UPPSALA
UNIVERSITET

IT 21 084
Examensarbete 30 hp
September 2021

Input Validation and Input Sanitization for Web Applications

Kevin Pedram Alemi

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Input Validation and Input Sanitization for Web Applications

Kevin Pedram Alemi

The rise of web-based applications is rapidly increasing with time and demand. As more and more web applications are being developed, so are the threats against these web applications' user-base. Input-based attacks against web applications are becoming more common. A new task is in the hands of the developers, namely input handling. The security of web applications is now more relevant than ever before.

This thesis investigates different methods used to build web applications that can prevent malicious input, specifically the infamous XSS attack. The methods covered in this thesis are using built-in implementations from frameworks for client-side development and popular developer-made libraries for server-side.
This thesis also collects real-life regular expressions from web applications used for input handling. These regular expressions have been evaluated with the Black Ostrich security scanner to determine whether a web application's given regular expression is safe enough to withstand malicious input.

The results indicate that the applications built can never fully trust the client-side for input handling for web application development. There should always be server-side handling the input. Developers should always avoid rendering user input to the DOM unless the input has passed several layers of input handling.

The thesis provides a script used to collect roughly 12000000 HTML documents, and from these documents, it gathered approximately 79000 regular expressions. These were the regular expressions that were evaluated on their safety, using the Black Ostrich security scanner. Roughly 13700 of them were deemed by the security scanner to be unsafe, indicating that nearly one-fifth of all regular expressions found from the parsed data set could be bypassed with an XSS-type string.

Handledare: Amanda Stjerna
Ämnesgranskare: Philipp Rümmer
Examinator: Mats Daniels
IT 21 084
Tryckt av: Reprocentralen ITC

Acknowledgements

I want to express my gratitude to my supervisor Amanda Stjerna and my reviewer Philipp Rümmer. I'm very grateful for being given this opportunity and being guided by both of you during this thesis. The knowledge you shared during our weekly meetings is what brought me to this point. I'm incredibly grateful for the continuous support and positive feedback I have received during these months.

I also want to thank my parents for teaching me everything that I know, constantly supporting me, and always being there through thick and thin. The life lessons you taught me have brought me this far in life, and I will forever be grateful for being your son.

"Good Thoughts, Good Words, Good Deeds"
—Asho Zartosht

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In modern-day software development, there has been a significant increase in web-based applications. There has been a change from the widespread usage of native applications to web applications [7]. A web-based application can be seen as a distributed application, whereas a native application would be a local application. The advantages of web applications are that they are easier to distribute since writing web applications for different platforms, i.e., web browsers, do not differ as much as writing applications for various native platforms, operating systems. However, with this change enters a new threat, namely the threat against the safety of the web applications. Due to web applications being distributed and public, they can quickly become a target for malicious users. This makes the security of web applications a critical aspect, they should be able to withstand incoming attacks from malicious users.

Web development is becoming one of the more popular and considerable revenue-generating industries [13, 27, 28]; with this increase in popularity, there is also an increase in consequences and risks. Web applications should aim to have no vulnerabilities at all. When constructing web applications, the developers need to predict possible attacks and thus write software that can prevent said attacks. An attack would be, e.g., a web application input field with no validating methods. If the web application does not have methods for validating the input, the application has a vulnerability, specifically for input. Attackers could exploit this vulnerability by providing malicious input, meaning that it has dangerous effects on the system. In the previous example of an input-based attack, such a scenario could lead to the application sending sensitive user data back to the attacker, thus leaking sensitive data into the hands of the wrong parties.

The more users a system has, the larger the consequences of malicious attacks are. Malicious attacks that cause large data leaks are just the tip of the iceberg that can cause massive implications for the users of a web application. According to PurpleSec [36], the consequences of malicious attacks on web applications are the most costly to fix, and on average, over 18 million websites at a given time each week are infected with malware.

Modern web development uses new frameworks and tools to combat these malicious attacks; however, the dangers and losses [36] remain. By looking at various sources [21, 39] for secure software development practices, a common denominator in all cases is that the most common type of malicious attack is caused by the lack of input validation and sanitization. The lack of input validation and sanitization can be exploited in XSS (cross-site scripting) attacks. XSS is when malicious input, in the form of a script or arbitrary program-code logic, is sent to web applications that proceed with parsing and executing the malicious input.

Having valid and well-functioning web security measurements are a must nowadays. Applications that countless users use must have sufficient methods to prevent attacks. Modern-day applications need stronger data integrity and prevention methods from leaking the private information of their users. A common acronym used in the security industry is the CIA model [41], which stands for *confidentiality*, *integrity*, and *availability*. This thesis will focus more on the confidentiality and integrity parts of web application security. Confidentiality refers to the privacy of users, i.e., the protection of user data from being leaked and thus exploited by malicious users [41]. For example, in modern-day XSS attacks, malicious users can exploit unsuspecting users to leak their data, from stored passwords to banking information, by sending them a malicious link. Integrity refers to web applications being able to prevent malicious attacks that can manipulate the applications' data [41]. A similar example as before, XSS attacks can be constructed to manipulate a web application's data since attackers can construct malicious attacks that are permanently saved on the web application.

The relatively recent **G**eneral **D**ata **P**rotection **R**egulation (GDPR) is an EU-based law [45] that protects user data and user privacy. The purpose of this regulation is to allow users to authority over their data and be able to manage it [45]. The GDPR law also enforces laws that businesses must, within 72 hours, report whether a data breach has occurred [45]; otherwise, the business can be fined. Companies should invest more resources in creating strong security measures to prevent GDPR punishments with this law in mind.

By looking at online surveys such as Stack Overflows 2020 developer survey [28], many of the respondents are currently working with full-stack and front-end development. Therefore, information and studies about correctly validating and sanitizing input are essential, especially for new developers who are starting in the field. They might not have knowledge nor experience of the importance of web application security. This inexperience could lead to security pitfalls once said new developers begin developing web applications. The full-stack development field is rapidly expanding. In their cases, many new developers come from self-taught environments, boot camps, and degrees with insufficient knowledge of cyber security, more specifically web security. With many rookies entering the field, a report on specifically XSS attacks and prevention methods will be beneficial for the field.

Since input handling must be performed both on the client-side and on the server-side, this thesis will analyze both aspects.

Besides studying input validation and input sanitization in modern-day web applications, the niche of this thesis is to collect data on commonly used regular expressions for client-side development. The regular expressions were collected from many publicly available HTML documents parsed from online web pages. These collected regular expressions will be tested on their safety against a web scanner that will attempt to break them by providing input that can bypass them. The web scanner has previously been used to try to break regular expressions collected from popular websites; this thesis does, however, provide a substantially larger data set for it to run against.

By studying input handling methods, this thesis will attempt to answer research questions such as what client-side frameworks are most commonly used when developing web applications' client-side and the most commonly used libraries for developing the server-side logic with NodeJS. The thesis will also inspect the methods these frameworks and libraries use to prevent input-based attacks and whether they are sufficient alone.

The results from collecting data on popular frameworks indicate that React is the most commonly used framework. For server-side libraries, the `validatory` library depicts the most significant results. React provides built-in sanitization techniques that can prevent malicious input to an extent but can not be trusted entirely alone. The `validatory` library offers an abundance of methods for validating and sanitizing input. The thesis concludes that using the methods client-side frameworks provide alone is not sufficient and must be further strengthened with server-side logic.

As mentioned before, the thesis collected a large set of regular expressions that are later tested against a web scanner to evaluate their safeness. These regular expressions were collected by parsing HTML pattern attributes, which can be found in HTML `input` tags; the values of the attributes are regular expressions. Roughly 12 million HTML documents were collected and parsed for this thesis experiment. From these 12 million HTML documents 79187 `input` tags with pattern attributes were found. The web scanner evaluated these collected pattern attributes values, i.e., regular expressions, and found that roughly 63k could not be bypassed by the web scanner and could thus safely prevent malicious input provided by the web scanner. However, approximately 13k of the regular expressions collected were able to be bypassed by the web scanner.

1.1 Goals

The goal of this thesis was to attempt to answer a set of different research questions. The thesis had to collect large amounts of data to back up the answer to answer the research questions. The thesis evaluated other methods used for building secure web applications that can withstand malicious input. These methods were namely input validation and input sanitization.

The thesis also set out to collect data on regular expressions that will be tested on their safeness. Since regular expressions are a method of input validation, it would demonstrate how secure real web pages' regular expressions are.

The following are the research questions that this thesis aimed to answer.

- The distribution of different frameworks used in web application development;
- The prevalence of bare-metal access methods in the framework ReactJS;
- The distribution of different NodeJS libraries used in web application development;
- The distribution of regular expressions collected that can be bypassed or not by the web scanner;

- Would client-side frameworks be enough to provide sufficient security to web applications alone?
- Would server-side libraries provide sufficient security to web applications alone?

Chapter 2

Background

This chapter of the thesis will present standard methods for preventing malicious input, the differences between client-side and server-side validation logic, and a brief explanation of regular expressions. It will also consist of a general explanation of the societal impacts of malicious attacks and different types of malicious attacks.

2.1 User Input or User Data

Since the beginning of software, a fundamental part of executing a set of instructions is to give it a form of input to view the transformed output. These inputs and software solutions vary from all kinds of different use-cases, from `N-body simulations` to simple `calculate-your-salary-after-taxes` programs.

In most modern-day user-based software, the applications need methods for handling of user-supplied input. A software developer can not cover or predict all possible input combinations supplied to its program. Instead, they must use different techniques to validate them properly. However, if the validation is not strong enough, the said input could have disastrous effects on the program. There are several different methods for preventing malicious input interpreted by the software. In the following sections, the thesis will explain the different methods.

This thesis explains two types of input, namely valid input and malicious input. When constructing web applications and general software, it is common to assume that input sent to the application follows a specific format. If input follows the expected form, it is classified as valid input, by not following the desired format and including arbitrary logic in the input classifies it as malicious.

Valid input is input sent by the user to the web application that is structured correctly and follows the input field's rules. E.g., in an input field expecting a phone number, inputs that contain characters would not be able to be interpreted. Only numeric values would. A malicious input is an input that does not follow the rules of the input field. Instead, the input contains arbitrary logic that will cause the web application to interpret and thus execute the logic. E.g., an input field with insufficient security could execute JavaScript code if it were to be sent as input.

Web applications consist of forms that users can fill out to perform different methods, anything from a search form to a registration form. When developing applications that rely on input from users to be processed, the logic needs to determine how to differentiate correct user input from malicious user input. The evaluation of user input is of utmost importance since it can lead to disastrous results if the evaluation method is insufficient and the application attempts to execute malicious input.

Applications need methods to determine whether a filled-out input field contains malicious input, i.e., arbitrary code that will be executed or if it has valid input data. This is why applications that rely on user input need methods such as input validation and input sanitization to process both valid input and malicious input. These methods of validating and manipulating input will be explained further in the upcoming sections.

2.2 Threat Models

Threat modelling [32] is a method of detecting and preventing threats that can arise when developing or building a product. The method works as such that it will detect, analyze and attempt to protect a system. The model contains information on what threats could lead to the system being attacked and harmed [32]. The model [32] usually consists of information about the system to be protected. Threats that could potentially harm the said system, measures used to prevent a said threat, and a way to confirm the system and the threats it had and to verify whether the efforts made were successful in preventing the threats [32].

In web application development, threat modeling is of great importance since it will allow the developers to create a nuanced image of potential threats. The threat model also consists of [32] methods to prevent these threats, descriptions of threats, validation methods for the threats, etc.

Threat modeling should be used since developers will visualize what can go wrong from a security point of view in an early stage. The OWASP motto for threat modelling [32] is *Threat modeling: the sooner, the better, but never too late* since the method of threat modeling does not necessarily need to be before developing an application but rather as quickly as possible. For example, when adding new features to an existing application, the developers should create a threat model since these new features could potentially lead to the application being prone to malicious attacks. Threat modeling will allow the developers to create prevention methods for the threats that they assume can occur. New threats can always arise, which makes re-creating threat models in sprints a good choice since the application will consistently be up to date.

Having a threat model early on allows developers to prevent potential attacks. By using threat models, a web application will fulfill the CIA model, specifically confidentiality. An example will be if an attacker performs an arbitrary XSS attack against a web application to steal the browser session's cookies. If this hypothetical web application was developed using no threat models in mind, the attack would presumably succeed and thus leak private data. This would be a strike against the confidentiality of the web application. Similarly, without threat modeling, attacks can also affect the integrity and availability of a web application. Where XSS attacks can manipulate data (integrity), and denial-of-service attacks can ruin the availability of the data by attacking the web application.

2.3 Input Validation

Input validation [34] is a method of ensuring that the input supplied to an arbitrary program follows the rules of how input may be constructed. For example, a program expecting an input of type `String` should not allow input of type `Integer`. According to OWASP [34], the validation of input should occur as early as possible to prevent malicious input successfully. Validating input should be checked on two different levels [34], namely syntax-based and semantic-based validation methods.

For a syntactical validation the method should, as the name suggests, check that the supplied input follows the syntactical rules that enforce how input is formed. For example, input 42 is not a valid email input.

The method should check that the supplied input values are in the correct range of what the program expects for the semantic validation. For example, that the email `hello-world@bork.com` is not a valid Uppsala University email.

There are several different ways of implementing input validation methods, such as block-lists and regular expressions. The concept of using block-lists [34] works as such that the system has two different lists: an allow-list and a block-list. The allow-list contains input that fulfills the validation rules, while the block-list includes all the input that does not exist in the allow-list. This allows the program to validate the supplied input against an allow-list to see if it fulfills the rules. Regular expressions will be explained later in this chapter.

2.4 Validation API

The validation API section refers to the HTML tag's built-in methods for handling input. Usually, when user-supplied input is written into a form that later sends the data, once submitted. In HTML5, when building forms, the developer can specify what type of input data it will expect [18]. If the user supplies input that does not follow the type specification, it will display an error. There is also support for the pattern attribute [17]. This attribute specifies a regular expression that the input must fulfill. The usage of these different validation methods that already occur in the client-side of the application is good to implement to give direct feedback to the user. However, it is not reliable against malicious attacks since attackers can bypass them, explained in the upcoming chapter.

2.5 Input Sanitization

Input sanitization [26] works as such that given input, it will modify it to a valid form, and thus preventing malicious input from being interpreted. Sanitizing an input removes malicious input by replacing, removing, or escaping characters [31].

Escaping characters is a method of sanitizing, i.e., manipulating malicious input to a non-malicious form. The character < is an example of a malicious character. This character would, after being escaped, be interpreted by the system as the representation of `&lt`, which means that the system interprets it as a *less than* sign, not as the beginning of an HTML tag.

One example of data sanitization in practice is the usage of JSX to prevent malicious input. JSX is a method of rendering HTML or other valid JavaScript expressions using the framework ReactJS. When using JSX [38], the React DOM will escape the content. This prevents malicious input from being parsed and incorrectly interpreted by the targeted web application. Instead, the application will just render the content as a string.

A combination of input validation and input sanitization is the best way to create programs that can successfully prevent malicious input. Combining the methods is what gives a web application more layers of security. However, both methods could be used alone by the web application. Validation would reject all invalid input, whereas sanitization would manipulate all invalid input. It is two different approaches to solve the same problem.

2.6 Regular Expressions

Regular expressions are patterns that are used to match an arbitrary string [3, 19]. Regular expressions define what kind of forms a string might take. They are used for input validation since they specify how the input should be formed (syntactical validation). Regular expressions can be used to describe infinite sets of input, as long as they abide by the syntactical rules.

An example for a simple regular expression that represents an email could be `([a-zA-Z0-9])+@([a-zA-Z])+.domain-name`. In the given example the pattern `[a-zA-Z0-9]+` represents that the string must consist of 1 or more lowercase letters, uppercase letters or digits, followed by an `@` sign, the email service provider, and lastly domain name.

By using regular expressions correctly, the input validation methods for a given program will cover infinite sets of input and still apply the rules intended by the developer.

2.7 Architecture of Web Applications

In this section, the thesis will explain where input validation and input sanitization methods may occur in the system architecture. Web applications constructed today consist of a visual entity, i.e., front end or client, that communicates with a business logic, i.e., back end or server, entity.

The web application is distributed since its overall functionality can be accomplished by creating multiple nodes, i.e., entities, that communicate together.
The web application's client-side consists of the visual layer, which is presented to the user. It also contains the logic of sending user-based operations to the server, which leads to updates of the visual layer. The server-side entity is where all user operations are interpreted. All business logic is implemented here.

Web applications should aim to have sufficient input handling. By using both input validation and input sanitization, a web application creates more layers of security. These methods of input handling can be performed on either the client-side or the server-side. The following image is a visual description of how a web application's architecture may look and components input handling may occur.

As seen in the image 2.1, input handling can occur in either both entities or in one of them. Validation on the client side is good since it will give the user direct and easily

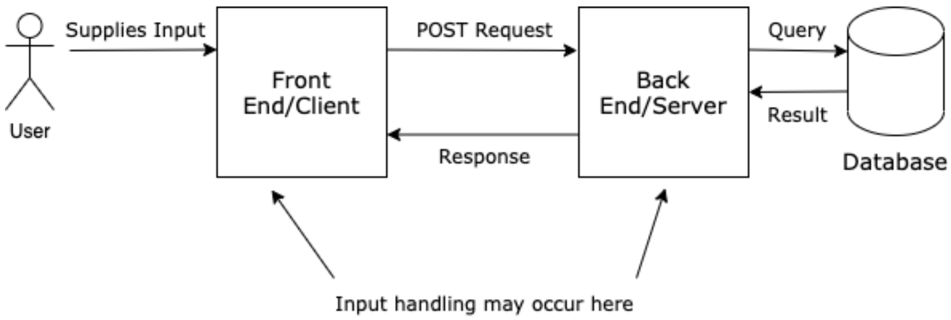

Figure 2.1: System overview of a client and server application

interpreted feedback that their input is incorrect. However, the client-side entity is the weak link of the architecture since attackers can bypass it. The client-side entity, i.e., the users' web browser, is not trusted since the user or attacker controls it. The client-side entity is thus able to be exploited, causing its input handling methods to be bypassed. Therefore the server-side entity is a must when protecting against input since the attacker can force the client-side entity to interpret malicious input. This extra layer, being the server-side, is a must since it will thoroughly check the input supplied and catch mistakes made by the client-side.

The choice of using either only client-side or server-side input handling or combining both will be discussed later in this thesis, backed up with practical examples that showcase the different choices.

2.7.1 Web Application Firewall

A web application firewall (WAF) is a firewall whose purpose is to protect a web application. The firewall can apply rules on HTTP requests being made to the web application [6, 35]. With these rules being applied to all incoming HTTP requests, the WAF can protect and prevent common attacks, such as the XSS attack [35].

The following image is a general visualization of how a WAF works.

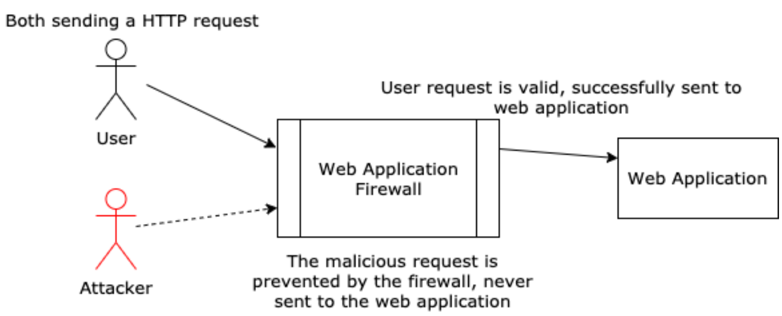

Figure 2.2: The WAF will prevent malicious requests from ever reaching the web application.

The WAF is also able to implement new rules and measures to prevent malicious requests. By using a WAF, developers can build a strong fortress of security around their web applications. Combining a WAF with input handling methods will even strengthen the application more. Examples of different WAFs are Cloudflare, Akamai, and AWS. The WAF is related to the input handling methods discussed in this thesis. One example of how it works is that it uses block-list and allow-list type solutions, meaning that it will allow input that fulfills its rules and block those that do not.

This method of applying a firewall to protect the application further can be related to the original military concept of defense in depth [43]. Defense in depth is a method of applying several layers of protection around the object to protect. From cyber security, specifically web security point-of-view, this would be combining several different layers of web security methods to protect the application [43]. For web applications, examples of several layers of protection would be threat modeling, using a WAF to prevent malicious requests, and combining those with both input validation and input sanitization on both client-side and server-side. There could potentially be more layers to protect the web application, but these are the protection methods covered in this thesis.

2.8 Client-side Logic, JavaScript and Frameworks

The client-side of a web application usually consists of 3 different building blocks, namely HTML, CSS, and JavaScript. HTML specifies how the website is built and its structure. CSS specifies the visual aspects of the website, the layout as well as colors. JavaScript is what makes the website dynamic and controls behavior, i.e., the website's client-based logic. This section, and thesis, will primarily focus on the JavaScript part of web applications client-side.

JavaScript is a programming language that was initially built to make websites interactive and create dynamic effects [20]. The language has, during later years, received a lot of attention and has branched out to different versions, e.g., NodeJS, which is a runtime environment that can execute JavaScript outside of the browser and also to a new programming language, namely TypeScript [20].

For the client-side, as seen in image 2.1, the entity's purpose is to send an arbitrary request to the server based on eligible user events. It will then receive a response and update the visual layer accordingly. In this part of the architecture, the malicious code is attempted to execute on the web application being targeted. Input validation is what can prevent the execution of malicious code.

Frameworks are tools created to introduce more lightweight and easily written web applications. JavaScript frameworks are software written to reduce the overhead of regular JavaScript web development and introduce libraries and methods that improve and simplify development. It can be seen as a large quality of life improvement for web development. A number of the larger JavaScript frameworks today [29] are jQuery [11], ReactJS [15], AngularJS [12], and VueJS [46].

2.9 Categories of Cross-Site Scripting Attacks

Many different dangers can occur when a system is attacked by malicious users, ranging from complete system shutdown to leaking sensitive data. There have been breaches, during only the first half of 2020, that leaked 36 billion records [40] of sensitive data from different public and private sectors. In 2020, the world also saw the first-ever case of an attack that led to a loss of life [42]. The attacks range from malware to phishing. However, it is not clear how many of these attacks were Cross-Site Scripting (XSS-based) attacks. These reasons should be enough for the tech industry to develop better and more robust defenses against malicious attacks.

In the following sections, the thesis will explain different types of attacks, specifically injection attacks. The XSS attack is an injection attack since the attacker will supply input to the system. Injection attacks are relevant for this thesis since they can be prevented by implementing solid and correct input handling methods.

This thesis will not explain malicious attacks such as Denial-of-service attacks, phishing attacks, and social engineering attacks.

2.9.1 Cross-Site Scripting

XSS attacks [10], short for cross-site scripting, is an injection attack that relies on malicious code being submitted to a website [33]. The attack takes its full effect once an unsuspecting user browses the web application, executing the injected malicious code. XSS attacks can take full effect when web applications have insufficient input handling methods [1]. Once the malicious code has been input and then executed, the attacker can potentially gain access to other user's data, send arbitrary HTTP requests. In the cases of web applications using HTML5, the attacker could access the user's location, webcam, mic, etc [1]. The application will struggle with determining whether the given input is code or data.

When web applications expect input from users, logic must exist that can differentiate which input is valid user data and which input is malicious data, i.e., arbitrary code. By having no methods of handling input, a web application could hypothetically interpret anything sent to it, meaning that it can not differentiate between malicious code and user data. There are also cases of vulnerable web applications executing data as code instead, which could also occur when there are no input handling methods. E.g., valid user input could consist of symbols or keywords that are valid code and thus cause the application to execute what would be normal user input.

By exploiting the lack of boundaries between code and data, the malicious input, in this case being arbitrary JavaScript code, can be executed on the targeted web application.

By having a web application that does not correctly validate and sanitize input from other users, web applications risk their users' sensitive data. In a report made by PreciseSecurity in 2019 [14], it was presented that approximately 40% of all cyber attacks during 2019 were XSS attacks. By investing time and resources in creating sufficient input handling, web applications can prevent and avoid leaking of sensitive data of their users and prevent them from performing dangerous HTTP requests without their knowledge.

2.9.2 Persistent XSS attack

A persistent XSS attack, also known as stored XSS, performs its malicious process by having an attacker send malicious JavaScript to the web application, which will store the malicious JavaScript code in the web application [2]. If the web application does not have any input validation, this maliciously injected code will be stored in the web application. Assuming that no threat modeling has been done, the web application is prone to be weak against potential attacks.

The lack of protection leads to danger for other users of the web application, since if they browse the web application that has been injected with malicious code, then the injected code can execute on their browsers. This attack imposes threats to the confidentiality and integrity of the application. The injection is performed due to a lack of input validation, resulting from insufficient threat modeling. As the injection is performed and data is either leaked, manipulated, or exploited, the web application is betraying the users' confidentiality and integrity.

An example that will be furthered visualized with image 2.3 would be that an attacker injects an arbitrary web application with malicious JavaScript code. The purpose of the code is to send the user's cookies (sensitive data) to a server that the attacker has control of. Since the code will execute on the user's browser, it can potentially send any arbitrary HTTP request to the attacker's server. The injection performed will blur the web application's boundary between what is code and actual data being sent as valid input. The application will struggle to determine how the malicious input should be interpreted and execute arbitrary malicious code since it struggles with interpretation.

The sensitive data being sent in the following images may be session cookies, i.e., they contain the user's login information.

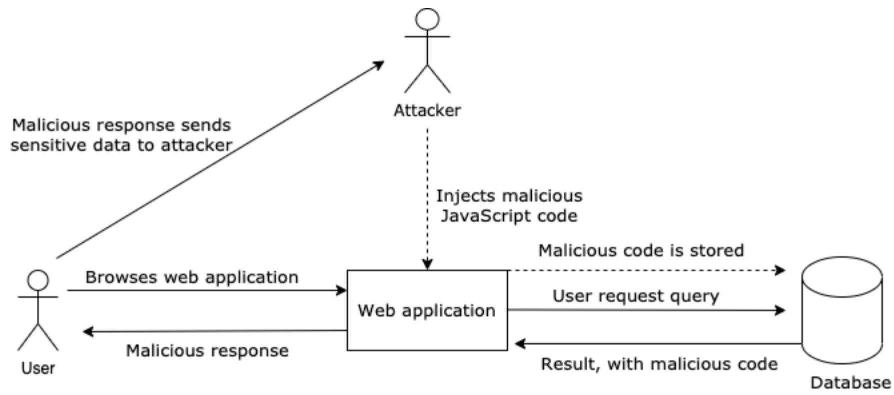

Figure 2.3: Visualization of how a persistent XSS attack works.

This type of XSS attack is the most dangerous type and can cause most damage to unsuspecting users.

2.9.3 Reflected XSS attack

A reflected XSS attack performs its malicious process by having an attacker sending a malicious link to trick the user into clicking it. The link will cause the user to send a request to a web application that will allow maliciously injected code to execute. The link is constructed so that the malicious code is often sent as a search query to the web application, and since the web application has insufficient input handling, it will execute that code. The user's browser session will interpret this as valid code and fully allow the malicious injection to effect. This leads to the injection's success since it can execute any arbitrary JavaScript instructions from the user's browser. What is important to notice for this type of XSS attack is that the attacker's malicious input is in the request sent by the user to the web application.

To clarify further, the significant difference between this reflected XSS attack and the previously mentioned persistent XSS attack is that the malicious input will be persistent. A persistent attack saves the malicious input on the web application, causing users that browse the web application to be exposed to the attack even though it has already been injected before. However, the reflected attack will consist of input that is not persistent, i.e., it is not saved on the web application. The malicious input here will not be exposed to all the users but rather to the user that clicked the maliciously manipulated link.

An example of a reflected XSS attack is demonstrated in image 2.4. In this visualization, an attacker sends a malicious link to an unsuspecting user. Once the user clicks the link, the user will send a request to the web application. The web application will execute the code in the user's browser session since it interprets it as a valid link. Since executed in the user's browser, the malicious code will successfully send sensitive data to the attacker.

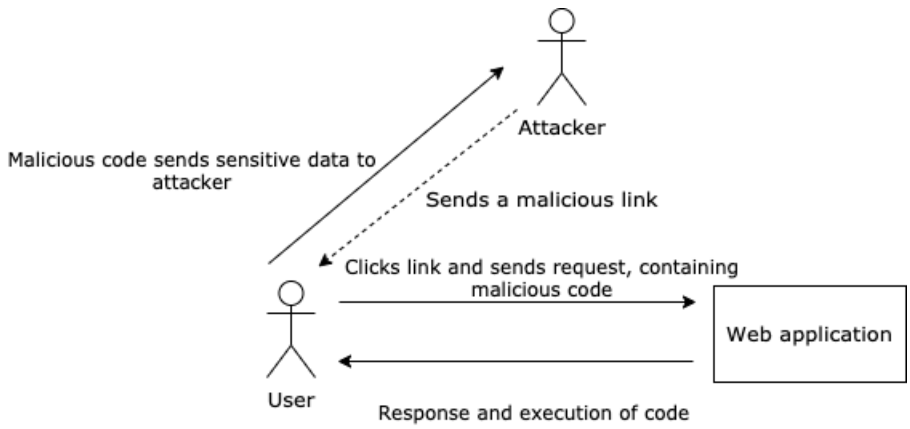

Figure 2.4: Visualization of how a reflected XSS attack works.

2.9.4 DOM-based XSS attack

To begin with, DOM stands for Document Object Model, which is the interface that represents an HTML document. The DOM is constructed by several nodes that represent a subset of the document. All these nodes together build a tree-like structure [44].

A DOM-based XSS attack performs its malicious process by directly writing malicious data to the DOM [1]. If the web application for the DOM can be dynamically manipulated by user input and the validation of that input is insufficient, then this attack can take its full effect. By having the DOM interpret malicious code, once the unsuspecting user's browser requests the DOM, it will execute any arbitrary JavaScript code used for malicious reasons. This type of attack is only executed in the client-side part of a web application since it will directly access and manipulate the DOM.

Examples could be web applications that directly append user input to their DOM, making the web application dynamic. If the user input is not properly handled, then it could be used for malicious intent.

An example of a DOM XSS attack is demonstrated similarly as in the image 2.4 from the previous section. The difference is that an attacker sends a malicious link that consists of DOM manipulating input to an unsuspecting user. As the user clicks the link, the client-side DOM will interpret this as valid input and execute it. This

JavaScript code that executes in the DOM of the web application will cause the user to send its sensitive data to the attacker.

2.10 Common Crawl

Common crawl [8] is an online data set that consists of petabytes of data that has been collected when crawling web pages. The data set is public and can be accessed by anyone for free. The data set available via Common crawl will be later used in this thesis since it contains raw HTML from billions of web applications. From this raw HTML, the previously mentioned pattern attribute can be collected, which is the purpose of utilizing the data set. Analyzing web application regular expressions might give an insight into how web applications construct their client-side input validation.

2.11 Methods to Prevent Threats against Web Applications

In the following chapter, this thesis will attempt to implement examples of input-based attacks and also input handling methods used to prevent these said attacks. The different examples demonstrated in the chapter will be primarily implemented using ReactJS for client-side demonstration and NodeJS for server-side demonstration.

ReactJS is a web development framework used to build the client-side of web applications with clean and efficient built-in methods. ReactJS uses JSX for rendering content dynamically to the visual layer of the application. ReactJS has its built-in JSX method for input handling, which will be used to demonstrate client-side prevention methods against input-based attacks.

NodeJS is a runtime environment that executes JavaScript outside of the browser, i.e., it is JavaScript code that can be used for purposes other than client-side web development. NodeJS can be used for server-side development, which is why it will be chosen to visualize and demonstrate prevention methods against input-based attacks in the upcoming chapter.

The upcoming chapter will also consist of shorter examples using plain JavaScript, which the previously mentioned tools are built upon, and a simple jQuery example.

Chapter 3

Common Threats to Web Applications & Prevention Methods

In this chapter, the thesis will attempt to re-create malicious attacks and prevent them by using input handling. The malicious attacks will be basic and mostly to visualize for the reader how such an attack is constructed and the impact it will have on a web application. Each visualized malicious attack will have a corresponding section that implements an input handling method to prevent the previously successful attack.

Lastly, this chapter will contain an example of both a client-side and a server-side input handling architecture. This example will visualize how both implementations work against malicious attacks to further prove to the reader the importance of using server-side input handling.

3.1 Construction of XSS Attacks

As previously mentioned in chapter 2 there are several ways to construct XSS attacks, be it persistent, reflected, or DOM-based. In this section, the thesis will attempt to construct XSS attacks by creating links and inputs that can cause web applications with insufficient input handling to become victims.

3.2 Mockup using Vanilla JavaScript and JQuery

The following section will show two different examples demonstrating how an XSS attack may occur and the results of falling victim to it.

3.2.1 Vanilla JavaScript

For this mockup, the web application being attacked is expecting input from the user in a text field, which will later be added to the web application's HTML. However, since there is no validation method on the input, the web application will append this input to its page raw. The JavaScript function is a simple one that adds the user input to the HTML after pressing the submit button.

The following image will display how the web application is used normally. It allows users to type in their input, which will be rendered to the DOM after submitting the form. This same example web application will be later used to demonstrate XSS attacks.

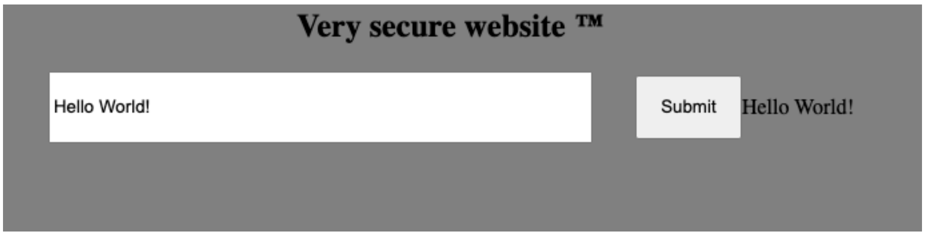

Figure 3.1: Normal usage of the web application.

The following code snippet will display how the web applications HTML and JavaScript are built, allowing XSS attacks to be made.

Listing 3.1: The web applications HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Test</title>
    <script src="xss.js" defer></script>
</head>
<body id="body">
    <h2>Very secure website TM</h2>
    <section id="f">
        <input type="text" id="inputfield">
        <button onclick="veryBadInputParser();">Submit</button>
        <p id="userinputparsed"></p>
    </section>
</body>
```

Listing 3.2: The JavaScript of the web application.

```
function veryBadInputParser() {
    let txt = document.getElementById("inputfield").value;
    userinputparsed.innerHTML += txt;
}
```

The same example as above will now be re-used to demonstrate how simple XSS attacks can be created to attack the example web application. This will be displayed in the following image, which demonstrates an XSS attack based on user input. This user input was not handled by any input handling methods and thus caused an unexpected change to the web application.

Figure 3.2: Sample input to a web application

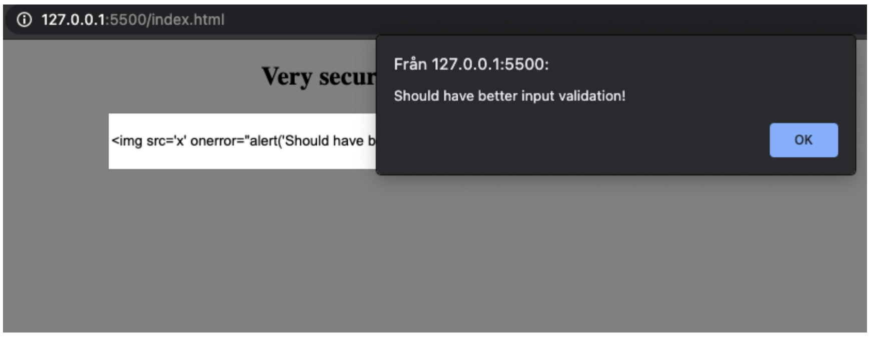

Figure 3.3: The result of evaluating malicious input without validation methods

While the example images above do not visualize data leakage caused by XSS attacks, it shows that insufficient input handling when developing web applications in vanilla JavaScript can lead to the web application being vulnerable. In the example above the following input is given to the input field
`<img src=’x’`
`onerror="alert(’Should have had better input validation!’)">`.
This input causes the web application to call a JavaScript alert function, which in this example is not that dangerous since it only prints a message. Still, in reality, it can be exploited for malicious intent.

One example of a dangerous input that works is the following
`<img src=x onerror=this.src=`
`’http://myevilserverthatstealscookies/?c=’+document.cookie>`.
This injectable code will send the users’ cookies to another web application. In this example, the web application to which it sends the cookies is a malicious one created for this intent.

Web applications with insufficient input handling are prone to be exploited because data leaks can occur, e.g., by sending sensitive user data. It is not similar to other cyberattacks that could execute programs on the attacked computers since XSS attacks are restricted to the web browser.

3.2.2 jQuery

For jQuery, an example of bad practice would be a similar method of receiving user input that has not been validated and adding it to the web application’s HTML.

Listing 3.3: The contents of the array may not have been correctly validated, it can contain malicious content

```
let listHtml = "<ul>";
for (var i = 0; i < arbitraryArray.length; i++){
    listHtml += "<li>" + arbitraryArray[i] + "</li>";
}
listHtml += "</ul";
$(document.body).append(listHtml);
```

As seen in the code snippet 3.3, a for loop iterates over the contents of an array and creates list tags for each of the array elements. The jQuery $ sign in this context is what finds the document object and its nodes, being the body. The array, however, can be an array filled with elements that are the results of several different input forms. E.g., a user fills a form containing name, address, zip-code, etc., all these input fields are appended to the array.

Without sufficient input handling, any or each of the array elements could potentially be a malicious input. E.g., the array elements could be XSS attacks that send arbitrary HTTP requests, which leak the user’s sensitive data to any arbitrary server.

3.2.3 Implementing Input Sanitization for Vanilla JavaScript and jQuery

A method of sanitizing the input received in the above examples on the client-side could be to implement a sanitization function that always executes before appending new user-supplied data to the web application. Such a function could be constructed using the JavaScript `replace()` function, which is based on a regular expression that replaces matching strings with a given value.

Listing 3.4: Function that replaces the dangerous characters in a user supplied input.

```

function removeMaliciousInput(userInput){
    return userInput.replace(/</g, "&lt;")
                    .replace(/>/g, "&gt;")
                    .replace(/’/g, "2#39;")
                    .replace(/"/g, "&quot;")
                    .replace(/&/g, "&amp;");
}
```

The function described in 3.4 demonstrates an implementation of how client-side logic can replace typical dangerous characters that malicious attack inputs consist of. Applying this function to the user input before integrating it into the HTML of the web application can help with dangerous input. However, it can still be bypassed since the validation only occurs on the client-side.

However, using such a simple implementation of a sanitization function is not recommended. In this case, using official and established sanitization libraries would be a much better choice. Implementing such functions yourself could lead to downfalls, e.g., by not managing all different types of malicious input. For this specific example, it should also be noted that using HTML tags that are not malicious, e.g., the usage of `<b>` tag which makes text bold, will also be sanitized.

3.3 Mockup using React

The first example will showcase a mockup of an XSS attack on a web application that uses React on its client side. This example will not have any server-side code. It will simply contain the default template that a newly created React project contains.

3.3.1 Using JavaScript XML (JSX)

For this mockup, the web application that will be attacked will be using the regular method of rendering data to the DOM. It will be using curly braces, which is a way of interpreting JavaScript. For example, if a web application wanted to render a JavaScript variable named `userInput` it would have used `{userInput}` to render, informing the JSX parser to read the variable as valid JavaScript.

As previously mentioned, when using Reacts built-in methods for rendering content, it will automatically escape illegal characters such as < which are usually found at the beginning of an HTML script tag used for XSS attacks. The following example will demonstrate how malicious input will be rendered in a web application using Reacts methods for rendering. Note that the input used will be the same as the input used in the Vanilla JavaScript section that caused an alert error to be performed once it was submitted.

The following code snippet displays how the web form has been implemented using ReactJS and its built-in sanitization techniques against XSS attacks.

The web application using React has been implemented using functional components. As before, a user can submit an input written in their input field. This submitted input will then be rendered to the DOM.

The following images will visualize how the web applications handles malicious input sent to it.

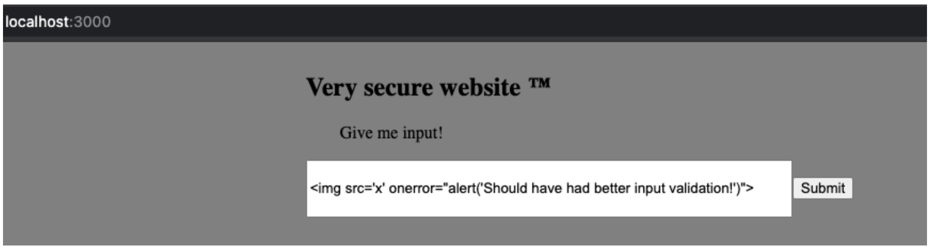

Figure 3.4: Sample input to a React based web application

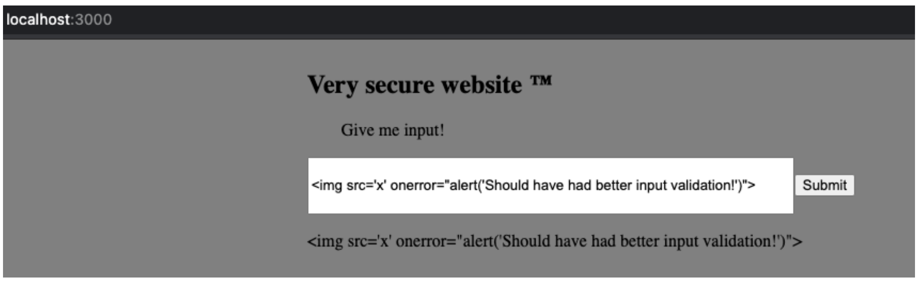

Figure 3.5: Result of evaluating malicious input using Reacts built-in rendering method

Listing 3.5: Implementation of the web application form in React.

```
import {React, useState} from ’react’;
import ReactDOM from ’react-dom’;
function RegularValidation(){

  const [userInput, setUserInput] = useState(’’);
  const [isSubmit, setIsSubmit] = useState(false);

  const onSubmit = (e) => {
    e.preventDefault();
    setIsSubmit(!isSubmit);
  }

  return (
    <div>
      <h2>Very secure website TM</h2>
      <label>Give me input!</label>
      <form>

        <input type="text" value={userInput} onInput={e => setUserInput(e.target.value)}></input>
        <button onClick={onSubmit}>Submit</button>
      </form>
      <p>
        {isSubmit && userInput}
      </p>
    </div>
  )
}

ReactDOM.render(
  <RegularValidation />,
  document.getElementById(’root’)
);
```

As seen in the examples above, the web page will not execute the malicious code but render it to the DOM. This prevents the method of sending basic XSS script tags to web applications using Reacts’ built-in rendering. The input used was the same as in image 3.2. In that example, the input was able to cause an alert window, which shows a great difference in input handling when using React instead of vanilla JavaScript.

The input handling used for this example is using a built-in method available when using React for developing the client-side. Compare this to the example in the code snippet 3.4, which is an implemented function created to sanitize malicious strings, an essential function for preventing malicious input in vanilla JavaScript and jQuery web applications. However, for the React example, no such function is needed.

The React built-in sanitization will also sanitize harmless tags such as the previously mentioned <b> tag, which makes text bold. This is since the built-in method escapes all values that are rendered to the DOM with JSX.

3.3.2 Using dangerouslySetInnerHTML

For this mockup, a web application similar to the previous one is created. However, instead of using JSX rendering methods, it will make use of the `dangerouslySetInnerHTML` element to demonstrate what effects it may have on the web application. This method is used quite a lot even though it has been said to be bad practice. As seen from the data collected in chapter 5, approximately 12% of the collected React Github repositories use the `dangerouslySetInnerHTML` element. Its name and existence are made to remind the developer that using such a method of directly inserting HTML in the DOM is bad practice [37].

For this example similar type of input as before is used, namely the malicious input `<img src=’x’ onerror="alert(’Should have had better input handling!’)">`. This will cause an alert window to pop up if there is no proper input handling of any kind.

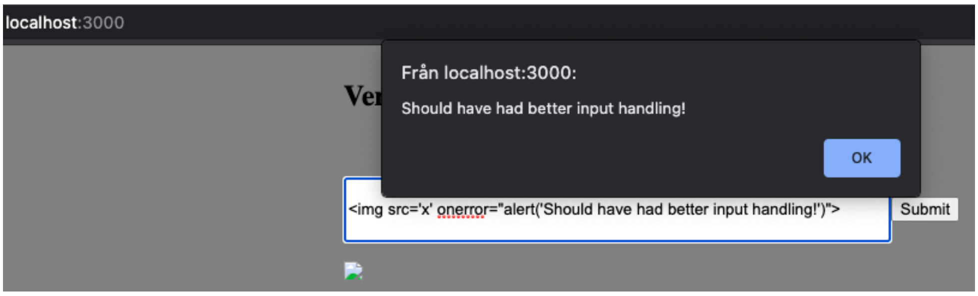

Figure 3.6: Inserting malicious input to React application that uses dangerouslySetInnerHTML

For the example demonstrated in this section, the following code snippet was used. Similar to the previous ReactJS example, the difference is that this code uses the dangerouslySetInnerHTML tag.

Listing 3.6: Implementation of the web application form in React using dangerouslySetInnerHTML.

```
import {React, useState} from 'react';
import ReactDOM from 'react-dom';

function DangerousExample(){

  const [userInput, setUserInput] = useState('');
  const [isSubmit, setIsSubmit] = useState(false);

  const onSubmit = (e) => {
    e.preventDefault();
    setIsSubmit(!isSubmit);
  }
  return (
    <div>
      <h2>Very secure website TM</h2>
      <label>Give me input!</label>
      <form>

        <input type="text" value={userInput} onInput={e => setUserInput(e.target.value)}></input>
        <button onClick={onSubmit}>Submit</button>
      </form>
      <p dangerouslySetInnerHTML={{__html: userInput}}></p>
    </div>
  )
}

ReactDOM.render(
  <DangerousExample/>,
  document.getElementById('root')
);
```

3.4 Mockup using NodeJS, server side

Due to several different libraries found for server-side input handling, each using similar methods and yielding similar results there needed to be a delimitation since the mockups would be nearly identical. Therefore, the library chosen is Validator since it contains methods for both input validation and input sanitization.

3.4.1 Validator Library, Input Validation

We have similarly created an HTML web page for the input validation method that contains an input field. This input field can be filled out from the user end and will be sent to a NodeJS server to be processed and evaluated. For the first example, the NodeJS server will be using the Validator library to validate input and perform arbitrary logic based on how the validation results.

Similar input as before,
<img src='x' onerror=
"alert('Should have had better input handling!')"> was used.
For input validation, the server-side Node logic used the validator libraries built-in isAlpha(arg: input) method to validate the input format. The method checks if

the input sent, only contains letters, ignoring upper or lower case.

The results of not using this validation method or any validation method are exactly as before. Since the logic will directly render the user input to the web application, the results will be as in figure 3.6.

However, since this example uses the `Validator` libraries built-in methods, the following results are achieved.

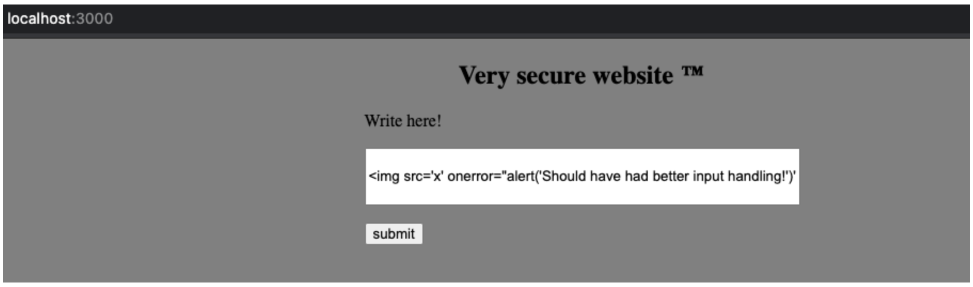

Figure 3.7: The XSS input sent to the NodeJS server

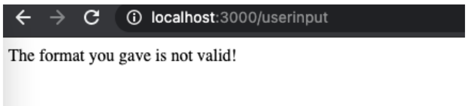

Figure 3.8: Validator library handling the POST request that contains the malicious input

As seen in the images above, the malicious input is sent to a Node server and parsed. Once parsed, since the server is using a validation method and is expecting specific input that only contains letters, the server will thus inform the user that the input is not allowed and will be prevented.

Listing 3.7: Function that handles the POST request and validates the input

```
app.post("/userinput", urlencodedParser, (req, res) => {
    if (validator.isAlpha(req.body.userInputField)){
        res.send(req.body.userInputField);
    }
    else {
        res.send("The format you gave is not valid!");
    }
});
```

The code-snippet 3.7 demonstrates the route and its function created to evaluate and handle `POST` requests made when submitting input. In this demonstration, the built-in method isAlpha is used. This method has a strong restriction that the input may only be letters. Even white spaces are rejected. The `validator` library has a large set of different built-in methods that can be used to apply restrictions to user input. By combining these methods or using them individually, the server-side logic will be correctly implementing from an input validation point of view.

3.4.2 Validator Library, Input Sanitization

The Validator library also contains a set of built-in methods that can be used to sanitize input from users, further strengthening its preventative abilities against malicious input.

The same HTML web page as before will be used. However, this time it will be running a sanitization method on the server-side instead of a validation method. Similar input as before is used
<img src='x' onerror=
"alert('Should have had better input handling!')">. For the input sanitization, the server-side method used the validator libraries built-in
escape(arg: input) method to sanitize the input text sent to the server. The methods purpose is to replace commonly used character for XSS attacks with their corresponding HTML entities, i.e it replaces < with &lt.

The results of not using this method are as expected, causes the web application to be exploited by XSS attacks since it will render the user input to the page. The results will be the same as in the image 3.6 from the previous section.

The following code snippet visualizes the implementation of using the Validator libraries sanitization method to sanitize XSS content.

Listing 3.8: Function that sanitizing the input incoming from the POST request made.

```
app.post("/userinput", urlencodedParser, (req, res) => {
    res.send(validator.escape(req.body.inputfieldname));
    // Visualizes HTML entities
    res.send(validator.escape(encode.encode(req.body.inputfieldname)));
});
```

By using the above code, utilizing the sanitization technique offered by the validator library we obtain the following results.

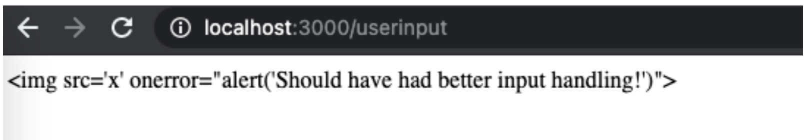

Figure 3.9: Using the escape method for parsing user input

As expected, the results are similar to before. However, to clarify even further what this method does, the experiment will be executed one more time. However, this time using the html-entities NPM package to visualize what the sanitize method does.

By using the html-entities, the XSS input will be converted to its actual form. The following is what is being rendered and parsed by the web application when executing the escape method.

As seen in the image 3.10, the actual results that the escape method returns are the entities of the reserved characters. The reserved characters being those commonly used in XSS attacks. The escape method will replace bad characters with their HTML entity, making them safe to render to the DOM.

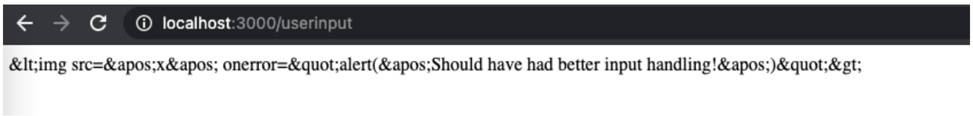

Figure 3.10: The actual representation of the sanitized XSS input

3.5 Mockup using both client and server

This section is worth noticing that there are many different approaches when implementing both client-side and server-side input handling. Hypothesize that a web application has both client-side and server-side input handler methods. Specifically, for this example, the client-side input handling method will be using the HTML pattern attribute, which only allows inputs that fulfill the regular expression given. The server-side will use the `validator` library methods once again.

By simply trying to send malicious input in the client-side interface, the pattern will catch and prevent the form from being submitted. The web application will begin using the pattern attribute in the following image, which only allows letters and digits. The regular expression used is `[A-Za-z0-9]+`.

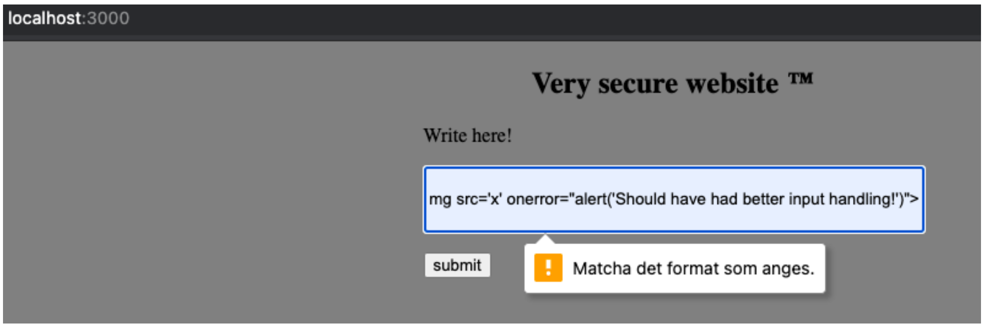

Figure 3.11: Using the pattern attributes, this commonly used XSS input can be prevented

Only using the pattern attribute is that the web application can successfully prevent this input without using the server-side logic.

However, malicious attackers may bypass this method of input handling. Since the logic is only client-side, attackers may use other approaches. There are several ways to mitigate this method of handling input, for example, disabling client-side logic, i.e., JavaScript, but the HTML pattern will still cover malicious input. But, the attacker can bypass it by using CURL requests to send the malicious data. The choice of using CURL requests is just a matter of taste. This experiment shows that interaction between a client and a server does not only have to occur when using a web browser, but you could also send crafted requests.

For the sake of demonstrating, the same client-side logic as before will be used, and the server-side logic will be removed to give a preview of how only relying on the client-side is wrong.

This example will demonstrate what happens in the web application when different types of input are sent to it. It will begin with sending valid input and then malicious input to see which are parsed and evaluated as valid input.

As previously mentioned, attackers may bypass the usage of the HTML pattern by sending CURL requests instead of manually typing the XSS attack into the user interface. This will be implemented by using the Python `requests` library to send a POST request to the URL containing malicious data as input. Each time the web application receives data as input, be it either via the user interface or via CURL requests from an external Python script, it will print each input that has been parsed and thereby be interpreted as valid input.

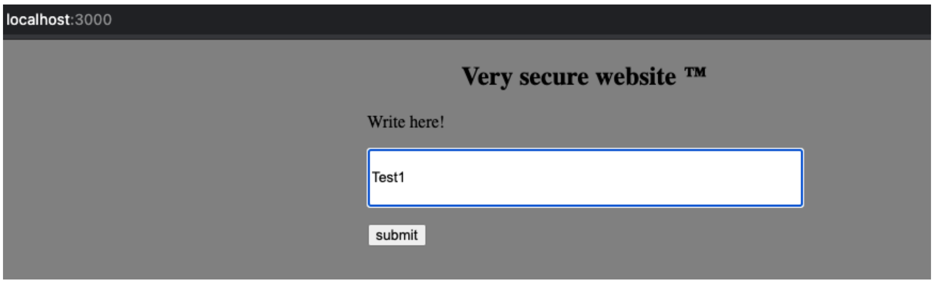

Figure 3.12: First example input sent to the server, using the user interface.

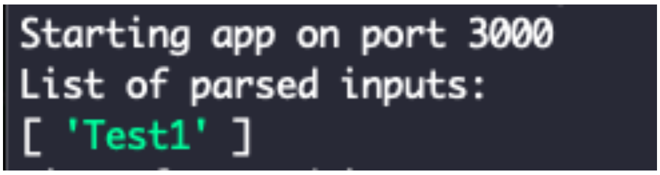

Figure 3.13: Server outputting the result of the first example input

As seen in the images above, the server will respond and update its list of valid inputs it has received for the sake of this experiment. Now, the server will be receiving malicious input to see how it reacts and whether it will interpret it even though it has client-side input handling.

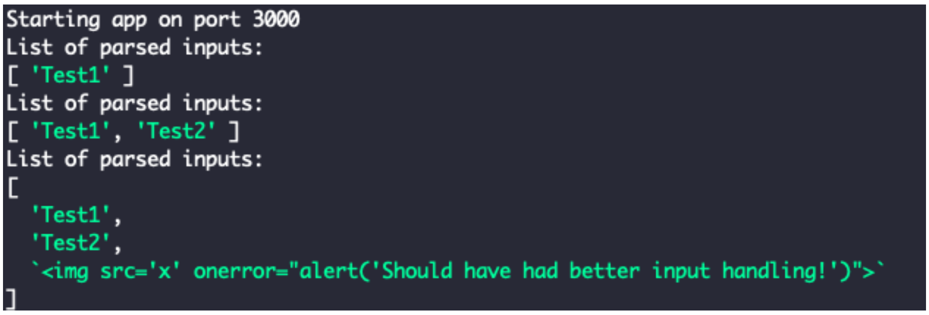

Figure 3.14: Server outputting the result of all valid input it has received

The results demonstrated in image 3.14 depicts that the server will interpret the previously used XSS input as valid and will thus parse it. This demonstrates that web applications that only use client-side logic can easily be bypassed with other methods of sending malicious input. This method of bypassing the client-side logic can also easily lead to persistent malicious attacks.

The same example will be re-implemented now, but with both client-side and server-side logic. The server-side logic will use a similar method as to the example in the code-snippet 3.8, it will this time use the `isAlphanumeric()`, so it will be using the same logic as the client-side regular expression. Similarly, a Python script that sends malicious CURL requests will be executed. The server will be outputting all valid input it has received to demonstrate whether malicious input is still able to bypass the input handling.

```
Starting app on port 3000
List of parsed inputs:
[ 'Test1' ]
List of parsed inputs:
[ 'Test1', 'Test2' ]
The format:
<img src='x' onerror="alert('Should have had better input handling!')">, is not valid and will therefore not be parsed.
List of parsed inputs:
[ 'Test1', 'Test2' ]
```

Figure 3.15: By using server-side validation the web application is able to prevent malicious input from being interpreted

As seen in the image 3.15, the server-side logic can prevent the CURL requests from being parsed and is only allowing alphanumerical input to be interpreted. By using both server-side and client-side input handling methods, this demonstration has created a web application that contains sufficient input handling methods as well as a user interface that provides the users with information about whether their current input is correct.

The following client-side code was used to implement this demonstration.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Test</title>
</head>
<body id="body">
    <h2>Very secure website TM</h2>
    <section id="f">
        <form id="sendinfo" method="POST" action="/userinput">
            <label for="inputfieldname">Write here!</label>

            <input type="text" id="inputfield" name="inputfieldname" pattern="[A-Za-z0-9]+">
            <input type="submit" value="submit">
        </form>

        <p id="userinputparsed"></p>
    </section>
</body>
```

The following server-side code was used to implement this demonstration.

```
app.post("/userinput", urlencodedParser, (req, res) => {
    if (validator.isAlphanumeric(req.body.inputfieldname)){
        res.send(req.body.inputfieldname);
        listOfValidInputs.push(req.body.inputfieldname);
    }
    else {
        res.send("The format you gave is not valid!");
        console.log("The format: ");
        console.log(req.body.inputfieldname + ", is not valid and will therefore not be parsed.");
    }
    console.log("List of parsed inputs:");
    console.log(listOfValidInputs);
});
```

The following Python script was used to send malicious input to the web application.

```
import requests
url = "http://localhost:3000/userInput"
toSend = {"inputfieldname":
            """<img src=’x’ onerror="alert(’Should have had better input handling!’)">"""}
r = requests.post(url, data=toSend)
```

3.6 Conclusions

In the demonstrations from this chapter, it should be clear that examples using server-side input handling were safer than their counterparts that did not use server-side libraries.

After reading through the previous demonstrations, the question to answer is how and why vulnerable web applications sometimes occur? The answer to this question is difficult. However, one might assume that one of the reasons could be that the developers do not think of covering this topic. The method of handling input is not a method that is given for free or built-in when developing and deploying an application, but rather something that the developer must explicitly implement. The methods for handling input, i.e., validation and sanitization, could also be assumed difficult to get right. One example could be the construction of regular expressions for input handling. Developers might construct a regular expression that does not cover all possible input since the set is infinite.

Chapter 4

Methodology and Data Collection

In this chapter, the thesis describes collecting GitHub data to explore and draw conclusions about which of the previously mentioned frameworks are the most common among open source projects.

The thesis will also attempt to answer several different research questions based on the data collected in this chapter. Questions such as which is the most popular JavaScript framework for writing client-side code? Which is the most popular NodeJS library for server-side input handling? Which are the most commonly used regular expressions for input handling? And lastly, the dangers of using modern frameworks, explicitly looking at ReactJS.

The ambition of the thesis was to obtain an overall picture of how web applications handle input data. The methods used in this chapter are to collect data from open sources such as Github, NPM, StackOverflow surveys, and the Common Crawl data set. All these different data sources are sufficient to obtain this picture, i.e., the results that will be produced will give a visualization of the state of practice. The data collected will help with answering the research questions stated in the goals section. Once all the relevant data has been collected, they will be explained further in an upcoming chapter that will depict them.

4.1 Finding The Most Popular Framework

For this thesis to determine the most commonly used framework, it will utilize the Github API to collect data. In almost all web applications, projects that are written in JavaScript or NodeJS, that use external libraries, i.e., npm packages, there will exist a `package.json` file [9]. The file consists of data that describes the project. It consists of a configuration of tools, version descriptions, scripts it runs, a set of libraries that the project uses, etc. For the different frameworks, they are often included in a web applications `package.json` file since the project relies on that external source.

By using the Github API to search for `package.json` files, the results will depict all `package.json` files available on public Github projects. By adding more constraints to the data search, the collection method can specifically search for `package.json` files that contain a given framework. For example, the search query can specify that it wishes to specifically find public repositories that have a `package.json` file, which contains the keyword *react*. This will return several public repositories that have a `package.json` file, which uses the react framework since it will be included in the file. Their results will also consist of `package.json` files that contain the keyword *react* followed by an arbitrary amount of digits or letters since the search API will count them as valid `react` results. This will be brought up in the threats to validity section.

The following search queries were used

- `filename:package.json+react+in:file`
- `filename:package.json+jQuery+in:file`
- `filename:package.json+angular+in:file`
- `filename:package.json+vue+in:file`

Other frameworks available were not considered since the ones listed above are generally viewed as the most popular [28] and well-established frameworks for client-side web development. These were easier to collect data on due to the substantial amount of material and sources available online.

4.2 Most Commonly Used Library for Input Handling

To find the most commonly used library for input handling, a similar method as before is used.

Server-side logic may be written in an abundance of different programming languages. Therefore, due to time constraints, the server-side programming language researched will be NodeJS. NodeJS was the choice since it is one of the most popular tools, specifically a run-time environment, used for server-side development of web applications [16, 30]. NodeJS is an excellent choice for server-side logic since it is the same language used for the client-side. A web application consisting of both a client-side, server-side, and database can be considered a full-stack application. This full-stack application makes it easy for the developer if the entities are written in the same language.

All client-side logic researched in this thesis is in JavaScript, so picking NodeJS as the server-side language is reasonable. NodeJS is a run-time environment that can execute JavaScript code outside of a web browser. It is a popular choice for several reasons, such as its lightweight design and a large amount of open-source external libraries.

Similar to the previous section of collecting data on client-side frameworks, different search queries were constructed to search for `package.json` files that include important keywords, these keywords being the name of input validation libraries used.

However, a new factor that has to be considered for collecting this data is the abundance of different libraries used. The previous section made it easier to define which frameworks to search for since those searched for are the largest and most popular frameworks. Due to a large number of different libraries, the metrics used to determine which libraries to evaluate further will be primarily based on the number of downloads on the online package-manager NPM.

Researching input handling libraries for NodeJS, one of the more popular libraries used, based on its roughly 5 million weekly downloads [24], is the `validator` library. Other popular libraries used are `dompurify` [22], the `xss` [25] library and `express-validator` [23].

The following search queries were used

- `filename:package.json+validator+in:file`
- `filename:package.json+express-validator+in:file`
- `filename:package.json+dompurify+in:file`
- `filename:package.json+xss+in:file`

4.3 HTML Pattern Regular Expressions Collection

Common crawl [8] is a massive data set that contains a substantial amount of different types of raw HTML content scraped from billions of web applications online. The reason behind collecting the pattern attributes from scraped web applications is to get a deeper knowledge and practical insight into how real-life examples develop their regular expressions for their client-side input validation.

The advantage of using the common crawl web scraper for finding these patterns is that it is easy to find many different and unique implementations of regular expressions based on the huge amount of data available. This gives a large data set of results to stand on and can visualize the most commonly constructed regular expressions.

However, it needs to be clarified that these web applications collected may have different methods of validating and handling input in general. It is important to note that the data collected is raw HTML documents rather than complete web applications, i.e., the data that is accessible will be part of a web applications client-side HTML. Due to the data being purely raw HTML, it makes the determination of frameworks used impossible since it can not be determined from viewing HTML.

The data collected does not contain server-side logic either, since that could not be collected. However, it is hypothesized that a given regular expression found from the HTML documents could be used as server-side regular expressions since the developer might assume that the regular expression is valid and thus uses it for server-side validation and client-side.

For collecting this data, a script has been made. To process a set of Common Crawl archive files, collect all HTML documents from these archived files. Once the HTML documents have been collected, the script will parse them to collect all relevant `input` tags with pattern attributes. The relevant tags mentioned implies skipping `input` tags that are buttons. Each tag with pattern attributes will be written to a CSV file. The data are written there will consist of the tag itself and other metadata. This is to create a CSV file that has self-explanatory entries that can also easily be parsed for any reason.

Once the data has been collected, the input tags regular expressions will be tested with a security scanner, which the thesis will later explain in more detail. This security scanner will attempt to evaluate whether a given regular expression collected from the HTML documents will withstand malicious input constructed by the security scanner. The security scanner, in other words, will be attacking the regular expressions found to see whether or not there exists input that can threaten the safety of the regular expressions used.

4.4 ReactJS Dangerous Possibility

ReactJS was shown to be the most commonly used framework based on the latest StackOverflow survey [29]. Therefore, this thesis will focus on understanding ReactJS and its methods of validating input and preventing malicious input.

One interesting part of the ReactJS language is the usage of `dangerouslySetInnerHTML` [37]. The usage of `dangerouslySetInnerHTML` is supposed to resemble the JavaScript `innerHTML` method that has the ability to dynamically manipulate the content of the DOM with any arbitrary HTML or text. The usage of `innerHTML` is considered being bad practice since it can lead to dangerous pitfalls. One example is that malicious users can send malicious input rendered to the DOM if there are no sufficient input handling methods. This creates a danger for web applications that do not properly validate input or sanitize the input but still use the `innerHTML` method for appending new content to the DOM.

As previously mentioned, the usage of React's `dangerouslySetInnerHTML` is supposed to be a replacement to `innerHTML` [37]. It has been specifically named dangerous to inform developers that the usage of this DOM element is dangerous and can lead to the web application being vulnerable to XSS attacks [37].

This makes it interesting for the thesis to use the Github API and construct a search query to figure out how many of the publicly available projects have JavaScripts files that use the `dangerouslySetInnerHTML` element. The results of such a search could further demonstrate why a report on input handling practice is needed.

The following search query was used
`extension:js +dangerouslySetInnerHTML+in:file`

Chapter 5

Results of Collecting Data

This chapter will display and describe the results from collecting data on the previously stated research questions. A subset of the results will later be further discussed as their purpose is to be used as a data set for an ongoing research project.

5.1 Most Popular Framework

The results from the Github API search queries indicate that ReactJS is the most commonly occurring framework in the public `package.json` files available. By explicitly requesting those files by using a Python web scraping method, the script written could correctly receive a `package.json` HTML URL and manipulate it to a raw URL to only receive the JSON object. After ReactJS follows jQuery, AngularJS, and lastly VueJS.

The latest StackOverflow survey [29] also depicts the popularity of different frameworks used by its respondents. The results of the survey are similar to the results of the `package.json` search.

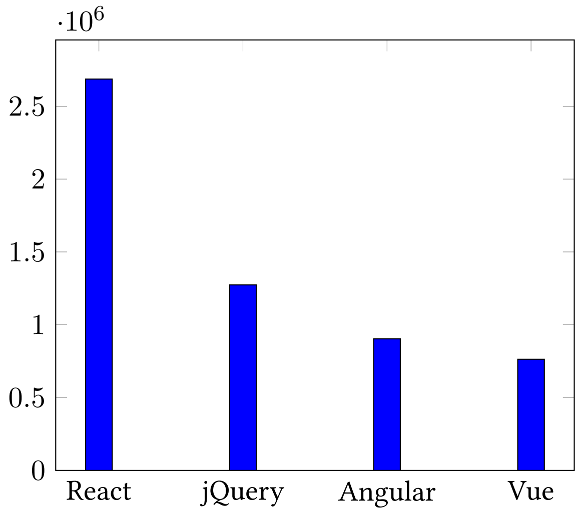

Figure 5.1: Results of collecting data on frameworks.

Based on these results, the demonstrations depicted in chapter 3 used the ReactJS and jQuery frameworks. This is because they are the most predominant frameworks used in open source projects from the data collected.

5.2 Most Commonly Used NodeJS Library for Input Handling

The results can be divided into two different result types. Firstly, the results of constructing search queries using the Github API. Secondly, by manually referring to the number of weekly downloads listed on the library's official NPM page.

Both the Github API and the NPM pages for the different libraries searched for demonstrating that the library `validator` [24] has a larger download rate and occurrence in `package.json` files than the rest of the libraries, making it the most popular.

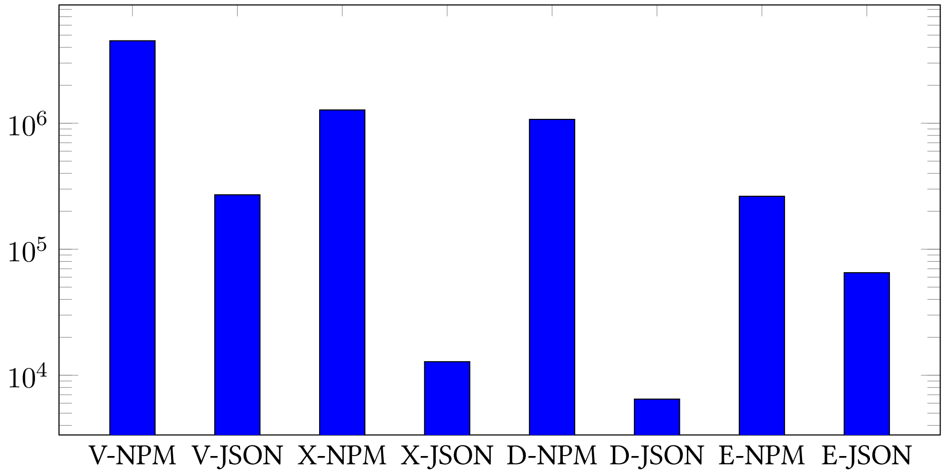

Figure 5.2: Results of collecting data on libraries.

In the results depicted in the chart above, there are eight different x-values. The first one `V-NPM` demonstrates the number of *weekly* downloads for the `validator` library on its official NPM page. The second value `V-JSON` represents the amount of times the `validator` library is present in public `package.json` files on Github. Likewise, for the rest, as depicted in the following table.

V-NPM	Validator NPM, **link**	4510929 downloads
V-JSON	Validator Package JSON files	270214 JSON occurrences
X-NPM	XSS Prevention NPM, **link**	1273585 downloads
X-JSON	XSS Prevention Package JSON files	12805 JSON occurrences
D-NPM	Dompurify NPM, **link**	1072470 downloads
D-JSON	Dompurify Package JSON files	6464 JSON occurrences
E-NPM	Express-Validator NPM, **link**	263120 downloads
E-JSON	Express-Validator Package JSON files	65131 JSON occurrences

Table 5.1: Compilation of the abbreviations

5.3 HTML Pattern Regular Expressions

5.3.1 Structuring the Data

The results from collecting all regular expressions used by pattern attributes were added to a CSV file. The CSV file contained 6 different columns.

URL	Type	Pattern	SMT_Status	Attack_String

Table 5.2: The first row of the CSV file

Each one of these columns contained their corresponding results of parsing the raw HTML of a specific web application. `URL` is a link to the web document parsed.

The `type` column is an HTML attribute which can be found in `input` tags. This refers to the type of input field that it expects and will render. E.g., if the input type is a button, it will render a button-like element in the HTML. When using the `type` attribute, e.g., the email type, there will be a built-in standard regular expression. This makes this data twice as interesting, since by the `type` attribute we can observe metadata about the `input` tag but also if there exists both a valid `type` attribute, it could make the `pattern` attribute redundant. Because the `type` attribute already specifies a built-in regular expression by HTML standards.

The values that the `Pattern` column represents are the actual regular expressions found for a specific `input` tag found in the data set. The script has been iterating through the HTML documents, and it has been extracting HTML tags with pattern attributes. These regular expressions are going to be essential for the thesis, which will be explained later on.

The `SMT_status` column is a representation of whether a given regular expression found in an `input` tag is safe or not. The regular expressions were sent as input to a web security scanner, which will also be explained later. This scanner evaluates regular expressions and returns a result that implies whether the given regular expression was safe. There are 4 different statuses the `SMT_status` can take, namely $\{sat, unsat, error, false\_positive\}$.

The sat status, short for satisfiable, implies that the regular expression accepts an attack string that contains `<script>` tags. A regular expression that is classified as sat is an unsafe regular expression.

The unsat status, short for unsatisfiable, implies that the regular expression is safe based on correct scanner semantics.

The false_positive status implies that the scanner suggested an attack string that could not be accepted when executed in NodeJS, which means that the string is not an attack.

The error status implies that the scanner could not read the regular expression or that another error had occurred.

The last column in this table is the `attack_string` column. As previously explained, if an `SMT_status` is sat, then there will be an attack string that can bypass the regular expression. This column is that attack string. The values of this column are sample XSS inputs that can bypass the regular expressions that are sat from the `SMT_status` column.

5.3.2 Results from Collecting

There were 3500 WARC archive files, i.e., common crawl data set files, parsed to receive these upcoming results. The 3500 files consisted of approximately 12 million HTML documents that were parsed by the script. This led to 79187 regular expressions being found and collected; however, only 1497 of these regular expressions were globally unique. This means that there was a total of 1497 regular expressions after removing all duplicates that were found when parsing the HTML documents.

Unsat	63724
Sat	13852
Error	806
False_positive	806

Table 5.3: Results from parsing the CSV files `SMT_status`

As seen in the results from the table above, most of the regular expressions found were unsat, meaning that they cover all the possible XSS attacks attempted by the web security scanner. Roughly 13k were sat, meaning that there existed an XSS attack that could bypass it. The error entries show that approximately 800 of the regular expressions were not understood by the web security scanner. The remaining entries were classified as false_positive, implying that the regular expression was verified to reject the attack string when re-executed in NodeJS.

The collection of this data led to the thesis concluding web application input patterns and their regular expressions.

The following questions would be most interesting to attempt to solve, based on the results.

- Which type attribute is the most commonly used with the pattern attribute?
- How often is a pattern used when the input type is email?
- Which type attribute is most prone to be satisfiable by the web security scanner?

The total number of HTML documents parsed was approximately 12 million. Out of these 12 million, there were roughly 19 million relevant and interesting `input` tags found, from about 620k unique web pages. Out of these 12 million documents, we found a total of 79187 `input` tags with regular expressions. However, since multiple web pages can use the same regular expressions, or even the same web page can use the same regular expressions several times, the results landed on a total of 1497 globally unique regular expressions.

Based on these roughly 79k `input` tags with regular expressions, the following graph was constructed to visualize the distribution of `type` attributes used when combining it with the `pattern` attributes regular expression.

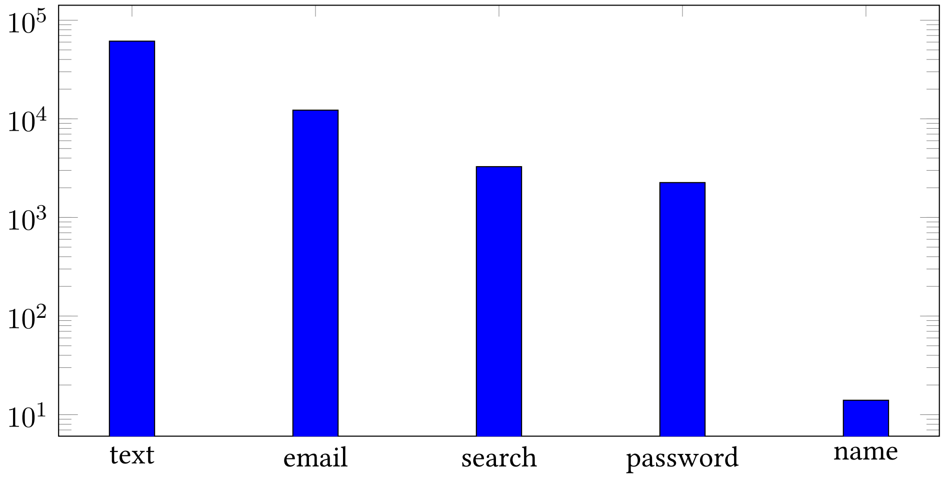

Figure 5.3: Results of collecting data on type attribute values from `input` tags with pattern attributes.

The graph above displays the results from the 79187 `input` tags with `pattern` attributes that were scraped. As seen, the most commonly used `type` attribute value is `text` followed by `email`. This is interesting since by using the `type` attribute and setting it as email, the HTML email standard regular expression will be applied to the input field. In this case, all these `input` tags with email types have specified the `type` being email but then also applied their regular expression to it. This is a redundant choice since the built-in `type` would validate the inputs given. This could also be a potential disadvantage for the creator since an individually made regular expression for emails might be less secure than the standard version. However, there are reasons to add the pattern attributes, e.g., if the developer wants to restrict the set of accepted addresses even further from the regular expression provided by the type attribute.

The final question of this section, i.e to check which `type` attribute is most prone to be attacked. The following graph was constructed based on the results.

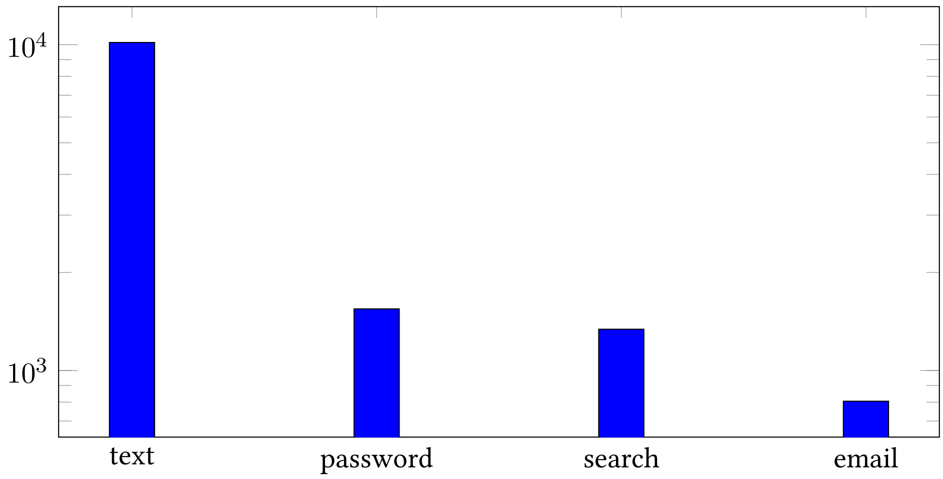

Figure 5.4: The following `type` attributes were the most prone to be satisfiable, based on the web security scanner.

The results displayed in the graph above depicts that the `type` attribute that was most likely to be satisfiable, i.e has an attack string and is unsafe, was the `text` type. There was an occurrence of the types `pin` and `city` in the data. However, their values were too insignificant to include in the graph.

5.4 ReactJS Dangerous HTML

The results of this search query demonstrate that a total of roughly `329000` public Github repositories have JavaScript files that include the usage of `dangerouslySetInnerHTML`. Compare this to the results of the previous section when `package.json` files were evaluated to check whether they used the React framework. This demonstrates that roughly $\frac{329000}{2700000} \approx 12\%$ of the publicly available React projects on Github use the `dangerouslySetInnerHTML` element.

Using the `dangerouslySetInnerHTML` element, a web application needs to present another method of validating the input. Therefore it is encouraged to use the built-in render methods that React provides, which has automatically removed malicious input.

Chapter 6

Discussion

6.1 Common Crawl Data

6.1.1 Collection of Regular Expressions & Their Significance

As mentioned in the results chapter, all HTML tags with pattern attributes were added to a CSV file. This was done to create an easily accessible file that can then be used by the Embedded Systems and Information Security research groups to ease their ability to conduct experiments on the regular expressions.

A CSV file consisting of many lines of regular expressions used in real-life web applications will be good test inputs for their project.

During the previous chapters, the thesis referred to a web security scanner that was used to check whether a given `pattern` attributes regular expression was safe or not. This web scanner is a tool developed by a research project that has been created by the Embedded Systems research group at Uppsala University and the Information Security research group at Chalmers University of technology. The project is called Web Sec. The Web Sec project has built a web application scanner that uses string solving methods. The string solving method is a satisfiability modulo theory solver (SMT) used to verify the correctness of programs and solve string constraints. The Black Ostrich project uses its SMT solvers specifically for strings.

Their project uses the Alexa most commonly used websites, they crawl the given websites and find the HTML pattern attributes to get to the regular expressions they use. The scanning done in the project is mainly to extract regular expressions used in HTML pages that the SMT solvers can later solve. They will be using the SMT solver Ostrich [5], which they implement further with more specific regular expression solving and parsing techniques.

For this project to solve as many regular expressions as possible, it will be good to have a large data set of real-life examples of regular expressions used as input patterns, where the Common Crawl data becomes topical. The data parsed in this thesis project uses Common Crawl to parse and save all pattern attributes, with their URL, regular expressions, other metadata, etc. Providing a data set of this size with abundant real-life regular expressions will be a good test case for their solver. Since their current data set is from the most commonly used websites, which could perhaps have more advanced and secure regular expressions used. This will be a good chance to see what less popular websites implement regular expression input validation as their client-side. They will be able to run this against many different types of regular expressions to determine whether or not they are safe.

The main findings from conducting the experiments on the regular expressions indicated that the Black Ostrich scanner could not bypass a majority of the regular expressions collected. This implies that many of the regular expressions found were able to prevent attack strings from being evaluated by the web application. However, a large set of regular expressions were still bypassed with an attack string supplied by Black Ostrich. Roughly 17% were satisfiable. Almost one-fifth of all regular expressions found from the given data set could be bypassed with an attack string. Preferably, the number of satisfiable regular expressions would be as close to 0% as possible since one such XSS attack could lead to devastating losses for a web application and its users. It could take only one vulnerability of a web application to cause cascading failures in the rest. A dangerous attack string can be used to cause reflected XSS attacks in poorly constructed search strings. For example, web page search functionality is often displayed either in the URL och rendered to the web page view. Therefore, if a search string is poorly constructed, an attack string could cause a reflected XSS attack.

Instead of constructing their regular expressions, the developers constructing these regular expressions should use built-in and tested methods instead. For example, by reconnecting to the server-side examples in chapter 3, the library chosen is a popular and tested library that should be used for input handling when developing with NodeJS. Not using well-established libraries and constructing regular expressions of your own is why the Black Ostrich found nearly one-fifth of the regular expressions used to be faulty.

6.1.2 Client-side & Server-side Assumption

This thesis has not collected any data on what server-side regular expressions there are. However, there could be a possibility of regular expression copying. If this would be the case, then the web applications parsed with insufficient regular expressions most likely have insufficient handling on their server-side if they use similar regular expression methods. In total, if a web application has insufficient input validation in the form of regular expressions for their client-side, there could be a possibility that their server-side regular expressions are too insufficient.

6.2 Client-side or Server-side

The thesis experiments and explanations indicate that both a client-side input handling method and a server-side input handling method are preferred. There are several different reasons why client-side validation is not sufficient to be used alone.
Reasons could be that the attacker can disable JavaScript on a web application and thus bypass the client-side handling. In general, an application should never fully trust what the client-side communicates with. The client-side will be in control of the user. This makes it the weak link in a full-stack web application. Attackers know how to bypass client-side restrictions and exploit the weakest link correctly. New methods of bypassing client-side input handling can also come that the current meta of client-side development does not cover.

However, the advantage of using client-side input handling is that it provides the user with direct feedback. It is therefore recommended to use client-side handling for the sake of providing a good user experience.

Applications that only have server-side input handling provide good security, but rather a less useful user experience developed without client-side input handling and user feedback. Therefore, a combination of both should be used. Combining the strength of server-side input handling with the visualizing user experience of client-side input handling is what makes a full-stack web application complete.

6.2.1 Frameworks

With the rise of frameworks and their quick adoption in web application development, developers need to be prepared for how much trust they are placing in the framework. Most frameworks provide solid methods for handling malicious input, but poor development is completely neglecting server-side handling in the belief that a framework will solve the issue. For example, web applications using old versions of frameworks are the first to fall victim to malicious attacks. Therefore, when using frameworks, developers should always consider that blindly trusting client-side frameworks are not sufficient to fully prevent malicious attacks.

6.3 NPM downloads & JSON occurrences

For this section, the thesis will discuss the reasoning behind its data collection methods. The thesis collected data on common frameworks by using the GitHub API to find `package.json` files. For Node libraries, the thesis collected data using both the GitHub API and the number of weekly downloads for a given library, found on its npm page.

The thesis began with choosing which specific frameworks were to be further looked into for collecting data on the frameworks. The choices of frameworks were based on two criteria. Firstly, the Stack Overflow survey [29] results, which depicted the most commonly used frameworks. Secondly, and less valid, based on what is most heard of today. It is impossible to have missed that frameworks such as React, Angular, and Vue are growing a ton in popularity. Then, using the Github API, determining how common the frameworks are in open source repositories could easily be delimited.

However, collecting data on Node libraries went a bit differently. A similar method of using the Github API was done, but to delimit which libraries would be searched for was a bit different. Anybody can write and publish a package on NPM that other developers can later use. However, due to it being so easy and no package being the industry standard, it made selecting the "*right*" packages harder than selecting the frameworks. The Node ecosystem [4] is mostly based on using online open-source developed packages when developing. This is not necessarily a disadvantage since open-source libraries allow developers to see what is going on behind the scenes when calling library-specific functions, but this leads to an abundance of libraries that often have similar purposes. Therefore, when choosing which Node libraries to look further into, the NPM search was primarily used. It is when searching for NPM packages that the most commonly downloaded were found. This was then later combined with Github `package.json` occurrences to further strengthen the argument of choosing said packages.

6.4 Threats to Validity

6.4.1 Usage of Package.json

A threat against the validity of collecting the data in this method could be based on the usage of the `package.json` file. Since the method of collecting data was used by checking the occurrences of a given framework or library in arbitrary public `package.json` files. The reasoning behind this method of collecting data is due to Github listing ReactJS and any other framework as JavaScript, which prevents a regular search query from being constructed for repositories written in a specific programming language or framework.

The first threat could be the public repositories that only consist of a `package.json` file. Suppose that an unfinished project only pushed its package file to the repository. This would not be an entirely accurate result; the only reason it is found is due to the `package.json` file.

Another reason why React might have been included in many package files could be due to the usage of React Native. React Native is a framework for writing mobile applications using JavaScript as the language. Package files that include the keyword *React Native* would also return based on the search query constructed, which could perhaps be a reason why React was much more common than the rest of the frameworks. However, similar XSS attacks can be constructed for React Native applications, thus making it included in the React total data set results.

Besides from React Native, other faulty results could, unfortunately, be included in the `package.json` files. E.g files that contain keywords such as *reactor*, *reaction*, *reactive*, etc, can also be interpreted and returned as valid results by the search query.

However, when searching for the previously mentioned keywords, the following results were found.

- reactor - 4059
- reaction - 1577
- reactive - 12089

The number of occurrences found when only searching for React is roughly greater than 2800000—making these keywords negligible in the total.

6.4.2 Multiple packages per repository

The case of a repository consisting of several `package.json` files could also potentially skew the results. If a repository consists of several different `package.json` files, perhaps one in the client-side directory and one in the server-side directory, then this would cause the results to double if they contain similar keywords being searched for. Optimally, when searching for a keyword in the `package.json` files available on public repositories, then the repository will only contain one such file.

6.4.3 NodeJS Libraries

A threat against the validity of choosing `Validator`, `XSS`, `Dompurify`, and `Express-validator` as the main libraries to search for could be if there exists other libraries that are more commonly used. The data collection referred to open source projects when searching on Github. However, the weekly downloads on their official NPM pages prove that they are used a lot. There were other libraries for validating input, and however, based on the fewer downloads they had, they were not selected.

6.4.4 ReactJS Inner HTML

The search query constructed for collecting data on `dangerouslySetInnerHTML` searched specifically for JavaScript files that include that keyword. A threat against the validity of the results could be that a comment that contains the word dangerouslySetInnerHTML would count as an occurrence, even though the project does not use it. However, it is unlikely that projects contain comments of those keywords.

6.5 Future work

This thesis has collected a rather large data set of regular expressions used in HTML patterns. The Embedded Systems research group can use this data for their program verification and string solving projects. It will be a great use case for their web scanning research.

Others can also re-use this data to identify how regular expressions used in client-side validation are changed over time. Currently, the majority of the data collected is from 2021. Who knows what kind of regular expressions will be used in the future? Therefore, a data set containing regular expressions from the current year can be good for future improvements to security using regular expressions as validation techniques.

Others can also use the data to analyze the regular expressions used by either client-side or server-side. As previously stated, the data collected is regular expressions used in HTML patterns, specifically client-side validation methods. There has not been any data collected on server-side regular expressions in this thesis. In the future, one could analyze the differences between client-side regular expressions and server-side. A good starting point would be to check whether the regular expressions used for the client-side are identical to the server-side, i.e., are the regular expressions just re-used?

Chapter 7

Conclusions

As stated in the goals, this thesis aims to present different methods used for building secure web applications and determine whether those methods are sufficient. These methods were demonstrated in the mockup chapter and backed up with several explanations of how they can successfully prevent malicious threats.

7.1 Common Crawl Data

The common crawl data collected demonstrated that out of a total of 12 million HTML documents parsed, we could find that there are 79187 `input` tags with `pattern` attributes. Out of these 79187 regular expressions found from their `pattern` attributes, there was a total of 1497 that were globally unique. This means that they only occurred once over these 12 million HTML documents. Which is roughly 0.01% of the `input` tags parsed.

These 79187 regular expressions were later input to the Black Ostrich web scanner to be evaluated on their safety. Roughly 63k, the majority, of these regular expressions were evaluated as unsatisfiable, implying that the Black Ostrich scanner could not solve them and thus cover the XSS attacks attempted by it. Approximately 13k were satisfiable, implying that the Black Ostrich scanner could bypass them and were thus supplied with a sample XSS attack that would bypass the given regular expression.

Out of all these roughly 13k unsafe regular expressions, it was interesting to observe what was the most common `type` attribute used in combination with it. The results demonstrated that `text` was the most common `type` by quite a margin, followed by `password`, `search`, and `email`. Interesting results since it depicts that, e.g., `text` fields are more easily bypassed than the rest. The fact that `search` was one of the more commonly bypassed `type` attributes was also an interesting observation. Since, as previously mentioned about DOM-based XSS attacks, they are usually constructed such that the XSS input is sent as a search query to a web application. The false_positive status was also an interesting value since it was a given status if a Black Ostrich string could bypass the regular expression but failed when supplying the string to the regular expression in NodeJS to check whether they match. This was done because the Black Ostrich scanner does not implement all the functionality in the JavaScript regular expression, which is why satisfiable strings needed to be verified by running them through NodeJS regular expression. The number of false_positive values compared to satisfiable values was still considerably less, implying that Black Ostrich did classify the majority correctly.

7.2 Combination of Client-side and Server-side

This thesis has given a walk-through of how different logical implementations of input handling can be made. It has been implemented both on the client-side and on the server-side. The thesis has also provided evidence demonstrating that input handling is of utter importance and needs to be implemented on both sides. There is also the ability to add a WAF to the application even further to improve its security against malicious threats.

7.3 Avoid direct input from users to DOM

An essential rule from this thesis is that developers should always avoid rendering input from users to the web application unless it has passed layers of input handling methods. If the input contains malicious content, the web application is bound to crash and burn.

7.4 Never trust client-side only

Another fundamental rule to consider when developing web applications is never to put complete trust in the client-side. As stated, the malicious user has full control of the client-side, which makes it an exposed part of the system since it can be exploited. The server-side, on the other hand, will not be in control of the malicious user. On the client-side, systems that put all their trust, i.e., their input handling methods, will be the first to be exploited. This was also proven in the experiments made, the image 3.14, where the client-side logic was easily bypassed by using CURL requests. There are many different ways to bypass client-side logic, and with time there can come even more ways to bypass it.

7.5 Hypotheses, were they valid?

We hypothesized that client-side input handling would not be sufficient for handling input. This hypothesis was proven to be valid since it exists several different ways to bypass client-side handling. Instead, client-side input handling should be used for a better user experience than blindly relying on its input handling methods.

It was also hypothesized that modern frameworks handle input, but they should be combined with server-side libraries. This was also proven to be true since developers should never entirely rely on the client. Modern frameworks, such as React, present methods for preventing malicious inputs, but using them blindly can lead to downfalls. Therefore, combining frameworks' built-in methods with server-side logic is the best choice for a secure web application.

Lastly, was that server-side libraries will be sufficient. This is true in the sense that they can cover and prevent malicious input, but the catch is that it provides a less helpful user experience.

Bibliography

[1] Acunetix. Cross site scripting xss, 2021. https://www.acunetix.com/websitesecurity/cross-site-scripting/.

[2] Acunetix. Types of xss: Stored xss, reflected xss and dom-based xss, 2021. https://www.acunetix.com/websitesecurity/xss/.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

[4] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node.Js in Action*. Manning Publications Co., USA, 1st edition, 2013.

[5] Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL):49:1–49:30, 2019.

[6] CloudFlare. What is a web application firewall (waf)?, 2021. https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/.

[7] Victoria Collins. The decline of the native app and the rise of the web app, April 2019. https://www.forbes.com/sites/victoriacollins/2019/04/05/why-you-dont-need-to-make-an-app-a-guide-for-startups-who-want-to-make-an-app/?sh=16791f056e63.

[8] Common Crawl. Common crawl, 2021. https://commoncrawl.org/about/.

[9] Flavio Copes. The package.json guide, 2018. https://flaviocopes.com/package-json/.

[10] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. 2007.

[11] OpenJS Foundation. jquery, write less do more, 2021. https://jquery.com/.

[12] Google. Angular, 2021. https://angularjs.org/.

[13] Jessica Henson. 55 web design statistics in 2021 [infographic], 2021. https://www.ukwebhostreview.com/blog/web-design-statistics/.

[14] Jastra Ilic. Cross-site scripting (xss) makes nearly 40% of all cyber attacks in 2019, 2019. https://www.precisesecurity.com/articles/cross-site-scripting-xss-makes-nearly-https://www.acunetix.com/websitesecurity/xss/40-of-all-cyber-attacks-in-2019/.

[15] Facebook Inc. React, a javascript library for building user interfaces, 2021. https://reactjs.org/.

[16] EBS integrator. Top 10 server-side programming languages in 2020, 2020. https://ebs-integrator.com/blog/basics-top-10-server-side-programming-languages-in-2020/.

[17] MDN-Docs. Html attribute: pattern, 2021. https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/pattern.

[18] MDN-Docs. The html5 input types, 2021. https://developer.mozilla.org/en-US/docs/Learn/Forms/HTML5_input_types#client-side_validation.

[19] MDN-Docs. Regular expressions, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions.

[20] MDN-Docs. What is javascript?, 2021. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#what_is_javascript.

[21] MITRE. Common Weakness Enumeration, March 2019. https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[22] NPM. Dompurify, 2021. https://www.npmjs.com/package/dompurify.

[23] NPM. Express validator npm, 2021. https://www.npmjs.com/package/express-validator.

[24] NPM. Validator npm, 2021. https://www.npmjs.com/package/validator.

[25] NPM. Xss npm, 2021. https://www.npmjs.com/package/xss.

[26] Oracle. What is input validation and sanitization?, 2021. https://download.oracle.com/oll/tutorials/SQLInjection/html/lesson1/les01_tm_ovw3.htm.

[27] Stack Overflow. 2020 developer survey, 2020. https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages.

[28] Stack Overflow. 2020 developer survey, 2020. https://insights.stackoverflow.com/survey/2020#developer-roles.

[29] Stack Overflow. 2020 developer survey, 2020. https://insights.stackoverflow.com/survey/2020#technology-web-frameworks.

[30] Stack Overflow. 2020 developer survey, 2020. https://insights.stackoverflow.com/survey/2020#technology-other-frameworks-libraries-and-tools.

[31] OWASP. Input validation and data sanitization, 2015. https://wiki.sei.cmu.edu/confluence/display/java/Input+Validation+and+Data+Sanitization.

[32] OWASP. Application threat modeling, 2021. https://owasp.org/www-community/Threat_Modeling.

[33] OWASP. Cross site scripting xss, 2021. https://owasp.org/www-community/attacks/xss/.

[34] OWASP. Input validation cheat sheet, 2021. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html.

[35] Stefan Prandl, Mihai Lazarescu, and Duc-Son Pham. A study of web application firewall solutions. In Sushil Jajoda and Chandan Mazumdar, editors, *Information Systems Security*, Cham, 2015. Springer International Publishing.

[36] PurpleSec. 2020 cyber security statistics the ultimate list of stats, data & trends, 2020. https://purplesec.us/resources/cyber-security-statistics/.

[37] React. Dom elements, dangerouslysetinnerhtml, 2021. https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml.

[38] ReactJS. Jsx prevents injection attacks, 2021. https://reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks.

[39] Robert Seacord. Top 10 secure coding practices, may 2018. https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices.

[40] Rob Sobers. 134 cybersecurity statistics and trends for 2021, 2021. https://www.varonis.com/blog/cybersecurity-statistics/.

[41] Smart Eye Technology. Confidentiality, integrity, & availability: Basics of information security, 2021. https://smarteyetechnology.com/confidentiality-integrity-availability-basics-of-information-security/.

[42] Joe Tidy. Police launch homicide inquiry after german hospital attack, 2020. https://www.bbc.com/news/technology-54204356.

[43] Wikipedia, 2021. https://en.wikipedia.org/wiki/Defense_in_depth_(computing).

[44] Wikipedia. Document object model, 2021. https://en.wikipedia.org/wiki/Document_Object_Model.

[45] Wikipedia. General data protection regulation, 2021. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation.

[46] Evan You. Vue.js, the progressive javascript framework, 2021. https://vuejs.org/.