Spring Security to already receive a default configuration. These classes are `Encryptors` and `Decryptors` and they can be used for symmetric encryption. These classes offer a rather simple API for the initiation of either byte or text en- or decryptors. The `ByteEncryptor` class, which returns only bytes, can be created by calling the methods `standard` or `stronger`. Both methods require a password and a salt to be passed to them. If the `standard`

method is chosen, the encryption is done using 256 bit AES encryption. The secret key is automatically created through PBKDF2 using the given password and the salt. Furthermore, it uses a 16 bytes long initialization vector that is created with Java's `secureRandom` method. By default, it uses the CBC mode, which is only recommended with additional authentication [72]. Thus, the `stronger` method should be chosen. It also encrypts with 256 bit AES encryption but using the GCM mode. This is the only distinction from the `standard` method. Furthermore, in order to create a `TextEncryptor` object, that returns strings when en- or decrypting, the methods `text`, which is the equivalent of `standard`, and `deluxe` with is the equivalent of `stronger`, can be used. Once the objects are defined, their `encrypt` and `decrypt` methods can be called. Within these methods, a `Cipher` object is created with the AES instance, the operation mode and the data is en- or decrypted with the `doFinal` method.
The `Cipher` class can also be used directly at a lower level. This is less foolproof since no defaults are used in that case.

Key Management

Express has no native functionality to manage keys.

Spring Boot on the other hand offers Spring Vault which provides an interface to HashiCorp's Vault. This is a secret management tool that helps to store tokens, secrets, or cryptographic keys. In addition, it implements the key management lifecycle with making it possible to create, rotate and revoke keys. Spring Vault gives the possibility to easily access Vault by defining a `VaultTemplate` that connects to the URL Vault is running at. The template can then be used to revive secrets stored in the Vault with the `read` method.

6.3.1.3 Libraries

Solutions for cryptography and key management, which can be found in libraries, are described in the following.

Cryptography

In the Node.js environment, the library *crypto-js* can be used. In contrast to the *crypto* module, it implements default configurations that are easy to use. For example the method call `CryptoJS.AES.encrypt("Message", key)` will encrypt the message with the AES algorithm. The concrete type of the AES algorithm is chosen depending on the key length that is passed (a 256 bits long key will result in AES-256). In case a passphrase is used, which is also supported, a 256 bits key will be generated. A 16 bytes long initialization vector is created. The default mode is CBC, which requires further authentication, and the modes recommended by [72], GCM, and CCM, are not implemented.

The Java Cryptography Extension (JCE) makes it possible to use further providers for cryptographic ciphers. The most common example for such a provider is *Bouncy-Castle*. By adding this library to the providers, its ciphers, that exceed the ones native to the JCE, can be chosen. However, it can only be used to increase the customizability of Spring Security's solution and does not offer a default configuration for the encryption.

Key Management

In Express also an interface for HashiCorp's Vault is existing. The library called *node-vault* does, just as Spring Vault, offer an interface to write, read and delete secrets in the vault.

No further libraries were found for Spring Boot.

6.3.2 Security Misconfiguration

As the literature already described, the process of security hardening is often combined with the build process of the web application. Therefore, only solutions for HTTP headers and error handling are analyzed in the following.

6.3.2.1 Expert Interviews

In question one, the experts all said that no library exists to do configuration. However, Expert 3 referred to linters and other tools to check configuration files, which could not be found while researching. Expert 4 puts his trust into configuration recommendations that are found on the internet.
For error handling, question two, the experts also agree that no library exists that implements the functionality of error handling holistically. As stated by Expert 3, engineering is needed to do error handling. Experts 1, 2, and 4 pointed out that setting the environment from 'Debugging' to 'Production' is important. In terms of implementation suggestions, Expert 2 suggested implementing one error handler that catches the whole application, as also found in the literature.
The HTTP headers, that were covered in question three, were generally important to the experts. As Expert 4 stated: "all of them are important". Concrete examples were mentioned by Expert 2: CSP, `cache-control`, `Strict-Transport-Security`, `X-Frame-Options` and the 'nosniffing' header called `X-Content-Type-Options`. Expert

3 adds the usage of CSRF tokens to the list. The headers mentioned by Expert 1 are already covered by Expert 2.
Expert 2 was able to answer further questions regarding the CSP (question four). He recommended a whitelist approach that only considers what is actually used and therefore is as restricting as possible. Furthermore, the usage of Google's CSP evaluator was recommended. Expert 1 mentions the `frame-ancestors` directive to be important. A further aspect that was mentioned by Expert 4 was that a whitelisting or 'Zero Trust' approach should be used when configuring a web application.

6.3.2.2 Frameworks

The following will highlight different solutions of frameworks for HTTP headers and error handling.

HTTP Header

Express does not have a native configuration of the necessary headers that were discussed in the literature. The usage of libraries should be considered.

Spring Boot, on the other hand, provides a default configuration of HTTP headers with Spring Security. An overview of the headers it sets and how these are configured can be seen in Table 6.5. The default configuration already offers a security baseline. However, certain configurations should be reconsidered or are missing. First of all, the `Strict-Transport-Security` does not set the `preload` directive. Since this directive is mainly important for high-risk web applications it does not need to set for every configuration [68]. Setting the `X-Frame-Options` to `deny` is aligning with the literature and the most secure way to configure this header. Furthermore, as described in the literature, the usage of a CSP can add a high level of security, which is completely missing from Spring Security's configuration. Not setting the `Referrer-Policy` header means its default is used, which is according to the literature also usable. The `X-XSS-Protection` on the other hand, is not set correctly by configuring it to `1; mode=block`. As described in the literature, this value should be set to `0`. Furthermore, different headers that are concerned with caching are set natively by Spring Boot. These headers are configured as described in the literature. Finally, missing headers can be added through the `addHeader` method.
To further enhance the security of HTTP headers, the CSP should be considered since it offers defenses against XSS and clickjacking. In addition, the incorrect configuration of the `X-XSS-Protection` needs to be overwritten.

Error Handling

This subsection deals with the handling of errors. Error messages, which either show detailed information about why the error happened or even stack traces, can lead to the leakage of information. Therefore, the following sections will cover the approaches of error handling for the different web frameworks.
The frameworks, in general, do not offer any solution to disallow error messages with too much information. As mentioned by Expert 3 it requires engineering to deal with this issue. Another general approach that was emphasized by Experts 1, 2, and 4 is that the developers should differentiate between development and production mode. The production mode often affects the way the frameworks handle errors, which will be described in the following.

Starting with React, a developer can use an `ErrorBoundary` component. This feature was introduced in React 16 and these components make use of the lifecycle method `componentDidCatch`. If this method is implemented, it will catch JavaScript errors that occur inside of child components. Thus, the error boundary needs to wrap all components for which error handling is intended to be implemented. In case an error is thrown, the error boundary component will catch it and offer possibilities to apply certain logic like displaying corresponding error messages. These error messages should not contain too much information and also a default message should be implemented if an unknown error happens. However, then a React application is created through the `create-react-app` command, error messages containing stack traces will be shown even in production. As stated in [145], disabling this is mandatory.

Vue.js is offering the possibility to define a global error handler in its configuration. The error handling logic needs to be passed to `Vue.config.errorHandler` in order to trigger it when uncaught errors appear. This configuration is also addressed if Vue.js is run in production mode and an error occurs. Another option is to use the lifecycle hook `errorCaptured`, which is called in the case of a captured error within child components.

Angular provides the `ErrorHandler` class that offers the possibility to implement a custom error handler. This handler can be used to centralize the exception handling of the web application. The necessary logic can be put inside of the `handleError` method. In Angular, no information regarding the influence the production mode has on the displaying of errors could be found. However, as stated in Angular's documentation, the code is uglified which reduces the value stack traces have for attackers by giving functions and variables unreadable names. [146]

In Express a developer can make use of the middleware. In case a function with four parameters is passed to `app.use` method, Express will treat it as an error handler since other functions have only three parameters. The fourth parameter, which is additional to the typical `res`, `req` and `next`, parameters is `err`. It is used to pass errors to the function. Inside of this function, the logic for error handling can be defined. Therefore, a developer can check the type of error, its message and handle it accordingly. To tell Express to go to the defined error handler, it is important to put the error handler middleware lastly in the program. The method `next` can then be used to tell Express that the current middleware should proceed to the next. In the case of a standard Express application, Express will then go from the called endpoint (`app.get(...)`) to the error handler. Thereby, default error messages can be defined and no stack traces will be printed to the user. In addition, if used in production, Express' default error will trigger when `next` is used and no custom error handler is present. [147]

Spring Boot offers multiple solutions for error handling. The first approach is an error handler at the controller level used via the `@ExceptionHandler` annotation. If this annotation is placed before a method, this method will be called in case an error is thrown within the controller. In addition, Spring Boot also offers the `@ControllerAdvice` annotation for global error handlers. By adding this annotation to a class, it will become a global event handler and a developer can implement methods with `@ExceptionHandler` in order to catch certain errors. This means a developer can throw an error without handling it and it will be caught by the class with the `@ControllerAdvice` annotation. Spring Boot offers the Whitelabel Error Page which acts as a default error page in case the error is not caught. With the configuration of `server.error.include-stacktrace` and `server.error.include-message` a developer can configure if the stack trace or the error message should be included. If both are disabled, the information passed to the client is minimized.

6.3.2.3 Libraries

Library-based solutions for HTTP headers and error handling can be found in the following paragraphs.

HTTP Header

Express recommends the usage of the library *helmet* as a standard configuration for the HTTP headers. The configurations for the headers reviewed within this thesis can be seen in Table 6.5. First of all, *helmet* does set the `max-age` directive of the `Strict-Transport-Security` header to a rather low value. The browser will remember to use HTTPS for only 180 days instead of the recommended two years. However, the

set value of 15552000 seconds is close to the minimum value of 15768000 defined by [68]. Increasing this value is recommended and adding the `preload` directive should be considered. Following, the `X-Frame-Options` header is set to `sameorigin` in contrast to the `deny` setting of Spring Security. As described above, if this feature is not used setting it to `deny` should be considered. However, setting it to `sameorigin` does not offer any security threat [148]. The CSP that is provided through *helmet* does not result in any errors within the CSP evaluator proposed in [89]. The `Referrer-Policy` is set to `no-referrer` which is the most secure way of handling this header. Moreover, disallowing the `X-XSS-Protection` and setting `X-Content-Type-Options` to `no-sniff` does align with the literature.
As seen in the table, further libraries exist, that also have the same maintainer as *helmet*, that add further configurations. For dealing with caching, the `nocache` library can be used that defines configurations that are aligning with the literature. Finally, the library *clearsitedata* can be used to set the corresponding header to `*` which can be used when a user logs out. This will delete all data stored inside of the cache, cookies, and storage. The configurations that these libraries undertake by default are mostly covering what is proposed in the literature and also configure them according to the recommendations found in the literature. Therefore, a high level of security can be achieved.
In addition, the library *node-guard* also proposes a solution for HTTP header. Even so, it is not defining defaults and therefore only adding a convenient way of defining headers. It is not further evaluated.

Error Handling

In general, all Node.js-based frameworks can make use of the *errors* library. This library does not offer security features but provides a convenient way to create custom errors.

For React the library *react-error-boundary* can be chosen. This library provides an `ErrorBoundary` component which has the possibility to pass a `FallbackComponent` and an `onError` method. Nevertheless, it makes also use of the `componentDidCatch` method and therefore only offers convenience. Further libraries such as *react-exception-handling*, *@bentley/itwin-error-handling-react* and *react-handling* were not considered due to their low popularity.

In Vue.js a library called *vue-error-boundary* exists which implements a functionality similar to the `ErrorBoundary` described in React. It makes use of Vue.js' `errorCaptured` method. Therefore, it does only implement a convenient way that uses Vue.js' native functions. However, since it is not maintained well with not receiving updates in the last two years, it is recommended to use Vue.js' implementation. The libraries *vue-*

Header / Configuration	**Express** *helmet*	**Express** *nocache*	**Express** *clearsitedata*	**Spring Boot** Spring Security
HSTS	max-age=15552000; includeSubDomains			max-age=31536000; includeSubDomains
X-Frame-Options	SAMEORIGIN			deny
Content Security Policy	default-src 'self'; base-uri 'self'; block-all-mixed-content; font-src 'self' https: data:; frame-ancestors 'self'; img-src 'self' data:; object-src 'none'; script-src 'self'; script-src-attr 'none'; style-src 'self' https: 'unsafe-inline'; upgrade-insecure-requests			
Referrer-Policy	no-referrer			
X-XSS-Protection	0			1; mode=block
Cache-Control		no-store, no-cache, must-revalidate, proxy-revalidate		no-cache, no-store, max-age=0, must-revalidate
Expires		0		0
Pragma		no-cache		no-cache
X-Content-Type-Options	no-sniff			no-sniff
Clear-Site-Data			*	

Table 6.5: Overview of HTTP Header Configurations of Frameworks and Libraries

reactive-error-handler and *vue-error-controller* were not considered due to low popularity.

Angular does not offer any popular libraries that help to deal with errors. The libraries that were found but not considered are: *angular-invocation-handler*, *@btapai/ng-error-handler* and *ehandler*.

Libraries that could be found for Express mainly offer no additional security. The libraries *express-async-errors*, *express-rescue* and *express-safe-async* offer the possibility to asynchronously handle Express errors. The handling of the errors themselves has still to be implemented similarly to Express' solution. Therefore, no change in security can be achieved.

Different libraries can be used to implement error handling in Spring Boot. The first library is *Problem* developed by Zalando. It replaces errors through so-called `Problems` that can be displayed to the user. A `Problem` is a JSON that contains data such as title, status, details, message, and more. Different default 'advice traits', which are methods that have the `@ExceptionHandler` annotation, are built-in that can be used to display common error messages with minimal information to the user. Through this, exceptions such as `UnsuportetMediaTypeException` or `AccessDeniedException` can be thrown without needing any input by the developer. Thus, simplifying Spring Boot's native approach. Furthermore, the library *errors-spring-boot-starter* can be used to enhance Spring Boot's native error handling with different error messages and codes by adding the `@ExceptionMapping` annotation. No further functionality than adding more information to the errors is implemented through this library. Thus, not adding further security.

6.3.3 Using Components With Known Vulnerabilities

This subsection highlights the differences of available dependency checkers.

6.3.3.1 Expert Interviews

The experts stated in question one that they mainly use npm's built-in vulnerability checker. Expert 2 also mentioned the usage of the OWASP Dependency-Check. Expert 4 emphasized the importance of own research and staying up to date regarding new vulnerabilities.
Expert 2 stated to not only trust these vulnerability scanners but to also do research in question two. Expert 3 mentioned patching the vulnerable libraries and to inform customers about possible vulnerabilities.

6.3.3.2 Frameworks

First, there is the automatic vulnerability checker of npm the package manager of Node.js. Therefore, these results are applicable for all frameworks that are within the Node.js environment. This checker is either called if a new library is installed or by writing the command `npm audit`. Thereby, the dependencies of the project are sent to the selected registry and scanned for known vulnerabilities. The dependencies that are checked are the following: `direct dependencies`, `devDependencies`, `bundledDependencies` and `optionalDependencies` [149]. These dependencies are put into a JSON object and then send to either the Bulk Advisory endpoint (primary source) or the Quick Audit endpoint. The vulnerability database can be seen here [150]. As stated in [151], this database is maintained mostly by volunteers. In addition, `npm audit fix` offers the possibility to directly update any vulnerable dependencies if a fix exists.

Spring Boot does not provide native functionalities to search for vulnerabilities within the used components.

6.3.3.3 Libraries

Since some scanners can find vulnerable components in both Spring Boot and Node.js, the following paragraph will be covering both of them concurrently.
An available solution is the npm library called *Snyk* which works for both Node.js and Java. In contrast to npm audit, this vulnerability database of Snyk is maintained by a dedicated team as described in [152]. The command `snyk test` can be used to find vulnerabilities and either `snyk wizard` or `snyk protect` can be used to fix vulnerabilities. While the first one goes through every vulnerability and offers different possibilities, the second one automatically applies patches. For all functionalities please refer to [153]. Snyk also has an integration for Java via Gradle or Maven.
OWASP is recommending the usage of *Retire.js* which can for example be used in the command line or as a browser plugin. If used via the command line, the command `retire` paired with different options will check for vulnerabilities, and the browser plugin prints out warnings to the developer console if a reference to an insecure library is found. This scanner does not mention the sources of its vulnerabilities and is only applicable for JavaScript. [154]
A Java-only solution is the OWASP Dependency-Check project which, as its name already tells, is developed by the OWASP Foundation. It can be used through a command-line interface or as a plugin in Maven. As a data source, it uses the NVD. Further scanners exist, such as the Nexus Vulnerability Scanner of Sonatype or Gitlab's dependency scanner.

6.3.4 Insufficient Logging And Monitoring

Finally, solutions to log data are considered. As defined in chapter 2, monitoring is a task that should be done through individuals, and also because only external tools could be found to automatically monitor logs, it is not further considered. In order to demonstrate the loggers, code examples were created for both Express[14] and Spring Boot[15].

6.3.4.1 Expert Interviews

The experts answered the first question with the libraries *winston*, *morgan* and *log4js*. In the second question, Expert 2 suggested doing structured logging instead of text-based logging. Furthermore, the experts all agree on not logging any sensitive data

[14] `https://github.com/moritzhuether/mastersthesis/tree/main/Express/10-InsufficientLoggingAndMonitoring`
[15] `https://github.com/moritzhuether/mastersthesis/blob/main/SpringBoot/demo/src/main/java/main/demo/Injection.java`

such as passwords or sessions.
For analysis, in question three, the experts suggested either manual analysis of the logs or the usage of third-party software to monitor logs.
Further aspects were that these logs should be analyzed regularly (Expert 2) and that this is especially important since systems become more and more distributed (Expert 4).

6.3.4.2 Frameworks

The following solutions for logging are divided into language- and framework-specific solutions.

Language-Specific Solutions

JavaScript offers different functions that help to log to the console with the `Console` object. It offers methods such as `log`, `warn`, `error`, or `time` that can be used to log strings. Therefore, some levels of severity can be achieved. However, the log can only be written in the console and not to any persistent storage. Furthermore, no formats exist, therefore the logs have to be written in a specific syntax that has to be defined and used throughout the whole application. This feature is rather focused on helping developers debugging the web application than offering support for logging.

Similar to JavaScript, Java is offering the possibility to log to the console. This can be done through the method call `System.out.print()` which prints the passed data to the console. It has the same restrictions as described above and therefore should only be used to debug the web application.
In addition, the *java.util.logging.Logger* class can be used to log data. It offers different levels such as `config`, `info`, `sever` and more. These logs already contain the current time and the class the log was issued from, answering the when and where question. In addition, if for example an exception is passed to the standard `log` method, the exception is logged as well, which is answering the why question. The other questions are not answered. Furthermore, a handler can be added through the `addHandler` method that enables logging to files. This handler does also accept an `Formatter` object, that defines how the logs are formatted within the files. Java provides a formatter for text, which writes the console output to a text file, or a formatter for XML content that writes the output within an XML file.

Framework-Specific Solutions

For logging, Express offers its supported library called *morgan*. This library helps with logging HTTP requests by adding it to Express' middleware. Five different types of formats can be chosen to shape the log. The most comprehensive ones are the `common` and the `combined` formats that are based on the Apache logging standard [155]. The `combined` type extends the `common` type with information about the used agent (for example the browser). The current date and time are logged in UTC (RFC 1123 format) to cover the when question. In order to cover the where question, the URL that was accessed is logged. The library covers the who question partially by logging the IP address of the user doing the request. However, no user identifier is added, if no basic authentication is used. The what question is not answered but defining the severity of a request is hardly feasible for a library since it is highly dependent on the web application. Furthermore, the how and why questions are also not answered. More aspects that are logged are the response time, the bodies of the request and response, the HTTP version and method, and the referrer information. To fill the gaps defined above, a developer can configure the middleware as desired. Therefore, custom logic is needed that is adapted to the web application defining data such as the severity of the action that was carried out and the reason why this log was created. Finally, the library offers also the possibility to log to a certain file or to use other libraries to log to for example MongoDB with *mongoose-morgan*.
Express' native logging offers a default, that is already covering most of the properties defined in the literature. However, the data, that is highly dependent on the web application, needs to be added through a developer. Therefore, the logging process should be evaluated further. Another disadvantage of this native functionality is that it has no general implementation for Node.js. This means that a logging syntax has to be defined so that both client and server produce logs in the same format.

Spring Boot uses the *spring-boot-starter-logging* dependency to implement logging. This functionality is making use of *Apache Common Logging* within the Simple Logging Facade for Java (SLF4J) that ist used to access the logger. This means that the code of a developer will access the SLF4J API that then accesses the reference implementation that logs data based on the API calls of SLF4J. Similar to the Java-specific solution, a `Logger` object is defined that is now implemented by SLF4J and can be used to trigger the logging in the reference implementation. The default reference implementation that is used by Spring Boot is *Logback*. *Log4j2* is also natively supported as a reference implementation. Since *SLF4J* defines what kind of data is logged both reference implementations log the following information by default: Date and time, a level of the log, a process id, the thread name, the name of the logger to make clear in which class the log was created, and a log message. Therefore, no information is given regarding the user causing the log. The description needs to cover information such as the action that was done or what URL was accessed to answer more than only the when and where questions. Furthermore, an XML file, in the case of *Log4j2* a JSON or a YAML file is also possible, can be created, which defines the pattern a log should have, where it should be stored, and which levels should be logged. *Log4j2* does also offer further implementations such as the support for lambda expressions, asynchronous logging, and storing logs in daily changing files.

6.3.4.3 Libraries

Node.js offers a vast assortment of different logging libraries. The most of these libraries offer the functionality to add levels to the logs: *loglevel*, *logging*, *signale* and *consola*. The libraries *log4js*, *log-driver*, *roarr*, *fancy-log* and *pino* do also log the current time, for certain libraries also more information, per default. The libraries *winston*, *tracer* and *bunyan* give the possibility to define a logging format and are therefore further evaluated in the following. An overview about them and also the library *morgan* can be seen in Table 6.6.
The first logger that is covered is a library called *winston*. The library defines seven different levels to log data. Furthermore, it has built-in transports which describe the way a log is displayed or stored. It can log data to the console, a file, send it via HTTP (can be useful when logging on the client-side) or in a stream. Further external transports enable the connectivity to a DBMS or the daily rotation of files. Unfortunately, the logger does not define a standard logging syntax. This means that the syntax needs to be defined by the developer using the provided `formats`. This offers extremely high customization by giving methods to define formats, combine them, display JSON or errors, or colorize and align them.
Secondly, the library *tracer* will be evaluated. It provides six log levels and also the possibility to daily log to a different file. Per default, the log includes the current time, the level of the log, the file and location in the file, the method that caused the log, and its message. Furthermore, a developer can customize these logs using a set of predefined tags. Nevertheless, even if these predefined tags are used, data regarding the user who caused the log is missing.
Finally, the library called *bunyan* is reviewed. It also implements six levels of logs, and the possibility to stream certain logs to files. The logs contain the name that was passed when creating the logger (the used class can be defined through this), the hostname, a process ID, the level, the message, the time on which the log was issued, and the version of the current format. The library provides the option to pass a serializer to the logger. The serializers take a `req` attribute and return a log that was created out of the

Library Name	Down.	First Rel.	Last Rel.	Rel. Freq.	Com. Freq.	Iss. Cov.	Avg Iss. c. T.	Contr.	Dep.	Qual.	Pop.
winston	5,8M	2011-01-18	2020-06-23	8.02 p.Y.	130.30 p.Y.	72%	214 d.p.Iss.	310	9072	97%	81%
morgan	2,7M	2014-02-08	2020-07-12	3.56 p.Y.	51.05 p.Y.	94%	19 d.p.Iss.	20	5332	88%	69%
bunyan	1,2M	2012-02-02	2021-01-08	12.25 p.Y.	76.32 p.Y.	55%	140 d.p.Iss.	62	2197	75%	61%
tracer	58k	2012-03-02	2020-10-31	5.52 p.Y.	26.00 p.Y.	92%	127 d.p.Iss.	32	406	96%	36%

Gathered: 2021-05-22

Table 6.6: Logging Libraries

`req` attribute through self-implemented code. Doing that, a serializer can be defined that adds data about the HTTP request or similar to the log. This is necessary to be holistic in regards to the literature.
The results of the library metric (see Table 6.6) suggest to either use *winston* or *morgan*. In case Express is used, *morgan* is offering a good baseline of relevant data by default. However, *winston* is offering more customizability and also the possibility to use it not only on the server-side. In general, all mentioned libraries can produce secure logs, but depend highly on the configurations of the developers.

In Spring Boot, the library *tinylog* can be used instead of the native logging mechanisms. After adding it to the dependencies, its *Logger* object can be used to log data. The library supports five different levels of severity and has different 'writers' to write the logs to the console, files, or database. The default format pattern that is used by *tinylog* logs the following information: the current date, the thread that created the log, the class, and method, the level of severity, and its message. Similar to *tracer*, a list of predefined tags are available to define a custom format pattern but they also do not include information about the user causing the log, or any other HTTP-related information.

7 Evaluation of Automated Tools

Finally, the results of the expert interviews regarding tools and processes are presented and automated tools, that were disclosed within the literature and the interviews, are analyzed in the following sections.

7.1 Expert Interviews

The second group of interviews was concerned with tools and processes that are included in the software development process.
The first block of the questionnaire was asking questions targeting the usage of tools. All of the experts named the SAST tool Sonarqube that is used in almost every project they encounter. Experts 7 and 8 both emphasized that it is "state of the art" to include the tool in projects. Further tools that could be gathered are PDM, which is a static code analyzer, Firebuck, which is a commercial quality assurance testing tool, and tools concerned with licensing. They are called Fossa and Whitesource and are only commercially available. Moreover, experts 2 and 4 also mentioned CodeQL in Github and therefore Looks Good To Me (LGTM), which is used by Github. Furthermore, the OWASP Dependency-Check was mentioned multiple times and Expert 7 mentioned Defect Dojo which is no scanning but a vulnerability management tool. Finally, Expert 2 mentioned the static analysis tool Find Security Bugs which is only usable for Java. All of the gathered tools are SAST tools that are placed within the continuous integration or deployment pipeline of a project and help to find rather simple and well-known vulnerabilities (Expert 4 and 5). They can also be included in pull requests which only succeed if no vulnerabilities were found (Expert 5). The identified advantages are that it saves time and cost (Expert 6 and 2), it is automated (Expert 2) and it helps to improve security and quality (Expert 8). The disadvantage that comes with the usage of such tools is that the rules define the results (Expert 2 and 8). Furthermore, the false positives and warnings that were identified as a problem beforehand in the literature of SAST tools can lead to the "Manager Problem" as described by Expert 6. This problem describes that managers, who do not fully understand the issues, enforce that every issue is to be solved regardless of its importance, costing money and time. In addition, as described by Expert 2, too many issues can lead to developers simply marking them

as fixed without evaluating them further.
The individuals, who are responsible for the introduction and correct usage of these tools, are often provided by the consulted company to ensure the companies security standards (Expert 5, 6, 7, and 8). Expert 8 stated the existence of team members with a pure focus on security when working for one specific client. These persons are called 'Security Matters Experts' and deal with anything related to security within a project. Expert 5 and 7 both stated that the architect has the responsibility for such tools. Finally, the answers indicate that often only the development team is responsible for addressing the issues found through the provided tools.

The majority of the experts mentioned that code reviews and penetration tests were performed (Expert 2, 5, 6, and 8). While code reviews are often done by team internal developers through for example pull requests or pair programming, which was also described as the "Four-Eye Principle", the penetration tests are mostly provided by external service providers. However, code reviews can also be carried out by external developers as described by Expert 8. This process is executed for every line of code that is written (Expert 5, 6, and 8). Expert 6 stated that the penetration tests are carried out every six months or before the release of new features. Moreover, most experts that mentioned code reviews seemed rather positive to find most of the common mistakes through this second glance. Both Expert 5 and 6 described code reviews as the first level of security enhancement and penetration tests as a second level.
In general, the results of these processes can lead to a release being delayed. This decision is made by the client the software is developed for, the leading architect, and the development team. Expert 5 also mentioned a quality gate that is provided through Sonarqube that determines whether the deployment continues or not. Expert 8 again mentioned the 'Security Matters Experts' to be responsible for these processes.
To conclude, one can say that within the industry, code is reviewed at least twice, by the person who wrote it and an additional individual and that penetration tests are carried out to make sure the software is secure. The code reviews are most often done through the team or the architect and individuals with pure security roles can only be found rarely.

7.2 Automated Tools

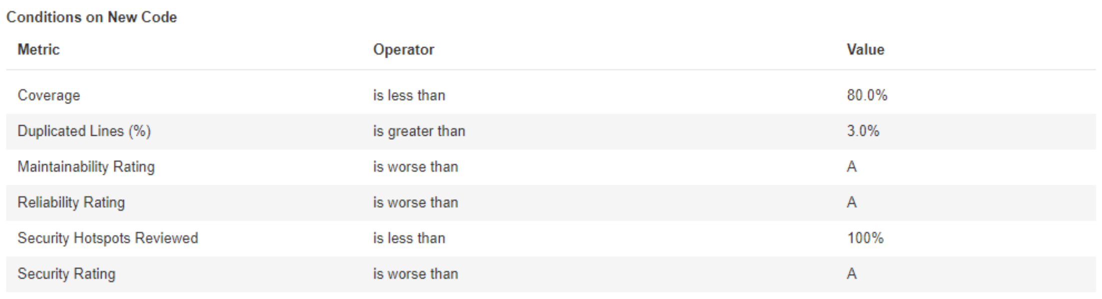
Many different tools can be found that support recognizing vulnerabilities within web applications. NIST and OWASP both offer lists for different SAST tools [156], [157]. OWASP also has a list for DAST tools [103]. Even so, most of these tools are only commercially available and therefore not accessible within this thesis. Hence, only publicly available tools will be evaluated. Evolved from the results of the expert interviews, the following decisions were made. Since only SAST tools were used to ensure code security, the focus will be put on those. DAST tools were not mentioned throughout the interviews, which is most likely because of the high level of configuration that is necessary for an appropriate test result [158]. This was also approved by Expert 2 during the interview. In particular, the tools that were mentioned the most, which are Sonarqube, which was also declared state of the art, and also LGTM will be reviewed more in detail.

7.2.1 Sonarqube

This subsection is concerned with the SAST tool Sonarqube, which helps to provide code quality as well as code security. In the beginning, general information will be provided and then its constraints will be evaluated.

7.2.1.1 General Information

Sonarqube is a SAST tool that was developed by SonaSource with its current version 8.7.1 being released in March of 2020. It evaluates the source code against a predefined set of rules. Doing so, it can detect bugs, vulnerabilities and code smells within the code. This tool can for example be included in the build process, or through SonarLint within the IDE, and hence works automated. It can be used to either evaluate the whole project or single pull requests. The rules that Sonarqube defines by default consist of different attributes. First of all, the language this rule applies to and its type is defined. The types can be bug, vulnerability, code smell, or security hotspot and will be evaluated further in the following paragraph. Tags can be added for more information. The rules originate from a repository, which is SonarQube per default, and have a certain severity. The severity of a rule is determined by the impact that it has on the system and the likelihood of it appearing. It ranges from minor with low impact and likelihood to blocker. The status of the rule defines if it is deprecated, in beta, or ready to use and the availability date describes the date when the rule was added. Finally, the template of the rule exists that can be adapted for custom rules and the quality profile that the rule is assigned to is stated.
Furthermore, Sonarqube offers quality gates, which were for example used by Expert 5, to stop the build process or pull requests in case certain thresholds are exceeded. By default, the built-on *Sonar way* quality gate is used. The conditions the default quality gate provides can be seen in Figure 7.1. The coverage defines the portion of code that is covered by tests and duplicate lines define how often duplicated code is present. The next four conditions are related to the different types of rules that were described



Conditions on New Code

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Hotspots Reviewed	is less than	100%
Security Rating	is worse than	A

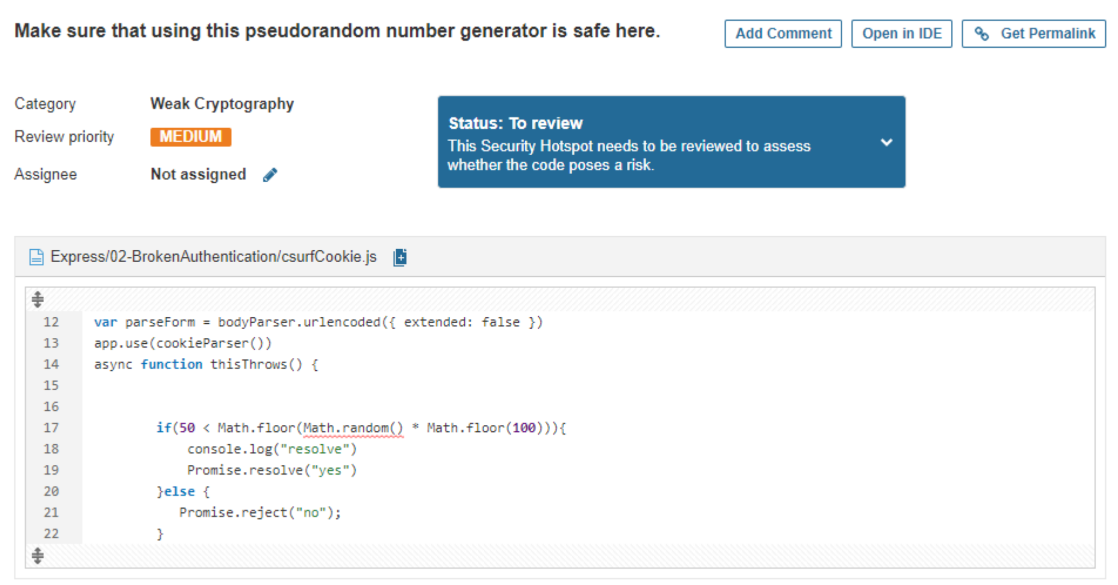
Figure 7.1: Sonarqube Default Quality Gate

earlier. For a detailed insight on how the ratings that are described in the following are calculated, please see [159]. The maintainability rating is calculated from the number of rules that recognize a code smell within the code. Code smells are for example code that is commented out, unnecessary semicolons in JavaScript, or classes that implement `Serializable` in Java that do not contain a `serialVersionUID`. For Java projects, there are 398 and for JavaScript 141 rules that recognize code smells. Moreover, the reliability rating is calculated through the presence and the severity of bugs. These bugs for example include the failure of tests or the usage of non-existing operators. For Java 155 rules are available and for JavaScript 52. Finally, the two security-related rule types, security hotspot and vulnerability, are considered. For the former, all found security hotspots, which are code blocks that are likely to have an impact on security, need to be reviewed. Figure 7.2 displays an example for such a process. The highlighted code requires to be reviewed and its status must be set to either fixed or safe. In contrast, the security rating is depending on the rules of type vulnerability. These include the for example the robustness of the used cipher algorithms or signing JWTs with secure algorithms. For Java 42 rules of type vulnerability exist and for Java only 14. The default security-related rules will be further evaluated in the following.

7.2.1.2 Evaluation

In order to evaluate the security-related rules that Sonarqube proposes, the rules themselves as well as their proposed security advices are compared against what is found within the literature. In addition, the code examples described in section 4.2 were used to gather further insights for certain vulnerabilities. In the evaluation process, the different rules assigned to a vulnerability will be reviewed.

For both Java and JavaScript, a general rule called ’Formatting SQL queries is security-sensitive (S2077) is available that searches for the insecure creation of queries or



Make sure that using this pseudorandom number generator is safe here.

Category Weak Cryptography
Review priority MEDIUM
Assignee Not assigned

Status: To review
This Security Hotspot needs to be reviewed to assess whether the code poses a risk.

Express/02-BrokenAuthentication/csurfCookie.js

```
var parseForm = bodyParser.urlencoded({ extended: false })
app.use(cookieParser())
async function thisThrows() {


        if(50 < Math.floor(Math.random() * Math.floor(100))){
            console.log("resolve")
            Promise.resolve("yes")
        }else {
           Promise.reject("no");
        }
```

Figure 7.2: Sonarqube Security Hotspot Example

commands. However, the rules do not find all vulnerable code in Java. Only the insecure implementation of JDBC using only Java was recognized in the code examples from Sonarqube. It does not find the insecure implementation of Spring's JDBC implementation and its `JdbcTemplates` class. In JavaScript, the insecure query, in which none of the *mysql* native security mechanisms is used, resulted in a security hotspot. The insecure usage of the *sequelize* libraries does however not lead to any security issues in Sonarqube.

For the vulnerability Broken Authentication multiple rules exist which align with the literature. First of all, for both Java and JavaScript a rule exists, which deals with session fixation by stating that a new session should be created when a user logs in (S5876). In Java, the rule matches with the solutions in chapter 6, but in JavaScript, only an example based on the library *passport* is considered. Thus, no security issue was found even if the `regenerate` method is not called after a user authenticates. Furthermore, both languages have a rule which checks whether hard-coded credentials are present (S2068). In Java, a further rule can be found, which enforces the usage of a secure password which is however only triggered if the password is left empty (java:S2115). Finally, another rule only applicable for Java is concerned with the usage of a password encryptor (java:S5344). In particular, the password encryptors bcrypt, PBDKDF2, and argon2 that are recommended in the literature. The other aspects covered in this thesis, like general authentication or password policies are not covered by the rules.

For Sensitive Data Exposure in total 28 rules exists. Rules that are related to cryptography are the following: robustness of cipher algorithms (S5547) and cryptographic keys (S4426), the usage of secure encryption modes and padding schemes (S5542), the inclusion of unpredictable salts in hash functions (S2053), the usage of pseudo-random number generators (S2245), signing JWTs with strong cipher algorithms (S5659) and finally the usage of non-standard cryptographic algorithms (S2257) or weak hashing algorithms (S4790). All of these align with what is proposed in the literature. Furthermore, cookies should be enhanced with the `Secure` flag (S2092) and in JavaScript, the usage of the `no-referrer` policy is recommended whose example is displayed with the library *helmet* (S5736). The latter does not match completely with the literature but the rule proposes the usage of the most secure configuration possible. Further rules exist, but they do not cover contents within this thesis and are therefore not evaluated.
For the XXE vulnerabilities, the general rule exists that the XML parsers should not consider external entities (S2755). It provides different examples of parser configurations.
Most of the rules for the Broken Access Control vulnerability are rather general. Relevant rules could only be found for Java. These rules disallow the usage of both safe and unsafe HTTP methods (S3752) which enforces that only one HTTP method is allowed and also emphasizes the implementation of strong decisions (S5808) by defining restrictions to custom `vote` and `hasPermission` methods.
The rules that belong to the Security Misconfiguration vulnerability are highly overlapping with Sensitive Data Exposure. Therefore only the non-overlapping rules will be evaluated. The first rule for Java is that the URLs within `HttpSecurity` should be ordered in a way that URLs containing /** should be put lastly (java:S4601). This rule should rather be positioned in the Broken Access Control vulnerability. Furthermore, the initialization vectors for the chaining of cipher blocks should be unpredictable (java:S3329) which is according to the literature but should rather be positioned in Sensitive Data Exposure. Finally, some rules are applicable for both Java and JavaScript. The first rule makes sure that CSRF tokens are used (S4502). In Java, the code example for the compliant solution is incorrect because it shows Spring's default CSRF tokens being disabled. In JavaScript, the guidance is given based on the *csruf* library and makes sure the library is used correctly. In the context of this thesis, the rule should be assigned to the CSRF vulnerability. Moreover, the `X-Powered-By` header should be disabled in both languages (S5689). For JavaScript, the solution provides guidance for the library *helmet* which was proposed in the solutions of this thesis. The final common rule is concerned with CORS, by disallowing it to be set too permissive. In the code examples, the code `app.use(cors())` did for example already result in a security hotspot, reducing the possibility that an inexperienced developer is using the *cors* library without being aware of the security concerns. On the other hand, further rules only for JavaScript exist. These are related to HTTP headers like the `X-Content-Type-Options`

which must be set to `noSniff` (javascript:S5734) and the CSP needs to be set and contain the `frame-ancestors` directive (javascript:S5728 and javascript:S5732). Both proposed solutions by Sonarqube make again use of *helmet* and are thus as described in the solutions. Further rules exist but are not evaluated because they do not align with the topics covered in the thesis.
In XSS the rules do not result in a secure solution. First, the `HttpOnly` flag of the cookies should be set to protect them from XSS attacks (S3330). The second rule is concerned with the disabling of automatic escaping of certain template engines (S5247). This rule does not consider any of the nowadays commonly used web frameworks like React, Vue.js, or Angular. Thus, if these are chosen as the client-side framework and any of the insecure methods that these provide are used, Sonarqube will not highlight their security concerns. Also, for example the sanitization of server output is not considered within the rule set.
The three rules which can are applicable for the Insecure Deserialization vulnerability do not describe a general insecure deserialization process. They only describe specific use cases with the *jackson* library or using LDAP. A general rule could for example search for the `readObject` method inside of Java code.
These rather specific rules are also used in the vulnerability Using Components with Known Vulnerabilities, evaluating the insecure creation of temporary files and the configuration of OpenSAML2.
For Insufficient Logging and Monitoring also only Java rules are available. This rule (java:S4792) is applicable for the most common logger in Java: *log4j*, *java.util.logger* and *logback*. This rule evaluates whether debugging logs are logged, data is only stored locally or the logging mode is chosen too restrictive so that too much information is filtered out.
The DoS vulnerability is covered by restricting the length of content that can be sent in a request (javascript:S5693) and through the restriction of slow regular expressions (java:S5852).
The other vulnerabilities were either already covered within the vulnerabilities above or are not provided with rules.

7.2.2 LGTM

In the following general information about LGTM will be provided and also its built-in constraints for Java and JavaScript projects will be evaluated.

7.2.2.1 General Information

LGTM is a SAST tool that was developed by Semmle and is nowadays also integrated into Github. Its current version, 1.27.0, was released in March of 2020. While being only commercially available if it is used for private or commercial repositories, the website `lgtm.com` provides free analyses of open source or public repositories. Furthermore, if used with Github, it can be integrated into pull requests and thus be executed in an automated manner. Its name, LGTM, is an abbreviation for 'Looks Good To Me'. The backbone of LGTM is a technology called CodeQL that can be used to define a query for a programming language. These queries are similar to the rules that Sonarqube defines. They are also assigned to a query pack, which is a set of queries that belong together. In addition, each query has an identifier, the language it can evaluate, the severity, tags, and, which is an addition to Sonarqube's rules, a Boolean which determines whether the results of the queries are displayed by default or not. The query itself is made out of QL, which is the query language used in CodeQL. An example of such a query can be seen in Listing 7.1. The example searches for comments in JavaScript code that start with 'TODO'. Thus, queries can be customized or developed completely from the ground up. [160]

```
import javascript

from Comment c
where c.getText().regexpMatch("(?si).*\\bTODO\\b.*")
select c
```

Listing 7.1: Example of a CodeQL Query [160]

7.2.2.2 Evaluation

LGTM uses its built-in queries to evaluate the security of source code. For Java, there are 164 different queries and for JavaScript 174 queries are evaluated. Because of this high number of queries, only the ones related to the vulnerabilities covered in this thesis will be evaluated. Again, these proposed constraints will be compared to the literature and also, if applicable, validated through code examples.

Starting with Java, SQL injection related queries do also exists (java/sql-injection and java/concatenated-sql-query) which both propose the same solution that is to use prepared statements as also defined in the literature. Next, multiple queries can be found that deal with cryptography. First of all, the usage of random number generators is monitored (java/predictable-seed and java/random-used-once) which was not further

evaluated in the thesis, and the usage of cryptographically safe algorithms (java/weak-cryptographic-algorithm and java/potentially-weak-cryptographic-algorithm) is considered which recommends the usage of at least AES-128 or RSA-2048. Even so, the code example does not contain any information about the used cipher mode which is declared important within the literature. Its description is emphasizing not to use the ECB mode, which is correct, but no guidance on proper cipher modes is given. A query being concerned with hard-coded credentials is also present (java/hardcoded-credential-api-call). Guidance for XXE is also existing (java/xxe). In addition, a query exists which is concerned with XSS (java/xss). The query searches for a response of a `HttpServlet` that contains user input in the send HTML. Since this is not the standard way of returning responses when using Spring Boot and also because Spring Boot, in the context of this thesis, is primarily seen as a server-side framework, the query will not find such vulnerabilities in Spring Boot. Furthermore, deserialization is considered (java/unsafe-deserialization). This query implements the proposed improvements of the corresponding Sonarqube rule that were described above. It disallows the usage of the `readObject` method and suggests the usage of primitive data types. The usage of a secure library as described in chapter 6 should also be considered. Also, a CSRF query exists (java/spring-disabled-csrf-protection) which, in contrast to Sonarqube, has a correct solution. The `Secure` setting of the cookies is enforced (java/insecure-cookie). Furthermore, also error handling is covered by LGTM by evaluating whether stack traces are returned as a response (java/stack-trace-exposure). Also, a queries exist that are related to input validation. These do not apply to the context of this thesis. The first query (java/insecure-bean-validation) disallows certain features in validators that can lead do code execution. Further queries, that can be connected to the Input Validation vulnerability, check for secure handling of user input in connection with arrays (java/improper-validation-of-array-index and java/improper-validation-of-array-construction), which is only partially relevant in the context of this thesis. Finally, a query exists that forces user-provided redirects to be compared against a whitelist of valid redirects (java/unvalidated-url-redirection). This is exactly as the literature recommends.
For JavaScript, there are also queries related to the Injection vulnerability. Firstly, SQL injections are covered, again emphasizing the usage of parameterized queries (js/sql-injection). The query description does only contain examples for the *pg* (PostgreSQL) library but it also found issues in the code examples for the *mysql* library. The usage of the `escape` method which is provided by the *mysql* library was however marked as an vulnerability even tho it has the same outcome as the usage of the prepared statements. In addition, this vulnerability was also found if *mongoose* or *mongodb* were used and data input was directly inserted into the query. Furthermore, a general code injection query can be found, which highlights the problems that the

usage of the `$where` operator involves. To handle the access control, the CORS query evaluated whether a whitelist was used to allow access from another origin (js/cors-misconfiguration-for-credentials). Furthermore, the queries regarding the safety of the cryptographic algorithm and secure randomness also apply to JavaScript. An XXE query exists enforces that external entities are disabled in the selected library (js/xxe). The greatest variety of queries can be found for the XSS vulnerability. At the beginning, for each type of XSS attack the corresponding prevention techniques are defined in a query (js/xss-through-dom, js/stored-xss and js/reflected-xss). A general XSS query is also present (js/xss). Further, escaping and sanitization is evaluated by making sure that certain characters are escaped (js/incomplete-multi-character-sanitization and js/incomplete-sanitization) and potentially evil HTML elements are removed (js/incomplete-html-attribute-sanitization). However, during tests, none of the insecure methods of React, Vue.js, or Angular resulted in a security warning. Insecure deserialization is also covered (js/unsafe-deserialization) by showing an example using the *js-yaml* library. Moreover, the usage of the library *csurf* is recommended (js/missing-token-validation) and multiple DoS related queries exists which are concerned with rate limiting (js/missing-rate-limiting) and the safe usage of regular expressions (js/redos and js/regex-injection). Finally, multiple queries can be found that deal with redirection. These include removing insecure schemes like `:javascript` (js/incomplete-url-scheme-check), sanitization of an URL (js/incomplete-url-substring-sanitization) and also the comparison of the potential redirect URL to a whitelist (js/client-side-unvalidated-url-redirection and js/server-side-unvalidated-url-redirection).

8 Discussion

This chapter lists the key findings of this thesis and also the different limitations that were faced during the research.

8.1 Key Findings

This section summarises the findings of chapter 6 and chapter 7 by going through every vulnerability. Table 8.1 displays an overview of the key findings. The table also displays a perceived coverage, which is represented through Harvey Balls. The coverage is defined by the number of security-related tasks a software developer still has to perform to make a web application secure. A mapping of the symbols and their levels of coverage can be found in Figure 8.1. It defines the three categories 'No/minimal Coverage', 'Partial Coverage' and 'Up to Full Coverage'. The first category defines that the solutions are only of a supporting nature with the security being completely up to the developer. The second category includes vulnerabilities whose tasks are only covered in certain aspects. They still require the implementation or configuration of security-related functionalities by developers to achieve complete security. The final category declares that a vulnerability is to a high probability covered through the proposed solutions and does not, or only to a low extend, require knowledge about the corresponding vulnerability or its prevention techniques. These values are built on the personal opinion of the author that formed itself through the evaluation process of the solutions and the expert interviews.

Starting with the Injection vulnerability, one can say that the selection of frameworks and libraries is not necessarily having an impact on security. The proposed solutions do all offer a secure way to query data, with Express requiring an additional sanitization library when working with MongoDB. Also, the majority of libraries do try to educate the user of potential security concerns when using possible insecure querying methods. These insecure ways are in most cases a method that allows direct access to the database with SQL. Depending on the chosen interface technology when querying data with SQL (ORMs or official libraries), the usage of an insecure query procedure is often less likely when choosing an ORM because their general procedure does not allow the

○	◑	●
No/minimal Coverage	Partial Coverage	Up to Full Coverage

Figure 8.1: Coverage Categories

direct execution of SQL commands. In contrast, the insecure query procedure of the *mysql* library is not only the one described first in the documentation but is also easier to implement than a secure solution. These differences could not be detected when working with MongoDB and NoSQL. Nevertheless, the official libraries have a higher coverage of constraints in Sonarqube or LGTM. Therefore, this vulnerability is coverable if one of the described native functionalities or libraries is used in combination with a SAST tool.

For Broken Authentication, differences in the provided security can be found. Concerning the handling of passwords, within both the Java and the JavaScript environment, implementations exist for the recommended hashing algorithms. Even so, the configurations of these algorithms do not always match completely with the literature, making them still require knowledge in the subject. The usage of bcrypt is recommended since it is per default configured correctly, while argon2's configuration does not match the recommendations entirely. Through the usage of additional SAST tools, these configurations can be further validated, covering this issue almost completely. The password policy can be defined through libraries in both JavaScript of Spring Boot, but they only propose a framework for setting constraints rather than a concrete secure default implementation. Hence, zxcvbn should be chosen to restrict the usage of unsafe passwords. However, a developer still needs to react according to the results of zxcvbn. Thus, only partial coverage can be achieved.
The Spring Boot framework is offering a standardized way of implementing authentication procedures and extends them through native or third-party libraries. In addition, it is also per default providing basic authentication restricting all resources. Thereby, one can say that the general authentication process is covered if Spring Security is used. Express, on the other hand, does not include any authentication method natively and is solely dependent on the usage of libraries. Therefore, authentication, in general, has to be defined by the developer, causing no coverage. All the reviewed JWT libraries were secure according to what was found in the literature by avoiding possible attacks through the `none` algorithm. Furthermore, session cookies were also implemented securely but were lacking automated session management in Express causing only partial coverage. Spring Boot is handling sessions almost completely automated and is

therefore covering the vulnerability.
In conclusion, developers receive different solutions to handle authentication adequately. Nevertheless, authentication is highly dependent on the underlying system and therefore is not holistically solvable through framework-native solutions or libraries. Thus, the vulnerability is not completely coverable but a high extend of security-related efforts is carried out by existing solutions.

In Sensitive Data Exposure again differences in security could be determined. When it comes to cryptography, Spring Boot is natively providing secure methods (`stronger`) that are simply callable by a developer without the need for any configuration. It should be mentioned that the method `standard` should only be used if further authentication is used. In contrast, the *crypto* module of Node.js, which can natively be used by all Node.js-based frameworks, is not providing a secure API. The configurations such as the used algorithm, the salt, and the initialization vector have to be set by a developer which can lead to security issues. Thus, the usage of the *crypto-js* should be used which offers an API that is similar to Spring Boot's `standard` method. These tasks are also supported by the reviewed SAST tools through a variety of constraints. One can say that the solutions cover this task. Furthermore, the general minimization of data that is used by the system is a constraint, which was often emphasized by the experts, cannot be solved through framework or library-based solutions. Hence, developers and architects need to consider this during the design since no solutions were found to deal with this highly on the underlying system depending task.
Moreover, key management is often abstracted through providers or third-party solutions. The frameworks offer libraries to connect to these external solutions. However, the leakage of a carelessly handled key is still possible causing the key management to only be covered partially.
In conclusion, this vulnerability is also not covered entirely. The support in cryptography is for all frameworks sufficient when the usage of either Sonarqube or LGTM is considered. However, key management still requires knowledge on how to handle keys.

XXE is natively only possible in Spring Boot since JavaScript cannot process XML without the usage of an additional library. In general, the solution is to disable external entities which are not always configured in that way by default in either Java or libraries for JavaScript. However, both Sonarqube and LGTM contained a constraint that deals with this issue for the developer, supporting the coverage further.

The vulnerability Broken Access Control is not completely solvable through frameworks or libraries. They do only offer a framework in which a developer can define the required behavior. In the end, the developer can make use of native roles, permission, etc. in Spring Boot and in Express through the usage of libraries, but the access restriction needs to be implemented correspondingly to the underlying system. Thus, no library or tool can be used to holistically implement access control. In addition, also no constraints provided by the SAST tools were found that would help in this regard. The CORS header can however be limited through Sonarqube and LGTM, alarming developers about the security issues that can arise through this HTTP header. Even so, it is not proposing a solution for each web application, making this task only covered partially. Based on these reasons, one can say that this vulnerability is only supported through the existing solutions but not covered.

In the Security Misconfiguration vulnerability, general security hardening was considered. Since this task is mainly handled in the build process, the frameworks and libraries do not offer a solution. Hence, developers need to undertake the configurations by themselves. For error handling, again all frameworks provide a solution to handle errors in general. Even so, error messages and the logic that is needed to recover from the error have to be provided by the developer, making this concern not covered by existing solutions. HTTP headers were also covered. With Express' *helmet* library, a secure configuration could be achieved while Spring Boot's default configuration is not without flaws. The incorrect setting of the `X-XSS-Protection` header and the absence of a CSP are security issues that need to be further addressed by the developer since the SAST tools mainly cover JavaScript solutions. The vulnerability is therefore only partially covered for Express.
Security Misconfiguration is a vulnerability with an enormous scope that goes beyond the application. Therefore, the support for this vulnerability is, except for HTTP headers, rather limited. This leads to only partial coverage.

XSS is providing different levels of security. React and Vue.js are both escaping values in their standard data binding, making it secure to put user input inside of it. React is going even further by renaming the `innerHTML` function and making it harder to use. Angular is, except for one rather specific use case, sanitizing everything that is bound to the DOM. From a security perspective, escaping is the preferred choice since it completely disables HTML to be interpreted as such. However, it comes with a limitation in functionality. If this functionality is needed (like for rich text editors) the reviewed sanitization libraries can be used, that all use a whitelist approach which is recommended in the literature. These libraries can also be used in Spring Boot or Express to handle output sanitization. Even so, as already stated in the solutions, sanitized input can be altered again or unsanitized input can get into the system. SAST tools can be used and especially LGTM implements many queries that help to deal with general XSS vulnerabilities. Unfortunately, none of the security concerns of the web frameworks is considered in the constraints. Therefore, the coverage of React and Angular can be seen as up to complete, while the other frameworks suffer from the mentioned disadvantages. Furthermore, the coverage of sanitizers should be further evaluated.

The vulnerability Insecure Deserialization is mainly an issue in Java. Hence, if JavaScript-based frameworks are chosen, this vulnerability can be neglected in most cases. In Java, the library *Apache Commons IO*, or a custom implementation of the 'look-ahead' approach should be used to implement a whitelisting of allowed classes. Consequently, security by default is rather minimal since the outcome highly depends on the knowledge of the developer. However, LGTM is offering queries for both Java and JavaScript, which stop developers from deserializing insecurely. In conclusion, one can say that the vulnerability is therefore only partially covered for Java. Insecure deserialization is generally recognized by tools, but the concrete prevention measures are still developer dependent. In case JavaScript is used, the vulnerability is covered through the usage of JSON.

Using Components with Known Vulnerabilities is coverable through the existing libraries. They check whether a dependency is connected to any known vulnerabilities. Hence, it is supporting the progress of finding such issues. Since most of the found libraries do not explicitly name their sources and the experts stated that also the research of vulnerabilities is required, the coverage of the vulnerability is only partial.

For Insufficient Logging and Monitoring, the reviewed libraries offered partial coverage. Express' *morgan* library and Spring Boot's native logger can be used to log many of the in the literature required information. However, similar to error handling, most libraries only propose a framework in which the developer has to enter the information required to make the log useful. In contrast to error handling, these solutions provide a majority of required information in their default log configuration. Furthermore, of equal importance is the monitoring of the created logs. Which is a task that, even if it is supported by third-party tools, still needs the attention of a person to understand the logs and especially the severity of certain ones. Thus, one cannot say that this vulnerability can be fully covered by the chosen frameworks, libraries, and automated tools.

The first vulnerability not included in the OWASP list [12] is called CRSF and receives attention natively in Express and Spring Boot. While Express' solution does not completely align with the literature and is therefore only covering the vulnerability partially, Spring Boot is offering an out-of-the-box solution according to the literature. Thus, the vulnerability can be covered if Spring Boot is used.

On the other hand, DoS is a vulnerability with greater scope. The solutions that were found mainly focus on the limitation of accepted requests from one specific source. For Express the safe usage of regular expressions was considered as well. Since DoS is rather resolved on the network layer, this vulnerability only receives no to minimal coverage.

Input Validation can be achieved through the *validator* library in Node.js or the *spring-boot-starter-validation* of Spring Boot. Again, a framework is given to the developers by defining methods that can be used to validate data. These methods do however need to be used in a meaningful manner to make untrusted data valid. Additionally, SAST tools do not offer any support. Thus, the coverage cannot be guaranteed.

Finally, for Open Redirects and Forwards, no framework or library-based solution can be determined since the best approach is the usage of a whitelist. LGTM is providing multiple queries to evaluate whether redirects are done insecurely. Therefore, even if a developer does not use a recommended whitelist, the SAST tool will find this vulnerability in certain cases. Hence, one can say this vulnerability is partially covered.

8.2 Limitations

During the research, the following limitations were encountered.
Literature: The literature research of this thesis also included grey literature whose information could not be holistic or contain errors. Furthermore, only one person performed the literature research which can result in bias through the selected search terms and literature in general.
Expert Interviews: The selected experts were all within one company, the msg systems ag, and therefore the gathered insights could be biased and also one-sided. To get a broader view, further companies working in the area of web applications or information systems should be questioned.
Evaluation: Since the evaluation of solutions is based on the literature research and the expert interviews, the above-described limitations also apply for the evaluation. In particular, the searched solutions highly depend on the search strings that were chosen and therefore offer no holistic view of all available functionalities within frameworks or libraries that exist to deal with vulnerabilities. Further, what will also be discussed more in the outlook, only built-in constraints from Sonarqube and LGTM were evaluated. Finally, no commercial tools could be included in the research, thus further tools with most likely high potential were not reviewed.

Vulnerability	**Solutions**	**Supporting SAST Tool Constraints**	**Perceived Coverage**
Injection			●
SQL Injection	Express: (mysql), (sequelize) Spring Boot: Spring Data (JDBC)/(JPA)	Sonarqube: S2077 LGTM: sql-injection, java/concatenated-sql-query	●
NoSQL Injection	Express: (mongodb), (mongoose) Spring Boot: Spring Data MongoDB		●
Broken Authentication			◑
General Authentication	Spring Boot: Spring Security		Express : ○ Spring Boot: ●
Password Policy	Node.js: zxcvbn, (password-sheriff) Spring Boot: zxcvbn4j, (Passay)		◑
Password Hashing	Node.js: bcrypt, (argon2), (crypto module) Spring Boot: PasswordEncoder	Sonarqube: S4790, java:S5344 LGTM: js/insufficient-password-hash	●
JWTs	Node.js: jsonwebtoken Spring Boot: io.jsonwebtoken, java-jwt		●
Session Cookies	Express: (cookie-session), (express-session) Spring Boot: (Default session cookies)	Sonarqube: javascript:S5876	Express : ◑ Spring Boot: ●
Sensitive Data Exposure			◑
Cryptography	Node.js: crypto-js, (crypto module) Spring Boot: Encryptor, Decryptor	Sonarqube: S2245, S2257, S4426, S5542, S5547, java:S3329 LGTM: weak-cryptographic-algorithm, potentially-weak-cryptographic-algorithm	●
Data Management			○
Key Management	Node.js: (node-vault) Spring Boot: (Spring Vault)		◑
XXE	JavaScript: Usage of JSON Spring Boot: (Disabling XXE in the used parser)	Sonarqube: S2755 LGTM: xxe	JavaScript: ● Spring Boot: ◑
Broken Access Control			○
Authorization		Sonarqube: java:S5808	○
CORS		Sonarqube: javascript:S5122 LGTM: js/cors-misconfiguration-for-credentials	◑
Security Misconfiguration			◑
Security Hardening			○
Error Handling	General: (Frameworks for error handling) Spring Boot: (Problem)	LGTM: stack-trace-exposure	○
HTTP Header	Express: helmet, no-cache Spring Boot: (Default configuration)	Sonarqube: S5689, javascript:S5728, javascript:S5732, javascript:S5734	●
XSS	React: Escaping, (Renaming of insecure function) Vue.js: Escaping Angular: Sanitization of HTML, URL, CSS Node.js: (DomPurify) Spring Boot: (Jsoup)	Sonarqube: S3330. S5247 LGTM: js/xss, js/xss-through-dom, js/stored-xss, js/reflected-xss, js/incomplete-multi-character-sanitization, js/incomplete-sanitization js/incomplete-html-attribute-sanitization	React: ● Angular: ● Others : ◑
Insecure Deserialization	Spring Boot: (SerialKiller)	LGTM: unsafe-deserialization	JavaScript: ● Spring Boot: ◑
Using Components with Known Vulnerabilities	Node.js: npm, Snyk, Retire.js Spring Boot: Snyk, OWASP Dependency-Check		◑
Insufficient Logging and Monitoring			◑
Logging	Express: morgan Node.js: (winston), (tracer), (bunyan) Spring (spring-boot-starter-logging)	LGTM: js/clear-text-logging	◑
Monitoring			
CSRF	Express: csurf Spring Boot: Native CSRF tokens	Sonarqube: S4502 LGTM: js/missing-token-validation	Express: ◑ Spring Boot: ●
DoS	Express: (ratelimiter), (safe-regex) Spring Boot: (Spring Cloud Gateway), (Bucket4j)	Sonarqube: S5693, java:S5852 LGTM: js/missing-rate-limiting, js/redos, js/regex-injection	○
Input Validation	HTML5: (Form types) Node.js: (validator) Spring Boot: (spring-boot-starter-validation)		○
Open Redirects and Forwards	General: WhitelistNode.js: (validator)	LGTM: js/incomplete-url-scheme-check, js/client-side-unvalidated-url-redirection, js/server-side-unvalidated-url-redirection, java/unvalidated-url-redirection	◑

If a solution is put in brackets, it only covers a part of the vulnerability or only supports the developer and engineering is still required.

Table 8.1: Overview of the Key Findings

9 Conclusion

In this chapter, the thesis is concluded. Firstly, the answers to the research questions will be summarized. Secondly, an outlook for further research is given.

9.1 Summary

The following section summarizes the answers to the research questions that were given in the main part of the thesis.

Research Question 0.1: *Which vulnerabilities are relevant for web applications?*
The vulnerabilities that could be identified through literature research and also expert interviews are the following: Injection, Broken Authentication, Sensitive Data Exposure, XML External Entities, Broken Access Control, Security Misconfiguration, Cross-Site Scripting, Insecure Deserialization, Using Components with Known Vulnerabilities, Insufficient Logging and Monitoring, Cross-Site Request Forgery, Denial of Service, Input Validation and Open Redirects and Forwards. Further vulnerabilities were gathered, which were declared out of scope for this thesis.

Research Question 1: *Which vulnerabilities are covered by framework-native functionalities?*
First of all, Node.js in general offers solutions for cryptography with the *crypto* module which needs secure configuration. Moreover, it deals with the Using Components with Known Vulnerabilities vulnerability through the Node package manager. React, Angular.js and Angular all three provide native solutions for the XSS vulnerability by either escaping or sanitizing user input automatically. Angular also includes validators that can help to deal with input validation. Express deals with logging through the *morgan* library and with CSRF trough the *csurf* library.
Spring Boot on the other hand provides more native coverage of vulnerabilities. It has natively supported the connection to SQL or NoSQL databases and provides partially secure solutions for querying data. Also, cryptography is handled by Spring Boot natively by providing a secure API. Authentication, also including password hashing, and authorization are natively supported. HTTP headers are set partially secure and

logging solutions exist. Finally, DoS protections can be found to a low extend and predefined input validators can be used.

Research Question 2: *How can libraries support the coverage of the vulnerabilities?*
Libraries for Node.js help to deal with injection, authentication, data encryption and authorization, sanitization of input, finding known vulnerabilities, logging, and input validation but do not cover these tasks completely. HTTP headers, on the other hand, are set securely through libraries.
For Spring Boot, fewer libraries exist. These mainly support the setting of a password policy and the implementation of authentication mechanisms such as JWT.
The results of the first two research questions show, that software engineering with security in mind is often still required to make the given solutions secure. The reason for this is that often both, framework-native solutions and libraries, only offer supporting features that need to be used adequately to make a web application secure.

Research Question 2.1: *How can similar libraries be compared with one other?*
Different characteristics of libraries were researched from literature and gathered through expert interviews. The characteristics with the most relevance were then put together in a metric that can be used to distinguish libraries with similar functionality. These characteristics contain the popularity, maintenance, stability and maturity, and also the quality of a library. Furthermore, the security-related characteristics 'Security by Default', 'State of the Art', and 'Documentation' were defined and evaluated through expert interviews.

Research Question 3: *Which tools and processes can be used to reduce vulnerabilities?*
The usage of SAST tools can increase the security of a web application by enforcing certain security rules. These can for example recognize the insecure creation of SQL queries, the insecure sanitization of input, and much more. However, the evaluated tools do often only propose general, but no framework-specific constraints making the support of these tools highly dependent on the chosen technology. The built-in rules for both Sonarqube and LGTM can be used to increase the security in a majority of the examined vulnerabilities. However, only a few vulnerabilities can be completely covered through them like CORS or XXE. Furthermore, DAST tools do not seem to be used commonly nowadays and the processes that are executed are mainly code reviews and penetration testing.

9.2 Outlook

This research gives an overview of existing relevant web application vulnerabilities and how to solve their security issues through the usage of libraries, frameworks, and automated tools. During the research, code examples were created to give a greater understanding of the solutions. To further support developers in their choice, these code examples could be further validated through expert opinions and also enhanced to a completely functioning demo web application that can be used as a guideline. Moreover, since the scope of this thesis was limited, not all aspects of every vulnerability could be examined. In further research, these aspects should be dealt with. Especially, vulnerabilities or subcategories of these vulnerabilities where no solution could be found should be the focus of further research to find ways to support developers. Also, the proposed library metric could be improved to include a single score for the level of maintenance or a total score to improve the comparability of the metric. Further, only the built-in constraints of the SAST tools were evaluated. Since companies can create their own set of constraints, further research could include a questionnaire in this direction. Finally, the evaluation of existing DAST tools and how well developers would accept the usage of such tools should be further evaluated.

A Framework and Library Versions

Framework	**URL**	**Version**
React	`https://www.npmjs.com/package/react`	17.0.2
Vue.js	`https://www.npmjs.com/package/vue`	2.6.12
Angular	`https://www.npmjs.com/package/angular`	1.8.2
Express	`https://www.npmjs.com/package/express`	4.17.1
Spring Boot	`https://spring.io/projects/spring-boot`	2.5.0

Table A.1: Reviewed Frameworks and their Versions

Library/Dependency	Language	**Category**	**URL**	**Version**
mysql	JS	SQL Injection	`https://www.npmjs.com/package/mysql`	2.18.1
sequelize	JS	SQL Injection - ORM	`https://www.npmjs.com/package/sequelize`	6.6.2
mongodb	JS	NoSQL Injection	`https://www.npmjs.com/package/mongodb`	3.6.8
mongoose	JS	NoSQL Injection - ORM	`https://www.npmjs.com/package/mongoose`	5.12.11
mysql-connector-java	Java	SQL Injection	`https://mvnrepository.com/artifact/mysql/mysql-connector-java`	8.0..25
spring-boot-starter-data-jdbc	Java	SQL Injection	`https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jdbc`	2.5.0
spring-boot-starter-data-jpa	Java	SQL Injection - ORM	`https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa`	2.5.0
spring-data-mongodb	Java	NoSQL Injection	`https://mvnrepository.com/artifact/org.springframework.data/spring-data-mongodb`	3.2.1
cookie-session	JS	Session Cookies	`https://www.npmjs.com/package/cookie-session`	1.4.0
express-session	JS	Session Cookies	`https://www.npmjs.com/package/express-session`	1.17.2
password-sheriff	JS	Password Policy	`https://www.npmjs.com/package/password-sheriff`	1.1.0
password-validator	JS	Password Policy	`https://www.npmjs.com/package/password-validator`	5.1.1
zxcvbn	JS	Password Meter	`https://www.npmjs.com/package/zxcvbn`	4.4.2
crypto	JS	Cryptography / Password Hashing	`https://nodejs.org/api/crypto.html`	16.2.0
argon2	JS	Password Hashing	`https://www.npmjs.com/package/argon2`	0.27.2
bcrypt	JS	Password Hashing	`https://www.npmjs.com/package/bcrypt`	5.0.1
bcryptjs	JS	Password Hashing	`https://www.npmjs.com/package/bcryptjs`	2.4.3
jsonwebtoken	JS	JWT	`https://www.npmjs.com/package/jsonwebtoken`	8.5.1
Passay	Java	Password Policy	`https://mvnrepository.com/artifact/org.passay/passay`	1.6.0
io.jsonwebtoken	Java	JWT	`https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt`	0.9.1
java-jwt	Java	JWT	`https://mvnrepository.com/artifact/com.auth0/java-jwt`	3.16.0
Bouncycastle	Java	Cryptography / Password Hashing	`https://mvnrepository.com/artifact/org.bouncycastle/bcprov-jdk15on`	1.68
zxcvbn4j	Java	Password Meter	`https://mvnrepository.com/artifact/com.nulab-inc/zxcvbn`	1.5.0
node-vault	JS	Key Management	`https://www.npmjs.com/package/node-vault`	0.9.21
Spring Vault	Java	Key Management	`https://mvnrepository.com/artifact/org.springframework.vault/spring-vault-core`	2.3.2
sax	JS	XML Parser	`https://www.npmjs.com/package/sax`	1.2.4
accesscontrol	JS	Access Control	`https://www.npmjs.com/package/accesscontrol`	2.2.1
rbac	JS	Access Control	`https://www.npmjs.com/package/rbac`	5.0.3
casbin	JS	Access Control	`https://www.npmjs.com/package/casbin`	5.7.1
cors	JS	Access Control	`https://www.npmjs.com/package/cors`	2.8.5
helmet	JS	HTTP Headers	`https://www.npmjs.com/package/helmet`	4.6.0
nocache	JS	HTTP Headers	`https://www.npmjs.com/package/nocache`	2.1.0
clearsitedata	JS	HTTP Headers	`https://www.npmjs.com/package/clearsitedata`	0.2.0
errors	JS	Error Handling	`https://www.npmjs.com/package/errors`	0.3.0
react-error-boundary	JS	Error Handling	`https://www.npmjs.com/package/react-error-boundary`	3.1.3
vue-error-boundary	JS	Error Handling	`https://www.npmjs.com/package/vue-error-boundary`	1.0.3
Problem	Java	Error Handling	`https://mvnrepository.com/artifact/org.zalando/problem`	0.26
errors-spring-boot-starter	Java	Error Handling	`https://mvnrepository.com/artifact/me.alidg/errors-spring-boot-starter`	1.4.0
escape-html	JS	Escaping	`https://www.npmjs.com/package/escape-html`	1.0.3
html-escaper	JS	Escaping	`https://www.npmjs.com/package/html-escaper`	3.0.3
escape-goat	JS	Escaping	`https://www.npmjs.com/package/escape-goat`	4.0.0
xss	JS	Sanitization	`https://www.npmjs.com/package/xss`	1.0.9
dompurify	JS	Sanitization	`https://www.npmjs.com/package/dompurify`	2.2.8
sanitize-html	JS	Sanitization	`https://www.npmjs.com/package/sanitize-html`	2.4.0
owasp-java-encoder	Java	Escaping	`https://mvnrepository.com/artifact/org.wso2.orbit.org.owasp.encoder/encoder`	1.2.0
jsoup	Java	Sanitization	`https://mvnrepository.com/artifact/org.jsoup/jsoup`	1.13.1
node-serialize	JS	Deserialization	`https://www.npmjs.com/package/node-serialize`	0.0.4
SerialKiller	Java	Deserialization	`https://mvnrepository.com/artifact/org.nibblesec/serialkiller`	3.0
Snyk	JS/Java	Dependency Checker	`https://www.npmjs.com/package/snyk`	1.606.0
Retire.js	JS	Dependency Checker	`https://retirejs.github.io/retire.js/`	3.0.0
OWASP Dependency-Check	Java	Dependency Checker	`https://owasp.org/www-project-dependency-check/`	6.1.6
morgan	JS	Logger	`https://www.npmjs.com/package/morgan`	1.10.0
winston	JS	Logger	`https://www.npmjs.com/package/winston`	3.3.3
tracer	JS	Logger	`https://www.npmjs.com/package/tracer`	1.1.4
bunyan	JS	Logger	`https://www.npmjs.com/package/bunyan`	1.8.15
spring-boot-starter-logging	Java	Logger	`https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-logging`	2.5.0
tinylog	Java	Logger	`https://mvnrepository.com/artifact/org.tinylog/tinylog`	1.3.6
csurf	JS	CSRF Token	`https://www.npmjs.com/package/csurf`	1.11.0
CSRFGuard	Java	CSRF Token	`https://mvnrepository.com/artifact/org.owasp/csrfguard`	4.0.0
body-parser	JS	Request Size Limiting	`https://www.npmjs.com/package/body-parser`	1.19.0
connect-timeout	JS	Rate Limiting	`https://www.npmjs.com/package/connect-timeout`	1.9.0
safe-regex	JS	Safe Regex Parsing	`https://www.npmjs.com/package/safe-regex`	2.1.1
Bucket4j	Java	Rate Limiting	`https://mvnrepository.com/artifact/com.github.vladimir-bukhtoyarov/bucket4j-core`	6.2.0
prop-types	JS	Validation	`https://www.npmjs.com/package/prop-types`	15.7.2
express-validator	JS	Validation	`https://www.npmjs.com/package/express-validator`	6.11.1
validator	JS	Validation	`https://www.npmjs.com/package/validator`	13.6.0
ajv	JS	Scheme Validation	`https://www.npmjs.com/package/ajv`	8.5.0
joi	JS	Scheme Validation	`https://www.npmjs.com/package/joi`	17.4.0
yup	JS	Scheme Validation	`https://www.npmjs.com/package/yup`	0.32.9
formik	JS	Validation	`https://www.npmjs.com/package/formik`	2.2.8
airbnb-prop-types	JS	Validation	`https://www.npmjs.com/package/airbnb-prop-types`	2.16.0
formsy	JS	Validation	`https://www.npmjs.com/package/formsy`	0.19.2
vuelidate	JS	Validation	`https://www.npmjs.com/package/vuelidate`	0.7.6
vee-validate	JS	Validation	`https://www.npmjs.com/package/vee-validate`	3.4.6

Table A.2: Reviewed Libraries and their Versions

B Questionnaires

B.1 Demographic

1. Bist du diese Person?
2. Wie lautet deine Berufsbezeichnung
3. Wie lange arbeitest du schon in dieser Rolle
4. Welche Berührungspunkte mit Security hast du?
5. Wie lange arbeitest du schon im Security-Bereich?
    a) Welche security-relevanten Rollen hattest du bereits?

B.2 First Round of Expert Interviews

Vulnerabilities

1. Wie vertraut bist du mit den Schwachstellen von Web Applications?
2. Würdest du sagen, dass die OWASP Top 10 Most Critical Web Application Security Risks (2017) heutzutage noch relevant sind?
3. Welche der Schwachstellen/Risiken von OWASP Top 10 sind aus deiner Sicht die relevantesten?
4. Mit welchen Schwachstellen/Risiken außerhalb der OWASP Top 10 hast du häufig Kontakt?

Web Frameworks

1. Wie vertraut bist du mit Javascript basierten clientseitigen Webframeworks wie Angular, React und Vue.js?
2. Wie vertraut bist du mit Javascript basierten serverseitigen Webframeworks wie Express?
3. Wie vertraut bist du mit Java basierten serverseitigen Webframeworks wie Spring Boot?
4. Gibt es deiner Meinung Webframeworks die nativ mehr Schutz vor Schwachstellen bieten als andere?
    a) Wenn ja, welche Webframeworks würdest du bevorzugen?

5. Vertraust du auf die Sicherheit von Webframework nativen Schutzmechanismen?
    a) Validierst du, ob diese Mechanismen wirklich sicher sind?

Libraries

1. Stell dir vor, du müsstest eine sicherheitskritische Funktion implementieren und möchtest hierfür eine Library verwenden. Deine Recherchen ergeben, dass es mehrere Libraries gibt, welche eine ähnliche Funktion implementieren.
    a) Anhand von welchen Kriterien ziehst du eine Library einer anderen vor?
    b) Wie validierst du die Sicherheit dieser Libraries?
2. Ich habe folgende Kriterien erstellt, um die Sicherheit einer Library zu bewerten.
    a) Wie würdest du diese einschätzen?
    b) Welche Kriterien fehlen?
3. Wann verwendest du keine Library um sicherheitskritische Funktionen zu implementieren?
    a) Was sind Gründe dafür in manchen Fällen keine Library zu verwenden?

Web Frameworks and Libraries

1. Injection
    a) Wie vertraut bist du mit „Injection“?
    b) Verwendest du die „offiziellen“ Libraries von Datenbankmanagementsystemen wie z.B. mysql und mongodb oder Object Relational/Document Mapper und Query Builder?
        i. Welche Library verwendest du und weshalb?
    c) Meine Recherchen haben ergeben, dass solche Libraries oft einen nicht sicheren Weg bereitstellen um Daten abzufragen wie z.B. Raw Queries in Sequelize oder query(‘sql query‘) in mysql.
        i. Was tust du, um die Verwendung von solchen Abfragen zu verhindern?
        ii. Verwendest du auch Source Code Analysis Tools?
    d) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?
2. Broken Authentication
    a) Wie vertraut bist du mit „Broken Authentication“?
    b) Passwörter
        i. Nutzt du Libraries um die Passwortrichtlinien festzulegen?

        ii. Verwendest du unterschiedliche Passwort Hashing Libraries abhängig vom Kontext?
            A. Wann verwendest du Peppering?
        iii. Bei dem Zugriff auf welche Funktionen ist deiner Meinung nach eine Multifactor Authentication gerechtfertigt?
    c) Authentifizierung
        i. Welche Libraries nutzt du, um Authentifizierung zu implementieren? (Token, Session Cookies, OAuth, ...)
    d) Allgemein
        i. Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?

3. Sensitive Data Exposure
    a) Wie vertraut bist du mit „Sensitive Data Exposure“?
    b) Wie beschützt du Daten, die von Client zu Server transportiert werden?
    c) An welchen Stellen verschlüsselst du Daten und mit welchem Algorithmus bzw. mit welcher Library machst du das?
    d) Gibt es Libraries, die du zum Managen von Keys verwendest?
    e) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?
4. Broken Access Control
    a) Wie vertraut bist du mit „Broken Access Control“?
    b) Nutzt du Libraries für die Implementation von Access Control?
    c) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?
5. Security Misconfiguration
    a) Wie vertraut bist du mit „Security Misconfiguration“?
    b) Konfiguration
        i. Gibt es Libraries, die die Konfiguration von sicherheitsrelevanten Einstellungen vereinfacht?
            A. Nutzt du andere Methoden um die korrekte Konfiguration zu garantieren?
    c) Error Handling
        i. Nutzt du Libraries um Fehlermeldungen mit möglichst wenig Informationen auszugeben?
    d) HTTP Header

            i. Welche HTTP Header sind deiner Meinung nach wichtig um die Sicherheit der Web Application zu verbessern?
            ii. Meine Recherchen haben ergeben, dass die Content-Security-Policy (CSP) sehr viel zur Sicherheit beitragen kann.
                A. Welche der Directives sind deiner Meinung nach die wichtigsten?
                B. Welche Directives sollte man nur mit Vorsicht verwenden?
    e) Allgemein
        i. Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?

6. Cross-Site Scripting
    a) Wie vertraut bist du mit „Cross-Site Scripting"?
    b) Nutzt du die Filtermethoden von z.B. React, Vue.js und Angular um bösartigen Input herauszufiltern?
        i. Nutzt du zusätzlich weitere Libraries wie z.B. DOMPurify?
        ii. Nutzt du hier zusätzlich noch eine Content-Security-Policy?
    c) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?
7. Insecure Deserialization
    a) Wie vertraut bist du mit „Insecure Deserialization"?
    b) Verwendest du Libraries wie SerialKiller, SWAT oder NotSoSerial ?
    c) OWASP empfiehlt hier hauptsächlich Monitoring als Schutz vor unsicherer Deserialisierung
        i. Wie gehst du dagegen vor?
    d) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?
8. Insufficient Logging & Monitoring
    a) Wie vertraut bist du mit „Insufficient Logging & Monitoring"?
    b) Nutzt du Libraries um Daten zu loggen?
        i. Welche Library nutzt du und weshalb?
        ii. Welche sicherheitsrelevanten Daten loggst du?
    c) Wie analysiert du die Logs, um z.B. automatisierte Attacken zu identifizieren?
    d) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?
9. Using Components with Known Vulnerabilities

   a) Wie vertraut bist du mit „Using Known Vulnerabilities“?
   b) Nutzt du Libraries um Schwachstellen von verwendeten Komponenten zu identifizieren?
   c) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?

10. XML External Entities
    a) Wie vertraut bist du mit der Schwachstelle „XML External Entities“?
    b) Meine Recherchen haben ergeben, dass External Entities komplett ausgeschalten werden sollten.
       i. Gibt es noch eine weitere Möglichkeit gegen External Entities vorzugehen?
    c) Gibt es weitere wichtige Aspekte, die deiner Meinung nach zu dieser Schwachstelle gehören?

B.3 Second Round of Expert Interviews

Questions regarding used tools:

1. Welche Tools nutzt du um Schwachstellen im Code zu verhindern?
   a) Sind sie automatisch ausführbar, oder müssen sie manuell gestartet werden?
   b) Wo sind diese Tools positioniert?
2. Was lösen diese Tools? Wo muss man selbst noch nachschauen?
   a) Was sind die Vorteile der Tools?
   b) Was sind die Nachteile der Tools?
      i. Wie wird mit False Positives oder zu vielen Warnings umgegangen?
3. Wer wählt die Tools aus? Wann werden sie ausgewählt?
   a) Wer kümmert sich um die Verwendung der Tools?
   b) Wer kümmert sich darum, dass Ergebnisse der Tools auch adressiert werden?
   c) Wer ist verantwortlich das solche Tools genutzt werden?

Questions regarding used processes:

1. Gibt es noch weitere Prozesse die durchgeführt werden um die Sicherheit zu verbessern?
   a) z.B. Code Reviews, Threat Modelling, Security Stories, Threat Poker oder manuelle/ (semi-)automatisierte Penetationtests?

2. Wann und von wem werden die Prozesse durchgeführt?
    a) Wer hat Verantwortung für solche Prozesse?
3. Was lösen diese Prozesse? Und was nicht?
    a) Welche Schwachstellen im Code erhofft ihr damit zu finden?
4. Wer entscheidet wann, dass das Sicherheitsniveau ausreichend ist um Änderungen in Produktion zu deployen?
    a) z.B. ist eine team-externe Freigabe notwendig?

List of Figures

List of Tables

Bibliography

[1] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*, IEEE, 2006, 6–pp.

[2] G. S. Leite and A. B. Albuquerque, "An Approach for Reduce Vulnerabilities in Web Information Systems," in *Proceedings of the Computational Methods in Systems and Software*, Springer, 2018, pp. 86–99.

[3] Smith, Zhanna Malekos and Lostri Eugenia, *The Hidden Costs of Cybercrime*, McAfee. [Online]. Available: `https://www.mcafee.com/enterprise/en-us/assets/reports/rp-hidden-costs-of-cybercrime.pdf` (visited on 05/24/2021).

[4] IC3, *Internet Crime Report 2020*. [Online]. Available: `https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf` (visited on 05/24/2021).

[5] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of XSS sanitization in web application frameworks," in *European Symposium on Research in Computer Security*, Springer, 2011, pp. 150–171.

[6] *Most used web frameworks among developers worldwide, as of early 2020*, 2021. [Online]. Available: `https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/` (visited on 05/17/2021).

[7] R. Battle and E. Benson, "Bridging the semantic web and web 2.0 with representational state transfer (rest)," *Journal of Web Semantics*, vol. 6, no. 1, pp. 61–69, 2008.

[8] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 245–256.

[9] B. Xu, L. An, F. Thung, F. Khomh, and D. Lo, "Why reinventing the wheels? An empirical study on library reuse and re-implementation," *Empirical Software Engineering*, vol. 25, no. 1, pp. 755–789, 2020.

[10] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security testing: A survey," in *Advances in Computers*, vol. 101, Elsevier, 2016, pp. 1–51.

[11] M. Howard and S. Lipner, *The security development lifecycle*. Microsoft Press Redmond, 2006, vol. 8.

[12] OWASP Foundation, "OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks," 2017.

[13] *2020 CWE Top 25 Most Dangerous Software Weaknesses*, 2020. [Online]. Available: `https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html` (visited on 03/06/2021).

[14] Positive Technologies, *Web Applications vulnerabilities and threats: statistics for 2019*, 2020. [Online]. Available: `https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020/` (visited on 03/06/2021).

[15] A. Pano, D. Graziotin, and P. Abrahamsson, "Factors and actors leading to the adoption of a JavaScript framework," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3503–3534, 2018.

[16] F. L. De La Mora and S. Nadi, "Which library should I use?: a metric-based comparison of software libraries," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, IEEE, 2018, pp. 37–40.

[17] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. S. Cruzes, "Myths and facts about static application security testing tools: an action research at telenor digital," in *International Conference on Agile Software Development*, Springer, Cham, 2018, pp. 86–103.

[18] Web Application Security Consortium, *WASC Threat Classification*, 2010. [Online]. Available: `http://projects.webappsec.org/f/WASC-TC-v2_0.pdf` (visited on 03/06/2021).

[19] A. Hevner and S. Chatterjee, "Design science research in information systems," in *Design research in information systems*, Springer, 2010, pp. 9–22.

[20] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.

[21] P. Kumar and R. Pateriya, "A survey on SQL injection attacks, detection and prevention techniques," in *2012 Third International Conference on Computing, Communication and Networking Technologies (ICCCNT'12)*, IEEE, 2012, pp. 1–5.

[22] W. G. Halfond, J. Viegas, A. Orso, *et al.*, "A classification of SQL-injection attacks and countermeasures," in *Proceedings of the IEEE international symposium on secure software engineering*, IEEE, vol. 1, 2006, pp. 13–15.

[23] J. Clarke-Salt, *SQL injection attacks and defense*. Elsevier, 2009.

[24] OWASP Foundation, *Injection Prevention Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html` (visited on 03/12/2021).

[25] A. Ron, A. Shulman-Peleg, and A. Puzanov, "Analysis and mitigation of NoSQL injections," *IEEE Security & Privacy*, vol. 14, no. 2, pp. 30–39, 2016.

[26] A. Ron, A. Shulman-Peleg, and E. Bronshtein, "No sql, no injection? examining nosql security," *arXiv preprint arXiv:1506.04082*, 2015.

[27] OWASP Foundation, *Testing for NoSQL Injection*. [Online]. Available: `https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.6-Testing_for_NoSQL_Injection` (visited on 04/07/2021).

[28] *Operators - MongoDB Manual*, 2020. [Online]. Available: `https://docs.mongodb.com/manual/reference/operator/` (visited on 03/10/2021).

[29] P. Spiegel, *NoSQL Injection*, OWASP Foundation, 2016. [Online]. Available: `https://owasp.org/www-pdf-archive/GOD16-NOSQL.pdf` (visited on 04/07/2021).

[30] R. Hogue, "A Guide to XML eXternal Entity Processing," 2015.

[31] OWASP Foundation, *XML External Entity Prevention Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html` (visited on 04/18/2021).

[32] ——, (2020). Cross Site Scripting (XSS), [Online]. Available: `https://owasp.org/www-community/attacks/xss/` (visited on 03/06/2021).

[33] ——, *XSS Filter Evasion Cheat Sheet*. [Online]. Available: `https://owasp.org/www-community/xss-filter-evasion-cheatsheet` (visited on 03/06/2021).

[34] Hoehrmann, *The 'javascript' resource identifier scheme*, 2010. [Online]. Available: `https://tools.ietf.org/html/draft-hoehrmann-javascript-scheme-03` (visited on 03/09/2021).

[35] OWASP Foundation. (2020). Types of XSS, [Online]. Available: `https://owasp.org/www-community/Types_of_Cross-Site_Scripting` (visited on 03/06/2021).

[36] ——, *Cross Site Scripting Prevention Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html` (visited on 03/09/2021).

[37] A. Klein, "DOM based cross site scripting or XSS of the third kind," *Web Application Security Consortium, Articles*, vol. 4, pp. 365–372, 2005.

[38] OWASP Foundation, *DOM based XSS Prevention Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.html` (visited on 05/21/2021).

[39] Bohannon, David and Biehn, Travis, *Exploiting the java deserialization vulnerability*, Synopsis, 2020. [Online]. Available: `https://www.synopsys.com/content/dam/synopsys/sig-assets/whitepapers/exploiting-the-java-deserialization-vulnerability.pdf` (visited on 04/18/2021).

[40] OWASP Foundation, *Deserialization of untrusted data*, 2020. [Online]. Available: `https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data` (visited on 04/18/2021).

[41] *Security Tip (ST04-015) Understanding Denial-of-Service Attacks*, 2019. [Online]. Available: `https://us-cert.cisa.gov/ncas/tips/ST04-015#:~:text=A%5C%20denial%5C%2Dof%5C%2Dservice%5C%20condition,resources%5C%20and%5C%20services%5C%20are%5C%20inaccessible` (visited on 04/28/2021).

[42] *DDoS Overview and Incident Response Guide*, 2014. [Online]. Available: `https://cert.europa.eu/static/WhitePapers/CERT-EU-SWP_14_09_DDoS_final.pdf` (visited on 04/28/2021).

[43] B. B. Gupta, R. C. Joshi, and M. Misra, "Distributed denial of service prevention techniques," *arXiv preprint arXiv:1208.3557*, 2012.

[44] T. Mahjabin, Y. Xiao, G. Sun, and W. Jiang, "A survey of distributed denial-of-service attack, prevention, and mitigation techniques," *International Journal of Distributed Sensor Networks*, vol. 13, no. 12, p. 1 550 147 717 741 463, 2017.

[45] OWASP Foundation, *Denial of Service Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Denial_of_Service_Cheat_Sheet.html` (visited on 04/28/2021).

[46] J. Clarke, "Code-Level Defenses," in *SQL Injection Attacks and Defense*, Elsevier, 2009, pp. 341–376. DOI: `10.1016/b978-1-59749-424-3.00008-6`. [Online]. Available: `https://doi.org/10.1016/b978-1-59749-424-3.00008-6`.

[47] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *2012 Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, IEEE, 2012, pp. 310–313.

[48] OWASP Foundation, *Input Validation Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html (visited on 03/23/2021).

[49] C. A. Shue, A. J. Kalafut, and M. Gupta, "Exploitable Redirects on the Web: Identification, Prevalence, and Defense.," in *WOOT*, 2008.

[50] J. Wang and H. Wu, "URFDS: Systematic discovery of Unvalidated Redirects and Forwards in web applications," in *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015, pp. 697–698. DOI: 10.1109/CNS.2015.7346891.

[51] OWASP Foundation, *Unvalidated Redirects and Forwards Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html (visited on 04/29/2021).

[52] R. Shay, A. Bhargav-Spantzel, and E. Bertino, "Password policy simulation and analysis," in *Proceedings of the 2007 ACM workshop on Digital identity management*, 2007, pp. 1–10.

[53] W. C. Summers and E. Bosworth, "Password policy: the good, the bad, and the ugly," in *Proceedings of the winter international synposium on Information and communication technologies*, 2004, pp. 1–6.

[54] K.-P. L. Vu, R. W. Proctor, A. Bhargav-Spantzel, B.-L. B. Tai, J. Cook, and E. E. Schultz, "Improving password security and memorability to protect personal and organizational information," *international journal of human-computer studies*, vol. 65, no. 8, pp. 744–757, 2007.

[55] *CWE-521: Weak Password Requirements*. [Online]. Available: https://cwe.mitre.org/data/definitions/521.html (visited on 05/05/2021).

[56] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Y.-Y. Choong, K. K. Greene, and M. F. Theofanos, "Digital identity guidelines: authentication and lifecycle management," Tech. Rep., Jun. 2017. DOI: 10.6028/nist.sp.800-63b. [Online]. Available: https://doi.org/10.6028/nist.sp.800-63b.

[57] OWASP Foundation, "Application Security Verification Standard 4.0.2," 2020.

[58] ——, *Password Storage Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (visited on 04/13/2021).

[59] Sebastian Peyrott, *A Look at The Draft for JWT Best Current Practices*, Auth0. [Online]. Available: https://auth0.com/blog/a-look-at-the-latest-draft-for-jwt-bcp/ (visited on 04/11/2021).

[60] OWASP Foundation, *JSON Web Token Cheat Sheet for Java*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html` (visited on 04/11/2021).

[61] George Koniaris, *How to securely store JWT tokens.* 2020. [Online]. Available: `https://dev.to/gkoniaris/how-to-securely-store-jwt-tokens-51cf` (visited on 04/12/2021).

[62] Tom Abbott, *Where to Store your JWTs – Cookies vs HTML5 Web Storage*, 2016. [Online]. Available: `https://stormpath.com/blog/where-to-store-your-jwts-cookies-vs-html5-web-storage` (visited on 04/12/2021).

[63] OWASP Foundation, *Session Management Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html#cookies` (visited on 04/12/2021).

[64] M. Kolšek, "Session fixation vulnerability in web-based applications," *Acros Security*, vol. 7, 2002.

[65] OWASP Foundation, *Access Control Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html` (visited on 05/13/2021).

[66] G.-J. Ahn and R. Sandhu, "Role-based authorization constraints specification," *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 207–226, 2000.

[67] V. C. Hu, D. Ferraiolo, R. Kuhn, A. R. Friedman, A. J. Lang, M. M. Cogdell, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, *et al.*, "Guide to attribute based access control (ABAC) definition and considerations (draft)," *NIST special publication*, vol. 800, no. 162, 2013.

[68] *Web Security*, 2018. [Online]. Available: `https://infosec.mozilla.org/guidelines/web_security` (visited on 03/15/2021).

[69] OWASP Foundation, *Cross-Site Request Forgery Prevention Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html` (visited on 03/22/2021).

[70] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 75–88.

[71] *CWE-352: Cross-Site Request Forgery (CSRF)*. [Online]. Available: `https://cwe.mitre.org/data/definitions/352.html` (visited on 03/23/2021).

[72] OWASP Foundation, *Cryptographic Storage Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html` (visited on 04/29/2021).

[73] V. Rijmen and J. Daemen, "Advanced encryption standard," *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pp. 19–22, 2001.

[74] Celi, Chris and Jing, Janet, *Cryptographic Algorithm Validation Program*, 2021. [Online]. Available: `https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program` (visited on 04/29/2021).

[75] M. J. Dworkin, *Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques*, 2001.

[76] R. Chandramouli, M. Iorga, and S. Chokhani, "Cryptographic key management issues and challenges in cloud services," in *Secure Cloud Computing*, Springer, 2014, pp. 1–30.

[77] E. Barker, "SP 800-57 Part 1 Rev. 5. Recommendation for Key Management: Part 1 – General," Tech. Rep., 2020.

[78] A. Mourad, M.-A. Laverdiere, and M. Debbabi, "Towards an aspect oriented approach for the security hardening of code," in *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, IEEE, vol. 1, 2007, pp. 595–600.

[79] A. Avizienis, J.-C. Laprie, B. Randell, *et al.*, *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[80] OWASP Foundation, *Error Handling Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html` (visited on 05/05/2021).

[81] C.-Y. Hsieh, C. Le My, K. T. Ho, and Y. C. Cheng, "Identification and refactoring of exception handling code smells in JavaScript," *Journal of Internet Technology*, vol. 18, no. 6, pp. 1461–1471, 2017.

[82] OWASP Foundation, *OWASP Secure Headers Project*, 2021. [Online]. Available: `https://owasp.org/www-project-secure-headers/` (visited on 03/15/2021).

[83] I. Dolnák and J. Litvik, "Introduction to HTTP security headers and implementation of HTTP strict transport security (HSTS) header for HTTPS enforcing," in *2017 15th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2017, pp. 1–4. DOI: `10.1109/ICETA.2017.8102478`.

[84] A. Lavrenovs and F. J. R. Melón, "HTTP security headers analysis of top one million websites," in *2018 10th International Conference on Cyber Conflict (CyCon)*, 2018, pp. 345–370. DOI: 10.23919/CYCON.2018.8405025.

[85] M. Ying and S. Q. Li, "CSP adoption: current status and future prospects," *Security and Communication Networks*, vol. 9, no. 17, pp. 4557–4573, 2016. DOI: https://doi.org/10.1002/sec.1649. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1649. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1649.

[86] I. Dolnák, "Implementation of referrer policy in order to control HTTP Referer header privacy," in *2017 15th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2017, pp. 1–4. DOI: 10.1109/ICETA.2017.8102477.

[87] *HTST Preload List Submission*. [Online]. Available: https://hstspreload.org/ (visited on 03/15/2021).

[88] *CSP: require-trusted-types-for*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-trusted-types-for (visited on 03/15/2021).

[89] *CSP Evaluator*. [Online]. Available: https://csp-evaluator.withgoogle.com/ (visited on 03/15/2021).

[90] *Referrer-Policy*, 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referrer-Policy.

[91] *Feature: XSS Auditor (removed)*. [Online]. Available: https://www.chromestatus.com/feature/5021976655560704 (visited on 03/15/2021).

[92] *Caching in HTTP*. [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html (visited on 03/16/2021).

[93] *About CVE*, 2020. [Online]. Available: https://cve.mitre.org/about/index.html (visited on 05/06/2021).

[94] K. Kent and M. Souppaya, "Guide to computer security log management," *NIST special publication*, vol. 92, pp. 1–72, 2006.

[95] OWASP Foundation, *Logging Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html (visited on 04/24/2021).

[96] A. Chuvakin and G. Peterson, "How to do application logging right," *IEEE Security & Privacy*, vol. 8, no. 4, pp. 82–85, 2010.

[97] R. Marty, "Cloud application logging for forensics," in *proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 178–184.

[98] S. B. Chavan and B. Meshram, "Classification of web application vulnerabilities," *International Journal of Engineering Science and Innovative Technology (IJESIT)*, vol. 2, no. 2, pp. 226–234, 2013.

[99] Y. Pan, "Interactive application security testing," in *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, 2019, pp. 558–561. DOI: `10.1109/ICSGEA.2019.00131`.

[100] J. Novak, A. Krajnc, *et al.*, "Taxonomy of static code analysis tools," in *The 33rd International Convention MIPRO*, IEEE, 2010, pp. 418–422.

[101] A. Brucker and U. Sodan, "Deploying static application security testing on a large scale," *Sicherheit 2014–Sicherheit, Schutz und Zuverlässigkeit*, 2014.

[102] F. Ö. Sönmez and B. G. Kiliç, "Holistic Web Application Security Visualization for Multi-Project and Multi-Phase Dynamic Application Security Test Results," *IEEE Access*, vol. 9, pp. 25 858–25 884, 2021. DOI: `10.1109/ACCESS.2021.3057044`.

[103] OWASP Foundation, *Vulnerability Scanning Tools*. [Online]. Available: `https://owasp.org/www-community/Vulnerability_Scanning_Tools` (visited on 06/09/2021).

[104] Migues, Sammy and Steven, John and Ware, Mike, *BSIMM 11*. [Online]. Available: `https://www.bsimm.com/` (visited on 05/24/2021).

[105] OWASP Foundation, *OWASP SAMM*. [Online]. Available: `https://owaspsamm.org/` (visited on 05/24/2021).

[106] R. Lepofsky, *The manager's guide to web application security: a concise guide to the weaker side of the web*. Apress, 2014.

[107] H. Atashzar, A. Torkaman, M. Bahrololum, and M. H. Tadayon, "A survey on web application vulnerabilities and countermeasures," in *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, IEEE, 2011, pp. 647–652.

[108] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "An empirical analysis of xss sanitization in web application frameworks," Technical report, UC Berkeley, Tech. Rep., 2011.

[109] K. Peguero, "Impact of Frameworks on Security of JavaScript Applications," PhD thesis, The George Washington University, 2021.

[110] A. Gizas, S. Christodoulou, and T. Papatheodorou, "Comparative evaluation of javascript frameworks," in *Proceedings of the 21st International Conference on World Wide Web*, 2012, pp. 513–514.

[111] D. Graziotin and P. Abrahamsson, "Making Sense Out of a Jungle of JavaScript Frameworks," in *Product-Focused Software Process Improvement*, J. Heidrich, M. Oivo, A. Jedlitschka, and M. T. Baldassarre, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 334–337, ISBN: 978-3-642-39259-7.

[112] S. Delcev and D. Draskovic, "Modern JavaScript frameworks: A survey study," in *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, IEEE, 2018, pp. 106–109.

[113] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[114] J. Gläser and G. Laudel, *Experteninterviews und qualitative Inhaltsanalyse: als Instrumente rekonstruierender Untersuchungen*. Springer-Verlag, 2009.

[115] J. W. Creswell and J. Creswell, *Research design*. Sage publications Thousand Oaks, CA, 2003.

[116] *npm scores #66*. [Online]. Available: `https://github.com/npm/feedback/discussions/66` (visited on 05/07/2021).

[117] OWASP Foundation, "OWASP API Security Top 10 2019 - The Ten Most Critical API Security Risks," 2019.

[118] O. B. Al-Khurafi and M. A. Al-Ahmad, "Survey of Web Application Vulnerability Attacks," in *2015 4th International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, 2015, pp. 154–158. DOI: `10.1109/ACSAT.2015.46`.

[119] S. Tyagi and K. Kumar, "Evaluation of Static Web Vulnerability Analysis Tools," in *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2018, pp. 1–6. DOI: `10.1109/PDGC.2018.8745996`.

[120] S. Gupta and B. B. Gupta, "Detection, avoidance, and attack pattern mechanisms in modern web application vulnerabilities: present and future challenges," *International Journal of Cloud Applications and Computing (IJCAC)*, vol. 7, no. 3, pp. 1–43, 2017.

[121] OWASP Foundation, *Injection Theory*. [Online]. Available: `https://owasp.org/www-community/Injection_Theory` (visited on 03/06/2021).

[122] *Ranking of the most popular database management systems worldwide, as of December 2020*, 2020. [Online]. Available: `https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/` (visited on 03/10/2021).

[123] *DB-Engines Ranking*, 2021. [Online]. Available: `https://db-engines.com/en/ranking` (visited on 03/10/2021).

[124] *Spring Data MongoDB NoSql Injection*, 2018. [Online]. Available: `https://stackoverflow.com/questions/51632992/spring-data-mongodb-nosql-injection` (visited on 03/11/2021).

[125] [Online]. Available: `https://www.npmjs.com/package/mysql` (visited on 03/11/2021).

[126] *HTML 5*. [Online]. Available: `https://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml0` (visited on 03/09/2021).

[127] *Web technology for developers*, 2021. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML`.

[128] *Introducing JSX*. [Online]. Available: `https://reactjs.org/docs/introducing-jsx.html` (visited on 03/09/2021).

[129] *DOM Elements*. [Online]. Available: `https://reactjs.org/docs/dom-elements.html` (visited on 03/09/2021).

[130] *Security*. [Online]. Available: `https://angular.io/guide/security` (visited on 03/09/2021).

[131] *ElementRef*. [Online]. Available: `https://angular.io/api/core/ElementRef` (visited on 03/25/2021).

[132] P. DeRyck, *Preventing XSS in React (Part 2): dangerouslySetInnerHTML*, 2020. [Online]. Available: `https://pragmaticwebsecurity.com/articles/spasecurity/react-xss-part2.html` (visited on 03/09/2021).

[133] *How To Write Secure Code In React*, 2019. [Online]. Available: `https://medium.com/@rezaduty/how-to-write-secure-code-in-react-937579011d3c` (visited on 03/09/2021).

[134] OWASP Foundation, *Deserialization Cheat Sheet*. [Online]. Available: `https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html` (visited on 05/31/2021).

[135] A. Chaudhary, *Insecure Deserialization*, 2018. [Online]. Available: `https://medium.com/@chaudharyaditya/insecure-deserialization-3035c6b5766e` (visited on 04/16/2021).

[136] *HTML: The Markup Language*. [Online]. Available: `https://www.w3.org/TR/2012/WD-html-markup-20120329/input.email.html` (visited on 03/25/2021).

[137] *Validators*. [Online]. Available: `https://angular.io/api/forms/Validators` (visited on 03/28/2021).

[138] *Validation in Spring Boot*, 2021. [Online]. Available: `https://www.baeldung.com/spring-boot-bean-validation` (visited on 03/28/2021).

[139] *Class SecureRandom*. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html (visited on 03/23/2021).

[140] X. de Carné de Carnavalet and M. Mannan, "From very weak to very strong: Analyzing password-strength meters," in *Network and Distributed System Security Symposium (NDSS 2014)*, Internet Society, 2014.

[141] OWASP Foundation, *Authentication Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html (visited on 05/03/2021).

[142] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: new generation of memory-hard functions for password hashing and other applications," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2016, pp. 292–302.

[143] *11. Authorization*. [Online]. Available: https://docs.spring.io/spring-security/site/docs/5.2.x/reference/html/authorization.html (visited on 04/21/2021).

[144] [Online]. Available: https://github.com/spring-projects/spring-security/tree/master/web/src/main/java/org/springframework/security/web/csrf (visited on 03/23/2021).

[145] *Error Handling in React 16*, 2017. [Online]. Available: https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html (visited on 04/23/2021).

[146] *Deployment*. [Online]. Available: https://angular.io/guide/deployment (visited on 05/31/2021).

[147] *Error Handling*. [Online]. Available: https://expressjs.com/en/guide/error-handling.html (visited on 03/16/2021).

[148] OWASP Foundation, *Clickjacking Defense Cheat Sheet*. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking_Defense_Cheat_Sheet.html (visited on 04/22/2021).

[149] *Auditing package dependencies for security vulnerabilities*. [Online]. Available: https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities (visited on 03/12/2021).

[150] *Security advisories*. [Online]. Available: https://www.npmjs.com/advisories (visited on 03/12/2021).

[151] *How npm audit works?* 2019. [Online]. Available: https://stackoverflow.com/questions/55569305/how-npm-audit-works (visited on 03/12/2021).

[152] *Snyk Intel Vulnerability Database*. [Online]. Available: https://snyk.io/product/vulnerability-database/ (visited on 03/12/2021).

[153] [Online]. Available: https://www.npmjs.com/package/snyk (visited on 03/12/2021).

[154] [Online]. Available: https://retirejs.github.io/retire.js/ (visited on 03/14/2021).

[155] *Log Files*. [Online]. Available: https://httpd.apache.org/docs/2.4/logs.html (visited on 05/08/2021).

[156] *Source Code Security Analyzers*. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers (visited on 06/09/2021).

[157] ——, *Source Code Analysis Tools*. [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools (visited on 06/09/2021).

[158] T. Rangnau, R. v. Buijtenen, F. Fransen, and F. Turkmen, "Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines," in *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*, IEEE, 2020, pp. 145–154.

[159] *Metric Definitions*. [Online]. Available: https://docs.sonarqube.org/latest/user-guide/metric-definitions/ (visited on 05/14/2021).

[160] *CodeQL Documentation*. [Online]. Available: https://codeql.github.com/docs/ (visited on 05/15/2021).