CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Capability of Security Scanners:

Evaluating and Extending the Coverage
with Regards to Cross-Site Scripting

Master's thesis in Computer Systems and Networks

Philip Antonsson
Vincent Carlson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Capability of Security Scanners:

Evaluating and Extending the Coverage with Regards to Cross-Site Scripting

Philip Antonsson
Vincent Carlson

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Capability of Security Scanners:
Evaluating and Extending the Coverage with Regards to Cross-Site Scripting
Philip Antonsson
Vincent Carlson

© Philip Antonsson, Vincent Carlson, 2024.

Supervisor: Pablo Picazo-Sanchez, Computer Science and Engineering
Supervisor: Daniel Hausknecht, Polestar
Examiner: Romaric Duvignau, Computer and Network Systems

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

Capability of Security Scanners:
Evaluating and Extending the Coverage With Regards to Cross-Site Scripting
Philip Antonsson
Vincent Carlson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Cross-site scripting is one of the biggest threats for websites and their users, and with the ever evolving technologies it can be hard for developers and users to know if they are vulnerable to cross-site scripting. In the last couple of years, server-side rendered frameworks have grown in popularity, though how vulnerable this technology is to cross-site scripting is not entirely clear. In this thesis, we evaluate how vulnerability scanners preform on different frameworks used in web development. As well as highlight what shortcomings they have along with potential problems this poses. We found that Remix especially does not have the support from the current tools that it needs, as Remix has server-side rendering, credence can then be put towards the speculation that it needs further focus. To mitigate this we propose an extension which extends CodeQL with custom queries to better suit the shortcomings we identified.

Keywords: CodeQL, Cross-Site Scripting, React, Remix, TypeScript, Vulnerability Scanning, Web-Development.

Acknowledgements

We want to take this opportunity to express our gratitude to Benjamin Eriksson, Pablo Picazo-Sanchez and Daniel Hausknecht for their support, guidance and help during the writing of this thesis. Additionally, we would like to extend our thanks to Chalmers and the teachers there that gave us the required knowledge for this thesis, specifically we want to thank Magnus Almgren, and Andrei Sabelfeld. They gave us the introduction to the security field and held, in our opinion, excellent courses. Lastly we want to thank Polestar for allowing us the insight into how security work is done withing the industry and also providing us with a place to work on the thesis.

Philip Antonsson and Vincent Carlson, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

API	Application Programming Interface
CSR	Client-Side Rendering
CSS	Cascading Style Sheets
DOM	Document Object Model
FOSS	Free Open-Source Software
HTML	Hyper Text Markup Language
JS	JavaScript
JSX	JavaScript XML
OWASP	the Open Worldwide Application Security Project
SSR	Server-Side Rendering
TS	TypeScript
TSX	TypeScript XML
XML	Extensible Markup Language
XSS	Cross-Site Scripting

Contents

List of Acronyms **vi**

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

2 Background **5**
2.1 Server-Side Rendering 5
2.2 Client-Side Rendering 6
2.3 Web Frameworks 6
2.4 Cross-Site Scripting 7
2.4.1 Reflected XSS 7
2.4.2 Stored XSS 7
2.4.3 DOM-Based XSS 8
2.5 Vulnerability Scanning 10
2.5.1 Black box Scanning 10
2.5.2 White box Scanning 11

3 Method **13**
3.1 Test Environment Creation 13
3.2 Scanner Evaluation 14
3.3 Tool Extension 15

4 Results **16**
4.1 Vulnerabilities Found 16
4.2 Test Environments 16
4.3 Existing Analysis Tools 17
4.4 Extended Queries 20
4.5 Summary 21

5 Discussion **22**
5.1 Website creation 22
5.2 Analysis methods 22
5.3 Extended Queries 24
5.4 Ethical Aspects 24

5.5 Typing and Framework Safety 25
5.6 Related Works 26

6 Conclusion **29**
6.1 Future Work 29

A Appendix 1 **I**
A.1 userControlled.ql II
A.2 sanitisedDangerouslySetInnerHtml.ql III
A.3 unSanitisedDangerouslySetInnerHtml.ql IV

B Appendix 2 **V**

C Appendix 3 **IX**
C.1 Arachni Report Main Page IX
C.2 Arachni Report Issue Detail View XII

D Appendix 4 **XV**
D.1 Remix Contacts: Main Page XV
D.2 Remix Contacts: Contact Create Page XVI
D.3 Remix Contacts: Contact Page XVII
D.4 Remix Contacts Image Upload Page XVIII

E Appendix 5 **XIX**
E.1 Remix Real Time App: Post Edit XIX
E.2 Remix Real Time App: Main Page XX

F Appendix 6 **XXI**
F.1 NextJS Dashboard: Comment Publishing Page XXI
F.2 NextJS Dashboard: Main Page XXII

G Appendix 7 **XXIII**
G.1 JQuery input: Main Page XXIII

List of Figures

2.1 Flow of requests and workload distribution between client and server when serving websites using SSR. The numbers indicate the order generic actions would be performed. 5
2.2 Flow of requests and workload distribution between client and server when serving website using CSR. The numbers indicate the order generic actions would be performed. 6
2.3 Illustration of the steps and interactions between the attacker, client and server in a reflected XSS attack. The numbers indicate the order generic actions would be performed. 8
2.4 Illustration of the steps and interactions between the attacker, client and server in a stored XSS attacks. The numbers indicate the order generic actions would be performed. 9
2.5 Illustration of the steps and interactions between the attacker, client and server in a DOM-based XSS attacks. The numbers indicate the order generic actions would be performed. 9
2.6 Chart showcasing the work flow of black and white box security scanners, and how they are used in the development process. 10

4.1 The result of the quantitative Arachni test. Showing the total amount of XSS vulnerabilities (*Total XSS*). The XSS vulnerabilities arising from insertion in HTML element content (*XSS-HTML-insert*). The XSS vulnerabilities where it was possible to force the page to execute JS code (*In script context*). Lastly the DOM-based XSS vulnerabilities, where JS code was executable by modifying the DOM (*DOM-based*). 18
4.2 Average and standard deviation of the vulnerabilities found in the quantitative Arachni test presented in Figure 4.1. 19
4.3 The amount of vulnerabilities verified as false positives in the quantitative Arachni test presented in Figure 4.1. 19
4.4 Average, average rate and standard deviation of the amount of false positives in the quantitative Arachni test presented in Figure 4.3. . . 20

5.1 Example of string shortening when using toString() in CodeQL. Leftmost column are the variables names. 25

List of Tables

3.1 Table listing the selected frameworks for this thesis. 13
3.2 Table listing the selected security scanners for this thesis. 15

4.1 Compatibility matrix of frameworks (X-Axis) and security scanners (Y-Axis). **Legend:** [C:] Compatible, [NC:] Non Compatible, [CF:] Complete Findings, [PF:] Partial Findings, [NF:] No Findings, [FP:] False Positives. 21

1
Introduction

The world wide web is static by nature, with predetermined HTML files that are served to the client. However, due to a desire for more interactive web pages rendition has moved from the server to the client. The server then only works to provide each client with the necessary data related to the requested page. This poses the question of what and how much data should be sent to the user as delays from the client making a request are undesirable. However, sending everything increases the risk of unauthorised information leakage and the potential of vulnerabilities grows. Clearly, some middle ground would be advantageous where only the data needed for interactivity is supplied and static information is generated by the server [1].

Motivation
In 2017 OWASP (the Open Worldwide Application Security Project) classified injection attacks, which includes cross-site scripting (XSS) attacks, as the number one threat in their top ten website security risk list. While in the most recent edition (2021) it has dropped to position three, this still indicates the severity of injection attacks [2]. OWASP goes on to show that 3.3% of all tested applications were susceptible to injection attacks. With the size of the internet, this constitutes a large number of applications. Furthermore, a large scale analysis of the Alexa top 5000 websites indicates that 9.6% of the analysed websites had at least one XSS problem [3].

Modern websites serve highly dynamic content, to support their development, component-based front-end frameworks like Angular, React, or Vue.js was created. What these frameworks have in common is that the website is compiled on the client-side and populated with data pulled through separate requests to the server. In recent years however, many website frameworks started combining the best of both worlds by using the power of component-based front-end frameworks with Server-Side Rendering (SSR). SSR eliminates the need for the additional data requests after the initial page load, effectively improving web page performance.

While the main motivation of this combination is performance, questions about the resulting security impacts SSR potentially causes are left unanswered. The usual flaws, such as XSS and information leaks, are especially interesting when evaluating a platform. Arguably even more so on newer, less tested, or recently altered frameworks, to catch any security flaws before they can cause harm. Additionally, due to the history of SSR, the possibility of old bugs and security flaws reoccurring in the modern implementations is a risk which warrants caution.

While there is a gradient to web served content we have chosen to refer to it all as "website(s)" in this paper. Essential to our work and central to XSS we also assume interactivity that modifies the displayed content to some extent. The reasoning being that other forms of websites exist but are less interesting when examining XSS vulnerabilities. Following this, "website(s)" will be used from this point on with no regard to the function as long as it fulfills the stated requirement.

Web frameworks are a specific type of system frameworks created to help during web development. There exists several alternatives with different strengths, weaknesses, and use cases. React is one of the most used full stack frameworks[1] according to polled developers [4, 5]. It lets the developer create interactive user interfaces with a mix of HTML, CSS and JavaScript/TypeScript. Remix is another web framework built upon React which adds several functionalities, though most importantly in regards to our thesis it introduces support for SSR [6].

JavaScript (JS) is a lightweight, high-level programming language often used to implement interactivity on web pages [7]. JS consists of two main components: the core language (ECMAScript), and the web Application Programming Interfaces (APIs) e.g. the document object model (DOM) [8]. As of 2024, it is found in 98.9% of web pages [9]. While this indicates the usefulness and versatility of JS, it does not indicate the safety or correctness of the language. There have been (and still are) several safety issues present in JS. Some of, if not most, arise from careless development and JS not alerting the developer to the resulting issues [10].

JS often exhibits unexpected behaviours, largely caused by it being an untyped language [11]. TypeScript (TS) is an extension of JS developed by Microsoft that adds typing and thus enables more of the errors to be caught before running the code. For example, the TypeScript compiler will give you an error if you try to execute a string as if it were a function, while JS would simply let you try. In turn, the runtime will attempt to handle what is supplied and potentially give an error [12]. Despite these advantages, the adoption of TS is sizeable but still somewhat lacking relative to JS. According to stack overflows´ developer survey in 2023, 38.38% of respondents selected TS while 63.61% answered JS when asked "Which programming, scripting, and markup languages have you done extensive development work in over the past year, and which do you want to work in over the next year?" [13].

When serving a user with web-content there are two main methods, SSR and Client-Side Rendering (CSR). These can both coexist or be used solely on their own depending on the need of the web-page and the capability of the server/client. SSR requires more from the server and is more static in regard to displaying the content. This comes from the fact that the server supplies the client with pre-rendered HTML. On the other hand CSR renders the web-page on the client (i.e. the browser takes some input and generates the HTML code) which allows a more dynamic

[1] Node.js has a higher percentage of respondents in the cited survey, though it is a run time environment and thus not a web framework.

representation suited to the clients device, but is also limited to the capabilities of said device. Thus it can be argued that besides increased control of the web-page, SSR also can provide a less volatile experience for the user in terms of response time, as the capacity of their device is only used to display and not calculate any logic related to the web-page. Hence a hybrid solution of both SSR and CSR can be beneficial where interactivity is maintained as well as controlled loading times if static elements are rendered on the server [14]. This solution is implemented in Remix with the use of components, where only the components affected by user input are rendered by the client and the remaining static components are rendered by the server to be combined for the complete website.

Both the placement in the OWASP top 10, and the high incident rate indicate the threat XSS poses for applications. This problem has been widely known for many years, but even so it remains as one of the biggest threats. This stems from the nature of the attacks, where the attacker designs payloads that appear as valid input but in reality escapes the intended path and executes as code. Further issues arise from the fact that to protect their application user supplied data needs to be sanitised, either through an allow-list or a block-list. Allow-list, as the name implies, works by defining input alternatives or what form these are allowed to be in. Block-list is the inverse of this where non valid terms or patterns are specified. They both have their advantages and disadvantages, the trouble with them is the need to know all allowed or blocked strings for the free form nature of user supplied text input. It is practically impossible to know all potential attack payloads that can result in malicious code being executed by the website, but conversely it can be equally hard to know all valid patterns or types of input. This can result in a miss configured allow-list that possibly hampers usage by not including all legitimate inputs.

Challenges
There is an uncertainty concerning the effectiveness of and mitigation's against XSS attacks targeting server-side rendered websites. As an example there existed an XSS vulnerability in React that was present in server-side but not client-side, until mid-2018 [15]. This shows that the threats to server-side and client-side are not necessarily the same, and therefore the two cannot always be secured in the same way. Another challenge is the vast amount of frameworks and tools which are used. The frameworks related to website development have many solutions and functions that overlap though they differ in order or exact wording. This introduces the challenge of what exact function to look for and where in the structure of the source code to find it.

Goals
The goal of this master thesis is to evaluate the potential security flaws and strengths of SSR in modern web frameworks, and how well the current tools are able to find these vulnerabilities. More specifically in the context of XSS attacks.

- **Ground Work** - The first part of this master thesis is, by necessity, a field review of the current vulnerabilities and prevention methods. This also includes installing all the necessary tools and environments, and making sure

they work as intended before they are used in the later parts of the project.

- **Framework evaluation** - Later, findings from the literature review portion of this mater thesis are directed at specific frameworks in an attempt to find patterns, matches or trends in what vulnerabilities are present in the modern-day frameworks. For the basis of this evaluation, XSS attacks are aimed at a combination of existing open source websites and websites created by us. Specifically the structure of the different frameworks will be investigated in regards to how existing vulnerability detection methods perform. To conclude if the methods work better or worse on less prevalent frameworks. The frameworks and detection methods will be cross compared to create a matrix over which methods work for which type of framework.
- **Vulnerability evaluation** - Vulnerabilities, both past and current, will be evaluated as to find patterns or specific problem areas, e.g. especially dangerous functions in Remix. These vulnerabilities are at the core of the thesis and will play a crucial role in a large portion of the work.
- **Detection method implementation** - The creation of a detection method is an important part of the thesis. The matrix created in the framework evaluation will be used to decide which type of framework will be targeted.

Paper Structure

The paper is structured into the following chapters: Background, Method, Results, Discussion, and Conclusion. Background (Chapter 2) introduces the knowledge needed for the thesis, as well as the underlying concepts used in our analysis and extension. Following this is Method (Chapter 3) where the steps we took are presented. Next is Results (Chapter 4), here the results and findings from the previous chapter are presented. After this is Discussion (Chapter 5) consisting of our observations of the results, work done, and technologies used during this thesis. Last in the thesis is Conclusion (Chapter 6) where our closing thoughts and future work is explained.

2

Background

This section introduces and explains background necessary for this thesis. It explains website rendering, web frameworks, XSS, and the different kinds of vulnerability scanning.

2.1 Server-Side Rendering

When using SSR, HTML is generated and sent by the server when a request is received from a client. This offloads more of the work that the client has in CSR, allowing for less powerful clients. The server side rendering also results in clients being able to display the website without downloading the JS. This can result in a speed-up, especially for clients with slower internet connections [16]. How a web page is served when using SSR is illustrated in Figure 2.1.

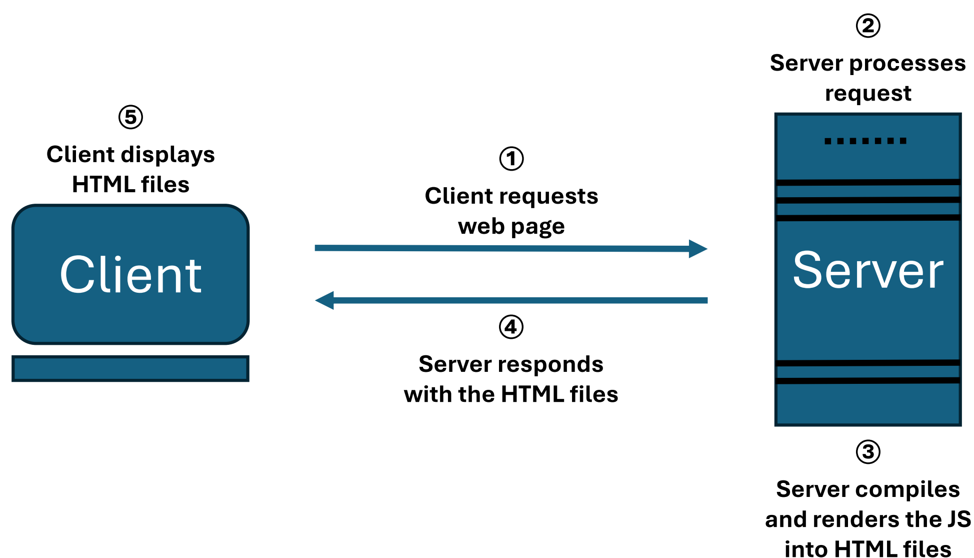


Figure 2.1: Flow of requests and workload distribution between client and server when serving websites using SSR. The numbers indicate the order generic actions would be performed.

2.2 Client-Side Rendering

Client-side rendering makes use of the clients hardware to a larger degree than SSR. With CSR, the server responds to the clients request with HTML and JS files. This means that the clients need to download and run the JS before it is able to correctly display the website. This requires clients with more processing power and better internet connection than the SSR equivalent. It does, however, place less requirements and importance on the server. Changing the contents of the website when using CSR can be done quickly by only sending a single JS file containing the necessary changes, instead of sending a whole HTML file [17]. How a web page is served when using CSR is illustrated in Figure 2.2.

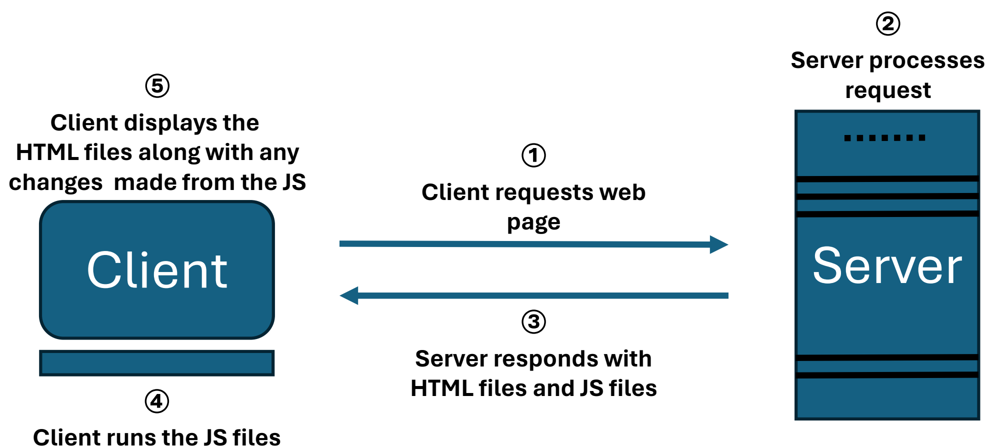


Figure 2.2: Flow of requests and workload distribution between client and server when serving website using CSR. The numbers indicate the order generic actions would be performed.

2.3 Web Frameworks

Web frameworks are a type of software library created to help the development of websites. Frameworks can have a large variety of use cases. Two examples being Express, which is made for simple single-page applications [18] and Remix, which is a full stack framework made to handle all development [6]. They generally use a combination of HTML, CSS, JS and depending on the framework, some other custom function and functionalities, such as the built-in route handling in Remix [19]. Web frameworks are libraries to make web development easier and allow complex pages to be made without the need to implement all the functions, much like libraries in other coding environments. The source code of the website can be more readable and standardised over different projects as the structure and function calls will be the same when the same web framework is used.

React is a front-end JS framework using a component based structure to build the user interface. It should be noted that React also supports TS. It is a Free Open-

Source Software (FOSS) library developed and maintained by Meta Platforms, Inc. (formerly Facebook) [5].
Next.js is developed by Vercel Inc, and is a FOSS React based full-stack web framework created in 2016. It makes use of React components, coupled with additional features and optimisations. Its main features are routing, rendering, simplified data fetching, styling support, optimisations, and improved TS support [20].
Remix is a full-stack open source framework based on React, created by Remix Software Inc then later acquisitioned by Shopify Inc. Remix consists mainly of four things: a compiler, a server-side HTTP handler, a server framework, and a browser framework. One of its main contributions is SSR and fluid route handling [6, 19, 21]. Remix can be deployed to both a Node.js or a non Node.js server environment.
Node.JS is open-source and a back-end solution that allows the server to run JS. It is currently developed by the Linux Foundation as it merged with OpenJS [22].
JQuery is a front-end JS framework with a main focus on the Document Object Model (DOM) and asynchronous data loading (refereed to as AJAX). It is a FOSS library using the MIT licence, initially released in 2006 with widespread usage since [4, 23].

2.4 Cross-Site Scripting

XSS is an injection attack, where a malicious user injects code to a website where another type of (non code) input is expected [24]. This code is then executed by the server or client, since the injected code is hard to differentiate from the websites code. There are several sub-types of XSS, with the most common being: reflected, stored, and DOM-based.

2.4.1 Reflected XSS

Reflected XSS attacks are XSS attacks where the user is redirected with a supplied link which has some malicious payload injected into the URL. This payload is then reflected by the server. Thus when the website is loaded for the client, the script in the payload is also executed. That happens since it appears to be part of the original, trusted, website. Reflected XSS is sometimes called non-persistent XSS, since the attack only exists for users which have been supplied and interacted with the link. The attack does not affect websites users which have not interacted with the altered link [24]. The typical flow of a reflected XSS attack is shown in Figure 2.3.

2.4.2 Stored XSS

Stored XSS attacks is a persistent type of XSS attack. In a stored XSS attack the malicious code is injected into the website and then affects every subsequent user of the website. This injection can occur in several ways, one example being in a comment section. When another user then requests information (e.g. the comments) from the web server the malicious script is supplied along side with the legitimate

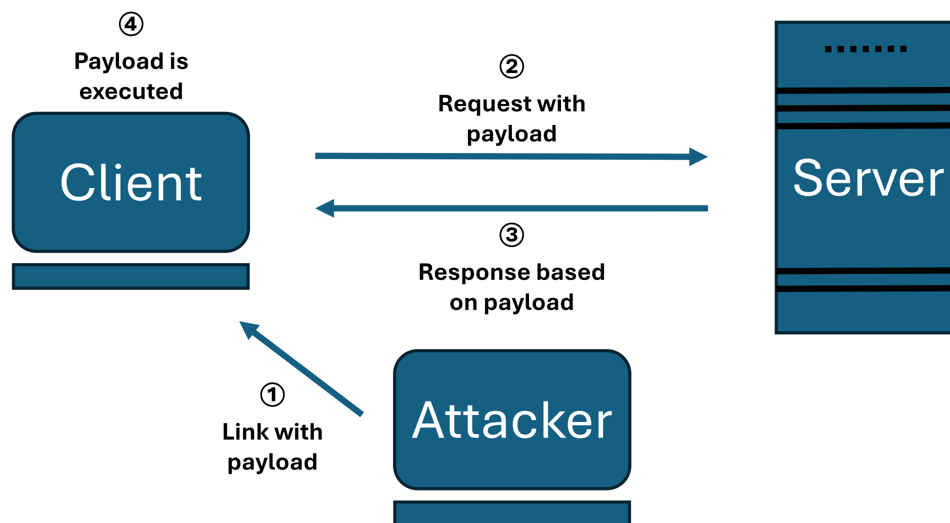


Figure 2.3: Illustration of the steps and interactions between the attacker, client and server in a reflected XSS attack. The numbers indicate the order generic actions would be performed.

content. This malicious script is then executed, since it is supplied by the trusted website. Stored XSS is sometimes referred to as persistent XSS [24]. The typical flow of a stored XSS attack is shown in Figure 2.4.

2.4.3 DOM-Based XSS

DOM-based XSS is a client side XSS attack. It is based on maliciously manipulating clients DOM environment, which then causes the browser to load a website in an unexpected manner. For instance, the DOM default language could be replaced with a payload that then could be executed when the browser attempts to load a website in the users preferred language. The malicious code in a DOM-based XSS attack is thus only present on the client [25]. The typical flow of a DOM-based XSS attack is shown in Figure 2.5.

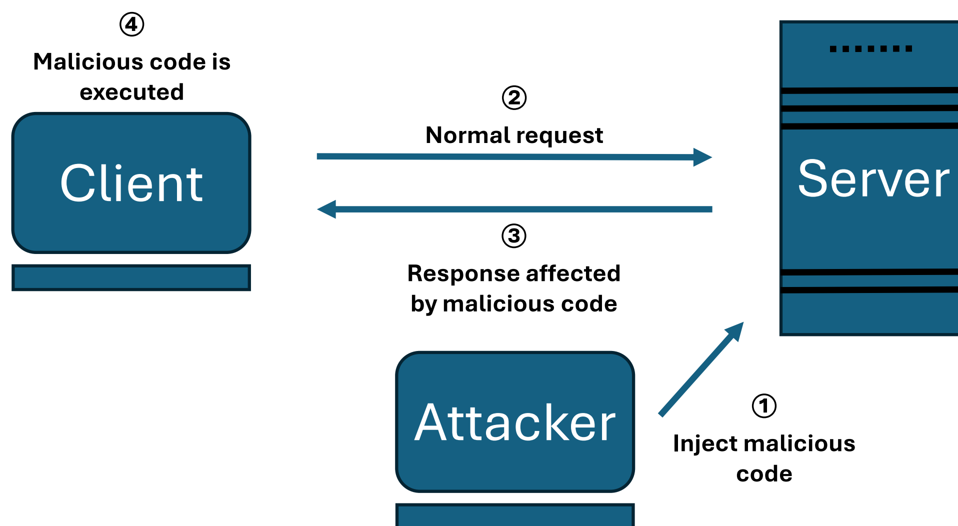


Figure 2.4: Illustration of the steps and interactions between the attacker, client and server in a stored XSS attacks. The numbers indicate the order generic actions would be performed.

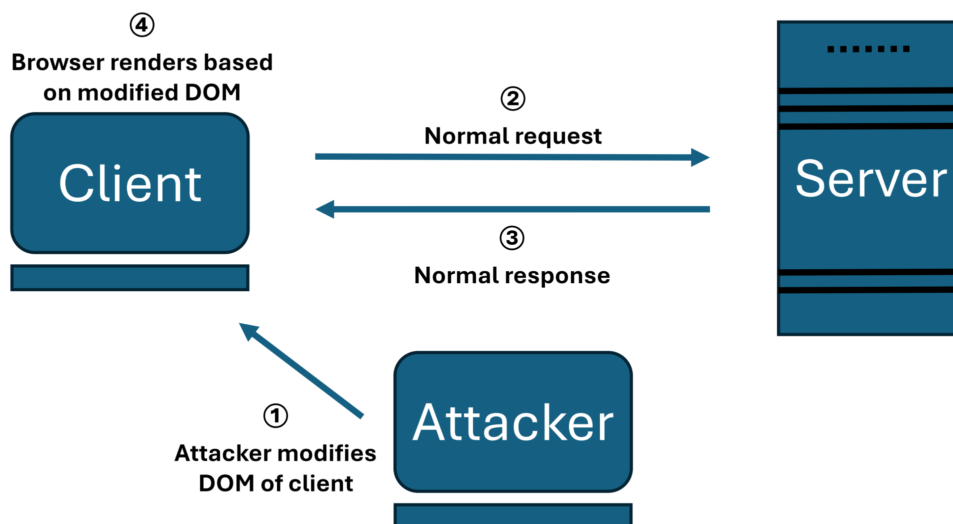


Figure 2.5: Illustration of the steps and interactions between the attacker, client and server in a DOM-based XSS attacks. The numbers indicate the order generic actions would be performed.

2.5 Vulnerability Scanning

Vulnerability scanning is the use of automated tools on websites to find vulnerabilities. They are usually divided into two branches of scanning. White box and black box. Figure 2.6 shows the work flow of black and white box scanners.

2.5.1 Black box Scanning

Black box scanning, also known as dynamic analysis, is a scan done without access to the source code. The scanner sends requests and interacts with the website in a similar fashion to a regular user. It does however often send thousands of requests and sends payloads which would trigger a specific vulnerability, if present.

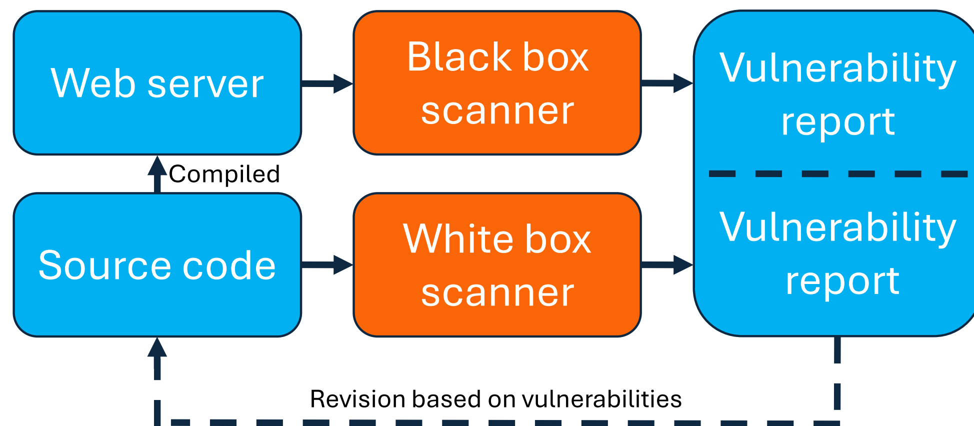


Figure 2.6: Chart showcasing the work flow of black and white box security scanners, and how they are used in the development process.

The primary advantage of black box scanning is that it can be deployed to evaluate any website, though the extent can vary. This website can either be a live version or a sandbox created from the source code. Either version selected should only be deployed with the permission of the owner of the website. The possibility to use these types of scanners comes from the fact that they work on the rendered files fetched by users of the website, then some traversal is performed. One method for this traversal is a random walk across the website with the goal of visiting all pages and interacting with all input fields. This universal approach has a downside in that it navigates without knowledge about the site and can potentially get stuck in loops. Traversal in high interactive websites exacerbate the problem since page creation and deletion without restrictions on the traversal can either make the application domain infinitely large or trivially small respectively, before any useful testing is done. Additionally good insight into the attack payloads is required, both of what makes a payload useful and which vulnerabilities they are associated with.

ZAP is maintained by the ZAP development team and is a black box security scanner. According to their own website, ZAP is the worlds most widely used security scanner[26]. Many other papers on the topic of vulnerability scanning include ZAP in their investigation, which strengthens their claims. ZAP launches automated attacks against a provided URL and generates a report based on the findings from the scan. It provides information about possible XSS attacks by attempting to inject payloads as well as alerting the developer about information leaks and missing headers. The reports generated by ZAP can be saved in a multitude of human readable formats, e.g. PDF, which was selected for the report example shown in Appendix B.

Arachni is a FOSS black box scanner and it had its latest and last release on the 29th of May 2024. It has been replaced by Codename SCNR which is neither free nor open source [27]. However, the latest release still works. Arachni is mainly a command line tool, though it can be set up as a local web application for the developer to use. Arachni, as other black box scanners, needs to be passed a URL to scan. Though unlike ZAP it does not terminate on its own - the user is required to set a timeout. Arachni also provides a high degree of customisation in its scans, where the user can specify what type of vulnerability to search for. It provides reports in a binary file specific to Arachni, which the user can open and read in Arachni. The user also has the option to save the reports in a human readable text format. For example HTML that then can be rendered as a PDF. Saving to HTML and then rendered to PDF was used to create the example Arachni report shown in Appendix C.

2.5.2 White box Scanning

White box scanning, also known as static analysis, is done with access to the source code, and thus usually does not have access to the running website. The white box scanner analyses the source code and usually builds some sort of database, syntax tree, or similar to then search the data for select patterns or code snippets which could be unsafe. This scanning method has the main advantage of being deterministic, meaning it does not have any variance between runs. One main obstacle here is the needed insight and knowledge of the system. Either the tool used or user of the tool needs to know about vulnerable functions and what data flows can expose the application to attacks. Examples of a white box scanner is CodeQL, Esprima and Semgrep [28–30].

CodeQL is a white box scanning tool developed by GitHub. It treats the code as data by generating a database from the codebase. Then it uses queries which find patterns in the database. These patterns can be vulnerabilities, errors, bugs, or any code which the user wants to find. If CodeQL gets a match in the database when running a query, it will give a code scanning alert. An alert is a result showing the location in the code where the query found the match, as well as any relevant data flow or control flow path used to get the result. CodeQL supports several programming languages. For this thesis, the most important ones are JS and TS. There exists a set of standard queries provided by GitHub, which do a wide variety

of checks. The users can however write their own queries, which are similar in style to SQL queries. CodeQL can both be ran locally on the developers device, or be configured to run as an action in GitHub. The GitHub alternative can be configured to run the security check when the repository is updated and/or during regular intervals [28, 31].

Esprima is another white box scanner, intended to perform lexical or syntactic analysis on JS code (with experimental support for JavaScript XML - JSX files). The tool has both a web version as well as a downloaded version to run locally [29].

Semgrep is the final white box scanner used in this thesis. It supports over 30 languages, although support for some of the languages is still in an experimental phase. It is a command line tool with a companion website which the user can connect to. This website provides a better user interface and some extra support. Semgrep finds vulnerabilities by matching them to rules. The rules designed reassemble the conventional code the developer is familiar with. These rules are fully customisable, allowing for expansion or alteration to better match the developers needs [30].

3 Method

The method used in this master thesis consisted of four different parts to cover the different aspects contained in the project. These were test environment creation, vulnerability detection, tool extension, and evaluation. In addition to these steps we initiated the work by analysing related papers to our topic to ensure novelty and utilise findings already made. An initial selection of frameworks and security scanners was decided upon, and expanded on as work carried on.

3.1 Test Environment Creation

In order to both verify and find vulnerabilities, several test environments were created. It was necessary for us to create at least one in each framework that was to be evaluated. The test environments were ideally created from the tutorials provided by the framework maintainer when available. In the cases where the maintainer had no or low function examples we looked for popular example websites on GitHub. For Remix we followed the standard tutorial [32], and cloned the real-time application example from their GitHub [33]. To create the NextJS website we made use of their dashboard app tutorial [34]. JQuery had no equivalent tutorial or example. Thus a single-page application containing a comment section had to be created. For detailed information about the websites see Section 4.2. For the list of selected frameworks see Table 3.1.

After the set-up of the test environments, we checked them for inherent XSS vulnerabilities. The found vulnerabilities were noted, and if none or only low impact ones were found, we modified the website to contain at least one vulnerability. This was done to, in later steps, be able to verify which vulnerability the security scanner was able to detect as well as which ones were undetectable by the scanners.

Selected Frameworks:
Remix
NextJS
JQuery

Table 3.1: Table listing the selected frameworks for this thesis.

3.2 Scanner Evaluation

A crucial step was to find and verify vulnerabilities, which either were inherent to the website or inserted by us. In order to determine the effectiveness of the vulnerability detection tools, we had to know what vulnerabilities were present. This was done in two primary ways. The first being manually reading the source code of the website, and identifying potential risk areas. The other was to probe the website with payloads, to see if any gave results. This was an iterative process, where the results (or lack thereof) were used in an attempt to figure out what the possible sanitisation or other defensive measures were in place.

The payloads used for manual probing were selected from a public list of common payloads and their variations. The list contains 6587 payloads [35], note that many of these are small variations to potentially bypass sanitisation. Out of these we selected characteristic variants and only tried its variants if it was necessary to sidestep a certain type of sanitisation. Regarding the fields tested we probed the different types of input fields present, i.e. each input field of a page but not those same fields on a duplicate page as this would be another instance of the recently probed field.

The vulnerability detection tools ZAP Proxy, Arachni, CodeQL, Esprima and Semgrep [26–30] were evaluated and tested on the different frameworks, they are also listed in Table 3.2. To avoid hardware and network differences affecting what the scanners found, all test were performed on the same local network and on the same hardware. To further avoid variations between runs, both the security scanners and the test environments were restarted between tests, to ensure all tests had the same starting conditions. Black box scanners can in theory keep scanning indefinitely by, for example, continuously adding new comments. Specifically in Arachni a timeout was placed at two hours, after which the scan ended. White box scanners on the other hand have a constrained time frame they require and unless the source code changes this stays the same. The time required relates to the hardware available and size of the codebase, faster hardware will decrease it and a larger codebase will slow it down.

The black box scanners generated reports at completion. Unlike the white box scanners, which displayed the results in command line output or inside the code editors. It is worth mentioning that since the focus of this thesis is on XSS, results reported about other kinds of vulnerabilities were ignored for instances other than evaluating the consistency of black box scanners. In the case of Arachni, CodeQL, and Semgrep, it was possible to limit the scan to only consider XSS vulnerabilities.

To evaluate the two black box scanners (Arachni and ZAP) ten scans each on the same website were run to evaluate the variance and better identify its detection capabilities. For this, the test environment remix-contacts was selected. Since it contain several different XSS vulnerabilities as well as functionality for user creation of new pages. This gave enough variability to examine the random walk property of the black box scanners. The test environment was reset after each test, to ensure

Selected Security Scanners:
Arachni
ZAP
Esprima
Semgrep
CodeQL

Table 3.2: Table listing the selected security scanners for this thesis.

that there were no initial differences between tests. From these tests, the average number of vulnerabilities of each type, the rate of false positives and their respective standard deviation was calculated. All XSS vulnerability reports on code which was not vulnerable with the proof provided by the scanner, or by manual attack attempts were counted as false positives. The rate of false positives was calculated as $FalsePositiveRate = \frac{FalsePositives}{FalsePositives+TruePositives}$.

To evaluate the white box scanners we ran the existing queries relevant to XSS. Due to the lack of random variance of white box scanners multiple scans was not necessary. In the evaluation of both types of scans the present vulnerabilities were noted along with the vulnerabilities reported by the scanners.

3.3 Tool Extension

Following the evaluation, CodeQL queries were constructed to cover vulnerabilities not consistently detected by the other scanners. Two vulnerabilities were chosen as targets of our extension: `setInnerHtml`, and `a-tag`. These two were selected with the reasoning: they both lack consistent identification by other scanners, and both of them have legitimate uses that warrant caution. Thus highlighting these functions to be assessed and potentially changed would be beneficial.

For each of the selected vulnerabilities a query was created. This process was slightly different for each query, but in general they work by defining patterns of the vulnerability and returning any matches in the given code project. In the case of the thesis the known vulnerable functions that alter the website were part of the pattern. Part of this pattern were also a step to see if the functions were using any sanitisation or user controlled inputs. When all patterns were fulfilled the query returned the location(s) in the code that potentially exposed the website to XSS attacks. Each query was language specific and the ones created in this thesis were made to identify vulnerabilities in Remix. Lastly, a matrix was constructed to compile and highlight the found differences between the different scanners and the extension queries.

4 Results

In this section we present the results of the thesis along with brief descriptions. All of the results and generated report results are available in the GitHub repository [36].

4.1 Vulnerabilities Found

In this work, we found and worked with two main vulnerabilities. The first vulnerability was using the `setInnerHTML` tag, which some frameworks have renamed to `dangerouslySetInnerHTML` to highlight this risk. In the frameworks tested, this function has been vulnerable to stored XSS attacks. `SetInnerHTML` can be secured by only allowing certain types of input. The second vulnerability, which also is susceptible to stored XSS attacks, is based on the `a-tag`. More specifically we found that input passed to an `a-tag` was possible to inject with pure JS which then was executed when clicked. This only applied when a complete URL was expected as input. That is, if the input was appended to an already determined base URL, XSS was not possible. For example if "twitter.com/" was the base URL and the input from the user was their username. The username would be appended to the base and no vulnerability would be present. If however, the user is allowed input the entire string "twitter.com/username" by themselves, that is, the complete URL. Then the user could exploit the vulnerability.

4.2 Test Environments

For this thesis we wanted to place an emphasis on TS based frameworks. Remix and NextJS were therefore selected, they also provided the benefit of both being React based. Them having a similar foundation makes it easier to generalise a solution across the two. We also wanted a JS based framework, as this is most prevalent on the web, which we could use as a comparison. For this JQuery was chosen, based on its popularity within JS based websites [4]. In Remix we created two test environments, one was created in NextJS, and one in JQuery.

The first Remix page, **Remix Contacts**, was from the basic Remix tutorial. The site consists of a contact book, and image upload page and a comment section. It had no inherent vulnerabilities, thus we inserted both the vulnerabilities mentioned in Section 4.1. In this website we also inserted a sanitised instance of the `setInnerHTML` vulnerability. This was to check for false positives. Appendix D

showcases the parts of the website containing relevant components to the XSS vulnerabilities. The second Remix page, **Remix Real-time**, was a clone of a Remix example available on their GitHub. This page is a real-time to-do list. Without modifications the website included a stored XSS vulnerabilities originating from the use of the `setInnerHTML` attribute without sanitisation. It is available in Appendix E.

For the NextJS page we also followed a tutorial. **NextJS dashboard** is a page showing invoices and other company information. It contained no vulnerabilities, so just as for Remix contacts, they were inserted. NextJS Dashboards XSS vulnerabilities are showcased in Appendix F. The last test page was made in **JQuery**. In the case of this framework no suitable tutorials were found. We therefore created a page containing a comment field where the JQuery version of `setInnerHTML` (the function `.html()`) was used. This input box and the rendition of its input (that is the comment field) made the test page vulnerable to stored XSS attacks as user supplied text was inserted into the page without any safeguard. The website is shown in Appendix G.

4.3 Existing Analysis Tools

As mentioned in Section 3 we evaluated five different analysis tools during this thesis, three of them static and two of them dynamic. The static tools were Esprima, CodeQL and Semgrep. The web version of **Esprima** did not manage to parse the syntax of any of the tested frameworks, neither pure TypeScript file nor TypeScript XML files. Therefore gave no meaningful results. Since TS compiles into JS we tried running it on the compiled JS files, but this also gave syntax errors. When using **CodeQL**, we found that running the queries worked, and the applicable queries gave alerts. The inserted (or inherent) vulnerabilities were however not found, and CodeQL has no standard queries which finds the relevant vulnerabilities. Lastly we ran **Semgrep**, which had the best results of the three. In Remix and NextJS it managed to find some of the `setInnerHTML` vulnerabilities, though none of the `a-tag` vulnerabilities. In JQuery, it had no applicable rules to find the respective `setInnerHTML` vulnerability. Thus, it had no findings even though it was compatible.

The dynamic tools used in this paper were ZAP and Arachni. **ZAP** was found to be highly consistent in the quantitative tests. It always reported five XSS vulnerabilities, two of which were stored XSS and three of which were reflected XSS. In all of the tests, one of the reported reflected XSS vulnerabilities was a false positive. Specifically it had a false positive rate of $\frac{1}{3}$. Another note of importance is that all the reported (true) XSS vulnerabilities were the same instance of `setInnerHTML`. It thus failed to find both the other unsanitised instance of `setInnerHTML` and the `a-tag` vulnerability. This points to a gap in ZAP's capabilities and leaves the developer without help in detecting this vulnerability. An extract of a report generated by ZAP is available in Appendix B.

The second dynamic tool, **Arachni** managed to find some vulnerabilities though not

all present in the test page. Also of note is that it reported a significant amount of false positives, and the results differed slightly between runs. Exerts from Arachni generated reports are available in Appendix C. The results from the quantitative test show that there was an average of 23.4 total XSS vulnerabilities with a standard deviation of 2.6. Arachni had an average false positive rate for XSS vulnerabilities of 37%, which correlated to an average of 8.6 false positives per scan with a standard deviation of 1.7. The XSS-HTML-insert category contains XSS vulnerabilities arising from insertion in HTML element content. Arachni found an average of 9.5 XSS-HTML-insert vulnerabilities, with an average deviation of 2.1. The false positive rate for XSS-HTML-insert was 76% with an average of 7.2 and a standard deviation of 1.7. Figure 4.1 shows the summarised results of the quantitative test, while Figure 4.2 shows their averages and standard deviations. Figure 4.3 shows the summarised results for false positives in the Arachni quantitative test, and Figure 4.4 shows their averages, standard deviation and average rate of false positives. These numbers show that while Arachni finds a large amount of vulnerabilities, it also finds a large amount of false positives. A developer tasked with fixing the vulnerabilities reported by Arachni will therefore have to spend significant effort sorting out the actual vulnerabilities.

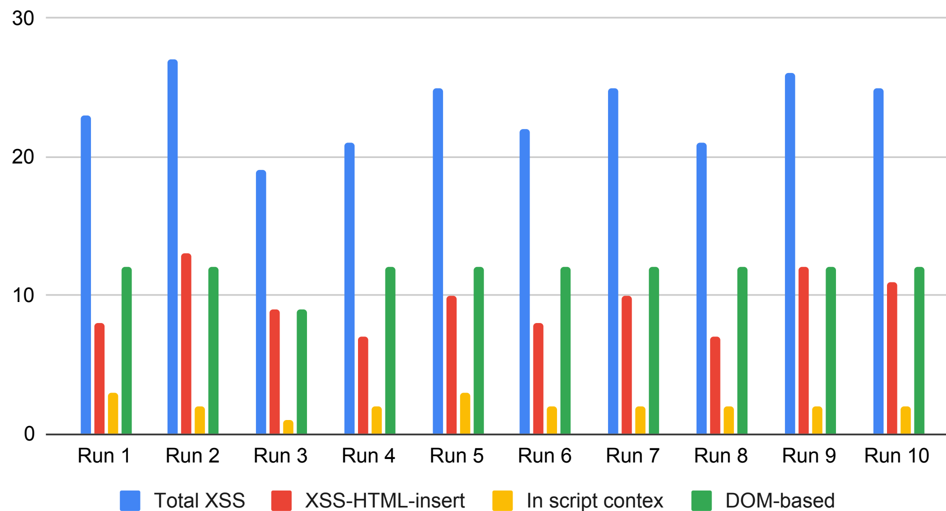


Figure 4.1: The result of the quantitative Arachni test. Showing the total amount of XSS vulnerabilities (*Total XSS*). The XSS vulnerabilities arising from insertion in HTML element content (*XSS-HTML-insert*). The XSS vulnerabilities where it was possible to force the page to execute JS code (*In script context*). Lastly the DOM-based XSS vulnerabilities, where JS code was executable by modifying the DOM (*DOM-based*).

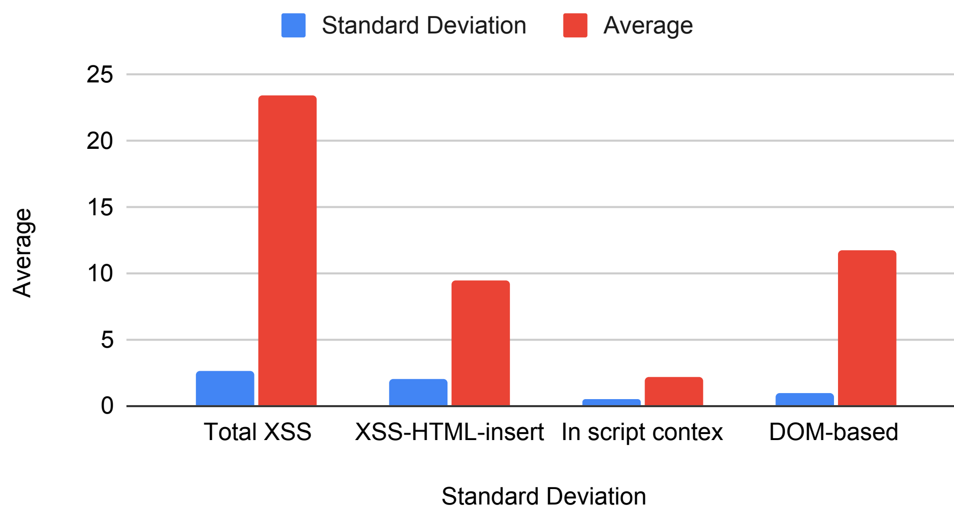


Figure 4.2: Average and standard deviation of the vulnerabilities found in the quantitative Arachni test presented in Figure 4.1.

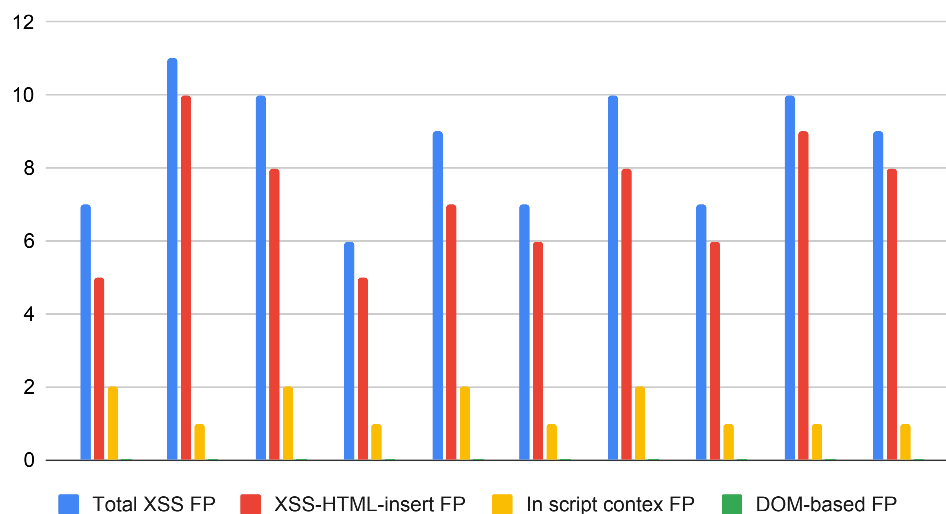


Figure 4.3: The amount of vulnerabilities verified as false positives in the quantitative Arachni test presented in Figure 4.1.

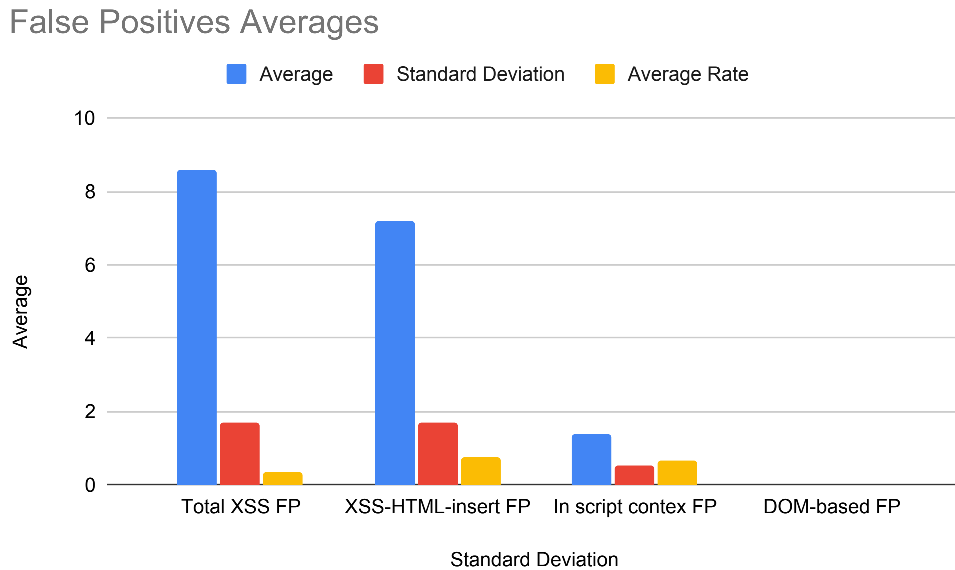


Figure 4.4: Average, average rate and standard deviation of the amount of false positives in the quantitative Arachni test presented in Figure 4.3.

4.4 Extended Queries

We extended CodeQL with three custom queries to cover the vulnerabilities discussed, **userControlled.ql**, **sanitisedDangerouslySetInnerHtml.ql**, and **unSanitisedDangerouslySetInnerHtml.ql**. The complete code is provided in Appendix A, as well as on the GitHub repository.

The query **userControlled.ql** works for instances of the `a-tag` vulnerability. It finds all instances of the elements corresponding to the `a-tag` that contain the attribute `href`. It also finds all input fields that assign the user supplied data to a variable. After these two steps it returns all cases where the `href` and input assignment work with the same variable. This means that the **userControlled.ql** query searches for, and returns, the uses of `a-tags` which links to user submitted URLs, or ones which are tainted by user input. These uses of `a-tag` were the ones who were found to be vulnerable.

sanitisedDangerouslySetInnerHtml.ql finds all instances of an element with the attribute `dangerouslySetInnerHtml`, which can allow for arbitrary code execution. It also finds what variables are passed to it and all variables which are assigned a value passed through a sanitize method call. The query then returns all instances of elements with the attribute `dangerouslySetInnerHtml` which uses a sanitised variable. The query **unSanitisedDangerouslySetInnerHtml.ql** has the same functionality, with the distinction that it returns the inverse. That is, all elements with the attribute `dangerouslySetInnerHtml` which uses an unsanitised variable. At first glance it might seem that only the **unSanitisedDangerously-**

SetInnerHtml.ql is of importance, since it can allow for arbitrary code execution of an unsanitised user provided variable. Having both is however important, since if the sanitisation is lacking then the sanitised instances are also at risk.

4.5 Summary

Table 4.1 presents the compatibility matrix derived from our testing. It shows which security scanner is compatible with which framework. It also shows what degree of findings was achieved. Two things especially of note is that CodeQL had no findings for neither Remix nor NextJS, while extended CodeQL achieved complete findings for both.

	Remix	**NextJS**	**JQuery**
ZAP	C,PF,FP	C,PF,FP	C,PF,FP
Arachni	C,PF,FP	C,PF,FP	C,PF,FP
Esprima	NC	NC	C
Semgrep	C,PF	C,PF	C,NF
CodeQL	C,NF	C,NF	NC
Extended CodeQL	C,CF	C,CF	NC

Table 4.1: Compatibility matrix of frameworks (X-Axis) and security scanners (Y-Axis). **Legend:** [C:] Compatible, [NC:] Non Compatible, [CF:] Complete Findings, [PF:] Partial Findings, [NF:] No Findings, [FP:] False Positives.

5
Discussion

In the following subsections we will discuss the results presented in the previous section. Further down we will discuss observations from working with the tools and issues we encountered during the process. Following that is a compilation of our thoughts and considerations made regarding the ethical aspects when working with computer security and the vulnerabilities found. Lastly we touch upon related works.

5.1 Website creation

When creating the websites to run the scanners on insight into the workings of the respective frameworks was needed, but also gained. This knowledge and insight was especially necessary when working with the white box scanners, both in understanding how they work (or in some cases did not work) and in creating the queries. Although as discussed the bulk of the understanding needed in our case was for CodeQL.

In the process of creating the websites many security related observations occurred. To name some of them, many recommend and use libraries which are flagged as vulnerable by Node. The names of functions with the same purpose vary, and some are better named to stress potential vulnerabilities than others. React based frameworks for example, has named the function to declare HTML from JS/TS as `dangerouslySetInnerHTML`. While in JQuery the standard function to assign HTML code is simply named `.html()`. Additionally, as mentioned before, one of the ready to go example pages for Remix contain a stored XSS vulnerability by default. This can lead to new or careless developers using or learning from this example, further propagating the unsafe use of `setInnerHTML`.

5.2 Analysis methods

As described in Section 2.5 there is merit to both branches of analysis though they also have distinct limitations. Here we will share our thoughts and observations from working with them. White box scanning lacks insight in the dynamic nature of websites as they work only on the source code and thus can not work with the potential interactions between the rendered pages. It is capable of highlighting data flows and indicating where sinks are but it does not definitively know where injected payloads will end up. Though with this constraint there arises a benefit not present

with black box scanners/dynamic analysis: there is a fixed size to the content worked on and therefore a fixed run-time of the tools. In contrast, black box scanners work by interacting with the website as it would be by a user. From this work flow arises the potential issue of looping through functionally identical pages or creating pages as it navigates, resulting in an infinite test environment through recursion. These tools therefore often run until stopped or by having some time limit defined when started. This creates the hard decision of deciding how long it needs to run. We set the timeout at two hours. This was mostly due to the relative small size of the test websites, and we found that after two hours it had explored most of the website several times. ZAP was able to determine independently when it was done, and its runtime was usually around 20 minutes depending on the targeted website. Arachni however, only finished once it was stopped or timed out.

In its testing of the website, the black box scanner tries multiple different attacks. The attacks include injections based on the main payload versions but also all the small variations that might defeat sanitisation of input to the website. Even in this regard it works in a mimicking fashion to how humans would, only with a significantly larger throughput of sent inputs. One issue we encountered when analysing the ZAP reports was that it often reused the same payload. This leads to issues of verifying if the recent vulnerability test ZAP performed worked, or if an old instance of the payload executed from another (actual) vulnerability. This was for example encountered in Remix Contacts, where ZAP injected and found a vulnerability in the index pages comment field. Then when it checked for vulnerabilities in the contact edit page, it deleted the contact which redirected the website to the index page, where the already found vulnerability was located and reactivated. Thereby tricking ZAP that sending the POST request to delete the contact containing the payload was also an XSS vulnerability and resulting in a false positive. This issue could be fixed by adding either some distinction to the payloads, or through incrementation or randomisation. Arachni implements randomisation in its payloads and while it also suffers from false positives, they are not caused by repeated payloads.

One important distinction between white and black box scanners is how they decide what a vulnerability is. Arachni and ZAP report a found vulnerability if they perform an action (inject JS for example) and then find proof of the action in the website. This proof is often that the inserted tag is present in the websites HTML though it does not take account for whether or not the websites displays the tag as an element or as a string. We considered the vulnerability as a false positive if the associated proof was an inserted tag which was treated as a string, since it does not lead to any execution of the tag. Arachni reported several instances of this type of false positives while ZAP only reported one. White box scanners on the other hand, report vulnerabilities when it finds specified code snippets or patterns in the source code. By nature this does not create false positives in the same manner as black box scanners do. If a query or rule is too broad it might find instances the developer does not consider a vulnerability, they however always find what the rule or query defines.

An important note regarding vulnerability scanning revolves around the relation

between ease of detection and the potential harm of the vulnerability. Specifically, the detection rate of the vulnerability with existing tools is essential. A less severe vulnerability which is hard to detect can be worse than a more severe vulnerability which is easy to detect. Imagine the case of a website breaking vulnerability that scanners catch with a 98% rate compared to a minor vulnerability that is only detected with a 2% rate. The former will exist on a fraction of live websites and probably only for a brief time when present. The later though might propagate through many websites and remain in place for extended times if not permanently.

5.3 Extended Queries

The extended CodeQL queries where proven to work in finding the a-tag and the `setInnerHtml` vulnerabilities for Remix and NextJS, though we have a few notes and thoughts about the extensions. First are some thoughts about generalization.

The queries as they work now are only suitable for Remix and NextJS. We speculate that they should work for other React based frameworks as well, due to their similarity (especially in structure). The nature of the solution, where child nodes of the inspected elements are continuously extracted to find the used variables causes it to be be highly dependant of the code structure. It is currently written to match a common pattern in Remix (which is inherited from React), though other frameworks, or other variations in Remix might not be caught. Thus the need to generalise the extended queries still remain. To what degree this is possible is not entirely clear, since there are a wide selection of frameworks (each with many possible patterns). While it might turn out to not be completely generalisable, it should be at least partly generalisable.

A second note is concerning strings in CodeQL, specifically strings longer than 19 characters. In CodeQL there exists, as in most programming languages, a toString() function to convert some other type, such as a node or a variable, into a more human readable one. Though CodeQL's implementations of toString() has a caveat. It shortens all strings longer that 19 characters into 19 characters. It does this by taking the seven first characters, then three dots, then the last seven characters. This was discovered by us when trying to extract a variable name from a longer pattern. While this was resolved by relying more on parent child relations in Remix, this could still be an issue if long variable names are used, and could then lead to false positives. An example CodeQL's shortening of strings, and how it can lead to false positives, is shown in Figure 5.1. SanitisedInputData would be equal to SanitisedOutputData and then complete a pattern match where none existed before using toString().

5.4 Ethical Aspects

Security related papers, including this one, often need to consider how the information posed will be used. It is a valid consideration to think that the methods, tools

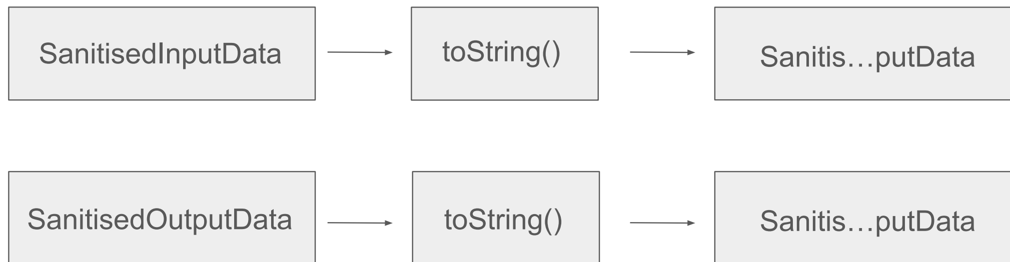


Figure 5.1: Example of string shortening when using toString() in CodeQL. Leftmost column are the variables names.

or security flaws described in the paper can be used not only to further secure and improve, but also by bad actors with malicious aims. This is not the goal of this or the other papers, but an almost unavoidable side effect. We argue however, that it is important for everyone to know what risks there are with the frameworks they are using, even if some may use that information maliciously.

The method proposed in this paper is intended to be used before the deployment to catch vulnerabilities before they are exposed to the general audience. It could be used by a malicious user to find vulnerabilities in web pages/sites where the source code is open source. We argue however that the method is still important and should be available, since it helps developers write safer code. The fact that the proposed method requires the source code significantly decreases the malicious use cases as compared to black box scanners, which can be aimed to any available page on the internet.

5.5 Typing and Framework Safety

TS has the potential strength to defuse most stored XSS if its types are used properly. Utilising the String/Int types (e.g. when handling user input) would safely handle any injected code as plain text, which at worst might cause a type error. In our opinion, a type error is strongly preferable to the alternative of mishandled user input. Currently this is possible in TS with typing as well as with specific functions, in the case of JQuery's `text()` function. The extra work this causes and the easier catch all functionality of `Any` and `.html()` does however allow developers to take the shortcut of sidestepping the use of more rigorous development in favor of faster and more familiar approaches. In both stated cases there are solutions that allow prevention of XSS vulnerabilities though with no warning or guarantee of these steps being taken. The risk of careless development exposes users to these vulnerabilities. As a middle ground or stepping stone we believe our extension and similar solutions that highlight the use of unsafe data paths gives developers a chance to catch these errors before they go live. This is somewhat comparable to how Rust, a programming

language with high memory safety, is restrictive when compared to other languages and will not let the code compile with careless memory management. Unless the developer deliberately decides to use the unsafe tag along with specific functions that bypass this compilation check.

5.6 Related Works

Several papers have previously added to, and investigated subjects related to our thesis. *XSS-SAFE: A Server-Side Approach to Detect and Mitigate Cross-Site Scripting (XSS) Attacks in JavaScript Code* by Shashank Gupta and B. B. Gupta [37], propose a server-side framework called XSS-SAFE. The framework found vulnerabilities through injection of benign functions. Then upon discovery it injects sanitisation into the effected JS, thus securing the application. They reported a false negative rate of 0% and a false positive rate of 10 - 15%, indicating the usefulness of XSS-SAFE. They also reported a low performance overhead, resulting in an increased response time of 1.25 - 5.75%. Their false positive rate was relatively small, especially compared to Arachni's measured false positive rate of 37%. Whether or not it would be able to find the a-tag and `setInnerHtml` vulnerabilities is uncertain since it was not tailored specifically in the manner as our extension of CodeQL. XSS-SAFE is however built to be an active defense mechanism, detecting and reacting to XSS attempts in a live environment. The extended CodeQL, and CodeQL in general, is in contrast built as a development tool and thus meant to run before a change is published.

Also on the topic of sanitisation and input validation is Gary Wassermann and Zhendong Su's paper *Static Detection of Cross-Site Scripting Vulnerabilities* [38]. They propose a combination of taint tracking and string analysis to track untrusted input and its sub-strings throughout the application. This allows the method to find vulnerabilities which might otherwise be missed. This paper has similarities to our thesis in that they also propose a static detection solution. Their main focus is on taint tracking, which while still an important step in our thesis, is not used to the same degree. For our CodeQL queries, we check the vulnerable elements input variables and whether they are user provided and/or sanitized. It is mostly an addition to decide on the seriousness of the vulnerability, whereas taint tracking is the focal point in the referenced paper.

On the topic of Server Side rendering, Oskar Lyxell investigates how it can improve the loading time of a website in the paper *Server-Side Rendering in React: When Does It Become Beneficial to Your Web Program?* [39]. Specifically the time from start of load until when a user is able to interact with the website. The main focus of the paper is not necessarily connected to our research but it is a useful resource to understand SSR better and showcase the utility of SSR to the web. Following the advantages presented in the paper it is likely we believe that an increasing share of websites will adopt this alternative to CSR, especially with the ease of deployment that frameworks like Remix supplies. In the predictive future of increased adoption, of the framework but particularly SSR, the risk of vulnerable websites will likely

come. This is the case both for websites converted from other frameworks as well as new websites made from scratch. Additionally and relevant to our paper there are some different behaviours that arise when a website utilises SSR, specifically alerts from stored XSS scripts that traditionally run immediately on submission. Though in our test environment payloads only acted as expected on a full page reload, we suspect this arises from the lazy loading feature in Remix that avoid re-calculation of compartments of the website and thus keep the payload dormant. This might in a way protect users from the vulnerabilities but also makes the vulnerabilities potentially harder to find as they behave abnormally. A black box scanner might for example not revisit the effected compartment and therefore not register the successful injection. It could also potentially connect the evidence to another injection, leading a developer on a wild goose chase. In this regard a white box scanner would be more suitable as it can potentially flag vulnerabilities regardless of optimisations related to the loading of the website.

Continuing the topic of scanner comparison, the paper *A Survey of Automated Tools for Probing Vulnerable Web Applications* done by Milad Barsomo investigates the effectiveness of black box scanners [40]. In the paper several, a majority, of them open-source, scanners were evaluated on their detection rate and false positive rate. Of the ones tested the author notes that ZAP, Arachni, and Vega performed the best. We evaluated two of these though of note is that between the papers release and the work on our paper Arachni has been discontinued and replaced by a commercial production and thus can not be expected to keep the same standard. Barsomo also notes that all scanners were evaluated on scans made with their default settings. Potentially this would adversely effect their performance, though as the author notes that the default setting is likely the one most developers would use when utilising the scanners. The paper concludes that there is a non-negligible variance between the scanners evaluated and that those early in development or unchanged for some time perform considerably worse. This highlights the need for continuous research and development within this field, consistent with much of security. Even further relevance to the conclusions drawn by us can be argued following this point. When scanners are applied to languages they were not made for (or techniques developed after their release), the performance is severely lacking. The observation presented in Barsomo's paper which can be connected to this is then that scanners in beta stages of development perform worse. The assumption then would be that beta stage scanners also lack the implementation insight. Specifically that all scanners require a considerable understanding of the framework or architecture to be investigated.

In *HXD: Hybrid XSS Detection by using a Headless Browser*, Hyunsang Choi, Seongjin Hong, Sanghyun Cho, and Young-Gab Kim present their tool Hybrid XSS detection (HXD)[41]. In the paper they propose a black box based detection method for XSS which is a combination of static string analysis and dynamic browser rendering. They present three main benefits of HXD as compared to other detection tools. The first benefit is the low amount of false positives, the second is that it does not require source code (since it is black box based), and the third being the automatic extraction of target URLs. The automatic URL extraction gets the URLs from web

logs (in their case web logs from the Korean major internet portal) which decreases the time needed for URL path discovery. While this paper focuses on black box based solutions, we focus our work on extending a white box based solution to avoid the variance and false positives present in black box scanners. Another argument as to why our focus is on white box scanners is to create a tool more suitable for frequent use during development, since black box scanners tend to require longer run time.

QL, the underlying language used in CodeQL, is presented by Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer in their paper *QL: Object-oriented Queries on Relational Data* [42]. In the paper they introduce the core concepts, its use cases, and effectiveness. They compare QL to Error Prone, which is a static analysis tool for Java. In the comparison they implement 101 Error Prone checks in QL. The paper shows that while Error Prone requires about 10,500 lines of code, while QL only requires slightly less than 2,000. The comparison also shows that when running the cases and queries, four Error Prone checks result in null pointer exceptions, while all QL queries return correctly. For the Error Prone cases that run correctly however, the average time is 97 seconds, while the average time for the QL queries is 201 seconds. The papers' main contribution is the QL language which is one of the foundations we require for our thesis. We build upon CodeQL, one of the implementations of QL, to further increase its usefulness. Specifically with queries aimed at XSS vulnerabilities as the included query packs to CodeQL are not made to target the vulnerabilities we implemented and examined.

6
Conclusion

Security is very important, and the developer currently shoulders a large part of the responsibility for it. Methods and tools, such as the one proposed in this thesis and the ones mentioned, have the ability to help the developer create safer code. The methods and tools are however only able to highlight existing or potential vulnerabilities, the user of these tools then has the responsibility to amend or report them.

In this thesis, we found that none of Arachni, ZAP, Semgrep, Esprima and CodeQL managed to find the a-tag and `setInnerHtml` vulnerabilities at a satisfactory rate. In addition Arachni, which out of the tested scanners had the greatest success of finding vulnerabilities, reported false positives at a rate of 37%. With the extension proposed in this thesis, CodeQL was strengthened to find all a-tag and `setInnerHtml` vulnerabilities, with no false positives.

6.1 Future Work

There are many directions the findings we have made could be taken. The main one and already hinted at by us would be to convert the queries we have made to apply to more languages. Specifically to match the structure of the framework in the cases where they are not built with node trees and thus lack the option of traversing with parent and child indication. Additionally it could be beneficial to make the queries work with flows, to catch cases where variables are assigned somewhere in the logic of the website.

Another avenue to explore would be to combine black and white box scanner into a "grey" box scanner. This combination could potentially provide stronger proof of vulnerabilities or avoid false positives in the cases where payloads appear to successfully effect the website, when in truth the payload is defused or even might not originate from the specific injection.

Bibliography

[1] A. Jiang, "The future (and the past) of the web is server side rendering," Feb. 2023. [Online]. Available: https://deno.com/blog/the-future-and-past-is-server-side-rendering (visited on 12/01/2023).

[2] OWASP. "Owasp top 10:2021." (n.d), [Online]. Available: https://owasp.org/Top10/ (visited on 02/14/2024).

[3] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 1193–1204, ISBN: 9781450324779. DOI: 10.1145/2508859.2516703. [Online]. Available: https://doi.org/10.1145/2508859.2516703.

[4] Statista. "Most used web frameworks among developers worldwide, as of 2023." (2023), [Online]. Available: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/ (visited on 02/14/2024).

[5] Meta. "React framework." (n.d), [Online]. Available: https://react.dev/ (visited on 02/14/2024).

[6] Shopify Inc. "Remix framework." (n.d), [Online]. Available: https://remix.run/ (visited on 02/14/2024).

[7] MDN Contribitors. "Javascript." (Sep. 25, 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 02/14/2024).

[8] MDN Contribitors. "Javascript technologies overview." (Nov. 20, 2023), [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview (visited on 02/14/2024).

[9] w3techs.com. "Usage statistics of javascript as client-side programming language on websites." (n,d), [Online]. Available: https://w3techs.com/technologies/details/cp-javascript (visited on 02/14/2024).

[10] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 177–187, ISBN: 9781450305624. DOI: 10.1145/2001420.2001442. [Online]. Available: https://doi.org/10.1145/2001420.2001442.

[11] G. Bernhardt. "Wat." (2012), [Online]. Available: https://www.destroyallsoftware.com/talks/wat.

[12] Microsoft. “Typescript documentation: The basics.” (Apr. 10, 2024), [Online]. Available: https://www.typescriptlang.org/docs/handbook/2/basic-types.html (visited on 04/10/2024).
[13] S. Overflow. “Stack overflow developer survey.” (2023), [Online]. Available: https://survey.stackoverflow.co/2023/#technology-most-popular-technologies (visited on 04/10/2024).
[14] MDN Contribitors. “Introduction to the server side.” (Jan. 1, 2024), [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction (visited on 02/14/2024).
[15] D. Abramov. “React v16.4.2: Server-side vulnerability fix.” (Aug. 1, 2018), [Online]. Available: https://legacy.reactjs.org/blog/2018/08/01/react-v-16-4-2.html (visited on 02/14/2024).
[16] VueJS. “Server-side rendering (ssr).” (2023), [Online]. Available: https://vuejs.org/guide/scaling-up/ssr.html (visited on 04/17/2024).
[17] Vercel Inc. “Client-side rendering (csr).” (2023), [Online]. Available: https://nextjs.org/docs/pages/building-your-application/rendering/client-side-rendering (visited on 04/17/2024).
[18] expressJS. “Express.” (2024), [Online]. Available: https://github.com/expressjs/express (visited on 04/02/2024).
[19] Shopify Inc. “Route configuration.” (n.d), [Online]. Available: https://remix.run/docs/en/main/discussion/routes#route-configuration (visited on 04/10/2024).
[20] Vercel Inc. “Introduction.” (n.d), [Online]. Available: https://nextjs.org/docs (visited on 04/15/2024).
[21] Shopify Inc. “Introduction, technical explanation.” (n.d), [Online]. Available: https://remix.run/docs/en/main/discussion/introduction#introduction-technical-explanation (visited on 04/15/2024).
[22] NodeJS. “Node.js about page.” (n.d), [Online]. Available: https://nodejs.org/en/about (visited on 05/08/2024).
[23] O. Foundation. “Jquery framework.” (n.d), [Online]. Available: https://jquery.com/ (visited on 05/08/2024).
[24] KirstenS, OWASP Contributors. “Cross site scripting (xss).” (n.d), [Online]. Available: https://owasp.org/www-community/attacks/xss/ (visited on 04/02/2024).
[25] OWASP. “Dom based xss.” (n.d), [Online]. Available: https://owasp.org/www-community/attacks/DOM_Based_XSS (visited on 04/11/2024).
[26] Z. D. Team. “Zed attack proxy (zap).” (n.d), [Online]. Available: https://www.zaproxy.org/ (visited on 05/06/2024).
[27] Sarosys LLC. “Arachni - web application security scanner framework.” (2023), [Online]. Available: https://github.com/Arachni/arachni (visited on 04/10/2024).
[28] GitHub. “About codeql.” (n.d), [Online]. Available: https://codeql.github.com/docs/codeql-overview/about-codeql/ (visited on 04/08/2024).
[29] A. Hidayat. “Ecmascript parsing infrastructure for multipurpose analysis.” (n.d), [Online]. Available: https://esprima.org/ (visited on 04/12/2024).

[30] Semgrep Inc. "Make shift left work." (2024), [Online]. Available: https://semgrep.dev/ (visited on 05/08/2024).

[31] GitHub. "About code scanning with codeql." (n.d), [Online]. Available: https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning-with-codeql (visited on 04/12/2024).

[32] Shopify Inc. "Remix tutorial." (2024), [Online]. Available: https://remix.run/docs/en/main/start/tutorial (visited on 05/29/2024).

[33] Shopify Inc. "Remix linear style realtime app web components." (2024), [Online]. Available: https://github.com/remix-run/examples/tree/main/_official-realtime-app (visited on 05/29/2024).

[34] Vercel Inc. "Learn next.js." (2024), [Online]. Available: https://nextjs.org/learn/dashboard-app (visited on 05/29/2024).

[35] *Payloadbox/xss-payload-list*, https://github.com/payloadbox/xss-payload-list. (visited on 06/03/2024).

[36] P. Antonsson and V. Carlson. "Evaluating-the-impact-of-server-side-rendering-on-security-and-stability-of-web-services." (2024), [Online]. Available: https://github.com/Phiant/SSR-XSS-Security (visited on 09/30/2024).

[37] S. Gupta and B. B. Gupta, "Xss-safe: A server-side approach to detect and mitigate cross-site scripting (xss) attacks in javascript code," *ARABIAN JOURNAL FOR SCIENCE AND ENGINEERING*, vol. 41, Oct. 2015. DOI: 10.1007/s13369-015-1891-7.

[38] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 171–180. DOI: 10.1145/1368088.1368112.

[39] O. Lyxell, "Server-side rendering in react: When does it become beneficial to your web program?" M.S. thesis, Umeå University, Department of Computing Science, 2023, p. 33.

[40] M. Barsomo, "A survey of automated tools for probing vulnerable web applications," M.S. thesis, Linköping University, Department of Computer and Information Science, 2017, p. 30.

[41] H. Choi, S. Hong, S. Cho, and Y.-G. Kim, "Hxd: Hybrid xss detection by using a headless browser," *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)*, pp. 1–4, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:4109951.

[42] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented Queries on Relational Data," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, S. Krishnamurthi and B. S. Lerner, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 56, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 2:1–2:25, ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.2. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2016.2.

A
Appendix 1

This appendix contains the extended CodeQL queries and brief explanations. For a more detailed explanation see Section 4.4 and Section 5.3.

A.1 userControlled.ql

This query, **userControlled**, handles the a-tag vulnerability. It matches variables used in *href* attributes to variables containing user input. If it finds that a user input variable is used as the *href* attribute in an a-tag it reports this as a vulnerability.

```
import javascript
import semmle.javascript.JSX

from JsxElement element, JsxElement source
where source.getName() = "input" and element.getName() = "a" and //
    ↪ source is input and is an a tag
  (source.getAnAttribute().getAChildExpr().toString() =
  element.getAttributeByName("href").getAChild().getAChildExpr().
      ↪ toString())
  //compares if input sets a variable and if a variable with the same
      ↪  name
  //then is used to set the href attribute of an a tag
select source, element, "<a> tag potentially controlled by user input
    ↪ ."
//print input and a tag location
```

A.2 sanitisedDangerouslySetInnerHtml.ql

This query, **santisedDangerouslySetInnerHtml**, find data flow between sanitised variables and the *dangerouslySetInnerHtml* attribute. This finds instances where arbitrary code execution is a potential risk if the sanitisation is incorrect.

```
import javascript
import semmle.javascript.dataflow.DataFlow

from DataFlow::MethodCallNode sanitize, Variable var, JsxElement elem
where sanitize.getMethodName() = "sanitize" //Finds instances of .
    ↪ sanitize
and var.getAnAssignedExpr() = sanitize.asExpr() //If a variable is
    ↪ assigned to the instance of .sanitize
and elem.getAttributeByName("dangerouslySetInnerHTML").getAChildExpr
    ↪ ().getAChild().getAChild().toString().regexpMatch(".*"+var.
    ↪ getName()+".*") //If the innermost child of a
    ↪ dangerouslySetInnerHtml element contains the variable


select elem, var.getName(), "If sanitazation of " + var.getName() + "
    ↪  is not correct this is an XSS vulnerability"
```

A.3 unSanitisedDangerouslySetInnerHtml.ql

This query, **unSantisedDangerouslySetInnerHtml**, find all instances of *dangerouslySetInnerHtml* where there is no data flow to a sanitised variables. This means that it renders an unsanitised variable, which is a vulnerability leading to arbitrary code executeion.

```
import javascript
import semmle.javascript.dataflow.DataFlow

from DataFlow::MethodCallNode sanitize, Variable var, JsxElement
    ↪ safeUse, JsxElement unsafeUse
where sanitize.getMethodName() = "sanitize" //Finds instances of .
    ↪ sanitize
and var.getAnAssignedExpr() = sanitize.asExpr()//If a variable is
    ↪ assigned to the instance of .sanitize
and safeUse.getAttributeByName("dangerouslySetInnerHTML").
    ↪ getAChildExpr().getAChild().getAChild().toString().regexpMatch
    ↪ (".*"+var.getName()+".*")//If the innermost child of a
    ↪ dangerouslySetInnerHtml element contains the variable
and unsafeUse.getAttributeByName("dangerouslySetInnerHTML") !=
    ↪ safeUse.getAttributeByName("dangerouslySetInnerHTML") //All
    ↪ other instances of dangerouslySetInnerHtml

select unsafeUse, "Unsanitized input, vulnerable to XSS"
```

B

Appendix 2

This appendix contains the first three pages of an auto generated ZAP scanning report. The entire report (114 pages) is available on the GitHub. In the alerts section on page one of the excerpt each unique vulnerability found is listed along with the number of instances found for each. For each unique vulnerability the instances where the scanner found them is listed along with its proof to recreate the vulnerability. Additionally a generic description of the vulnerability is provided along with generic steps that might prevent the vulnerability.

ZAP Scanning Report-P6

Site: http://10.250.3.109:3000

Generated on Thu, 30 May 2024 17:18:42

ZAP Version: 2.15.0

ZAP is supported by the Crash Override Open Source Fellowship

Summary of Alerts

Risk Level	Number of Alerts
High	2
Medium	3
Low	2
Informational	3

Alerts

Name	Risk Level	Number of Instances
Cross Site Scripting (Persistent)	High	2
Cross Site Scripting (Reflected)	High	3
Absence of Anti-CSRF Tokens	Medium	81
Content Security Policy (CSP) Header Not Set	Medium	31
Missing Anti-clickjacking Header	Medium	20
Application Error Disclosure	Low	4
X-Content-Type-Options Header Missing	Low	38
Information Disclosure - Suspicious Comments	Informational	59
Modern Web Application	Informational	11
User Agent Fuzzer	Informational	432

Alert Detail

High	Cross Site Scripting (Persistent)
	Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML /JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology. When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances

Description	which load content from the file system may execute code under the local machine zone allowing for system compromise. There are three types of Cross-site Scripting attacks: non-persistent, persistent and DOM-based. Non-persistent attacks and DOM-based attacks require a user to either visit a specially crafted link laced with malicious code, or visit a malicious web page containing a web form, which when posted to the vulnerable site, will mount the attack. Using a malicious form will oftentimes take place when the vulnerable resource only accepts HTTP POST requests. In such a case, the form can be submitted automatically, without the victim's knowledge (e.g. by using JavaScript). Upon clicking on the malicious link or submitting the malicious form, the XSS payload will get echoed back and will get interpreted by the user's browser and execute. Another technique to send almost arbitrary requests (GET and POST) is by using an embedded client, such as Adobe Flash. Persistent attacks occur when the malicious code is submitted to a web site where it's stored for a period of time. Examples of an attacker's favorite targets often include message board posts, web mail messages, and web chat software. The unsuspecting user is not required to interact with any additional site/link (e.g. an attacker site or a malicious link sent via email), just simply view the web page containing the code.
URL	http://10.250.3.109:3000/?q
Method	GET
Attack	<script>alert(1);</script>
Evidence	
Other Info	Source URL: http://10.250.3.109:3000/?index
URL	http://10.250.3.109:3000/?q
Method	GET
Attack	<script>alert(1);</script>
Evidence	
Other Info	Source URL: http://10.250.3.109:3000/?index&q
Instances	2
	Phase: Architecture and Design Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket. Phases: Implementation; Architecture and Design Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies. For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters. Consult the XSS Prevention Cheat Sheet for more details on the types of encoding and escaping that are needed. Phase: Architecture and Design

Solution	For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. Phase: Implementation For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping. To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set. Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use an allow list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a deny list). However, deny lists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.
Reference	https://owasp.org/www-community/attacks/xss/ https://cwe.mitre.org/data/definitions/79.html
CWE Id	79
WASC Id	8
Plugin Id	40014

High	**Cross Site Scripting (Reflected)**
	Cross-site Scripting (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML /JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology. When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust

C

Appendix 3

This appendix contains an Arachni report displayed as an HTML page. The binary file produced by Arachni is uploaded to a locally hosted web interface called ArachniWeb, after this step the uploaded scan can be viewed in the manner displayed here. Specifically two pdf prints of the web UI is displayed below: one of the report overview and one example of a detailed view for an issue.

C.1 Arachni Report Main Page

This is the main page, containing an overview of the scan results. Grouped into severity classes, in this case all are classed as high since they are all XSS vulnerabilities.

Arachni v1.6.1.3 - WebUI v0.6.1.1 (/)

Administrator

/ Scans / http://192.168.50.148:3000/

Arachni is heading towards obsolescence, try out its next-gen successor Ecsypno (https://www.ecsypno.com/) SCNR (https://ecsypno.com/scnr-documentation/)!

http://192.168.50.148:3000/

Imported from 'P1_192.168.50.148 2024-05-28 19_10_31 +0200.afr'.

Edit description

Share scan | Update schedule | Toggle comments

The scan completed in 02:00:13 .

Download the full report as HTML (/scans/10/report.html), JSON (/scans/10/report.json), Marshal (/scans/10/report.marshal), XML (/scans/10/report.xml) or YAML (/scans/10/report.yaml).

Issues [23]

Expand all | Collapse all

High severity [23]

- (/scans/10/issues/141) Cross-Site Scripting (XSS) in Form input 'notes'
- (/scans/10/issues/142) Cross-Site Scripting (XSS) in Form input 'last'
- (/scans/10/issues/152) Cross-Site Scripting (XSS) in Form input 'notes'
- (/scans/10/issues/153) Cross-Site Scripting (XSS) in Form input 'avatar'
- (/scans/10/issues/154) Cross-Site Scripting (XSS) in Form input 'last'
- (/scans/10/issues/155) Cross-Site Scripting (XSS) in Form input 'notes'
- (/scans/10/issues/157) Cross-Site Scripting (XSS) in Link input 'index'
- (/scans/10/issues/162) Cross-Site Scripting (XSS) in Form input 'comment'
- (/scans/10/issues/140) Cross-Site Scripting (XSS) in script context in Form input 'notes'
- (/scans/10/issues/156) Cross-Site Scripting (XSS) in script context in Link input 'index'

(/scans/10/issues/161)	Cross-Site Scripting (XSS) in script context in Form input 'comment'
(/scans/10/issues/143)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/144)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/145)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/146)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/147)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/148)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/149)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/150)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/151)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_form_dom input 'comment'
(/scans/10/issues/158)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_form_dom input 'comment'
(/scans/10/issues/159)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'
(/scans/10/issues/160)	DOM-based Cross-Site Scripting (XSS) in script context in Ui_input_dom input 'comment'

© 2013-2022 Ecsypno (http://www.ecsypno.com/)

C.2 Arachni Report Issue Detail View

This is the detailed view of a specific vulnerability. It contains the descriptions, details and proofs needed to recreate and verify the vulnerability.

Arachni v1.6.1.3 - WebUI v0.6.1.1 (/)

Administrator

/ Scans / http://192.168.50.148:3000/ (Created for imported scan #10 pr…

Cross-Site Scripting (XSS) in Form input 'last'

Arachni is heading towards obsolescence, try out its next-gen successor Ecsypno (https://www.ecsypno.com/) SCNR (https://ecsypno.com/scnr-documentation/)!

Cross-Site Scripting (XSS) in Form input 'last',

at http://192.168.50.148:3000/contacts/ztej322/edit (http://192.168.50.148:3000/contacts/ztej322/edit) – found by http://192.168.50.148:3000/ (Created for imported scan #10 profile) (/scans/10).

High severity

Client-side scripts are used extensively by modern web applications. They perform from simple functions (such as the formatting of text) up to full manipulation of client-side data and Operating System interaction.

Cross Site Scripting (XSS) allows clients to inject scripts into a request and have the server return the script to the client in the response. This occurs because the application is taking untrusted data (in this example, from the client) and reusing it without performing any validation or sanitisation.

If the injected script is returned immediately this is known as reflected XSS. If the injected script is stored by the server and returned to any client visiting the affected page, then this is known as persistent XSS (also stored XSS).

Arachni has discovered that it is possible to insert script content directly into HTML element content.

Awaiting review.

Technical details | Discussion [0] | Review | Timeline

Identification data

Injected seed	Signature	Proof
1<xss_09655385f5e6bcc9e0a5bf8481ced8c2/>	Not available	`<xss_09655385f5e6bcc9e0a5bf8481ced8c2/>`

Vector data

Type	URL	Inputs	
form	http://192.168.50.148:3000/contacts/ztej322/edit (http://192.168.50.148:3000/contacts/ztej322/edit)	**notes**	1
		first	1
		last	1<xss_09655385f5e6bcc9e0a5bf8481ced8c2/ >
		twitter	1
		avatar	1

HTTP data

Request

```
GET /contacts/ztej322 HTTP/1.1
Host: 192.168.50.148:3000
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Gecko) Arachni/v1.6.1.3
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.8,he;q=0.6
X-Arachni-Scan-Seed: 09655385f5e6bcc9e0a5bf8481ced8c2

notes=1&first=1&last=1%3Cxss_09655385f5e6bcc9e0a5bf8481ced8c2%2F%3E&twitter=1&avatar=1
```

Response

Proof is highlighted in red and scroll-centered.

```
HTTP/1.1 302 Found
location: /contacts/ztej322
Date: Tue, 28 May 2024 15:57:44 GMT
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked

HTTP/1.1 200 OK
content-type: text/html; charset=utf-8
Vary: Accept-Encoding
Content-Encoding: gzip
Date: Tue, 28 May 2024 15:57:45 GMT
Connection: keep-alive
Keep-Alive: timeout=5
Transfer-Encoding: chunked

<!DOCTYPE html><html lang="en"><head><meta charSet="utf-8"/><meta name="viewport" content
="width=device-width, initial-scale=1"/><link rel="stylesheet" href="/build/_assets/app-V
3UKCKY4.css"/></head><body><div id="sidebar"><h1>Remix Contacts</h1><div><form method="ge
t" action="/" id="search-form" role="search"><input id="q" aria-label="Search contacts" c
lass="" placeholder="Search" type="search" name="q" value=""/><div id="search-spinner" hi
```

© 2013-2022 Ecsypno (http://www.ecsypno.com/)

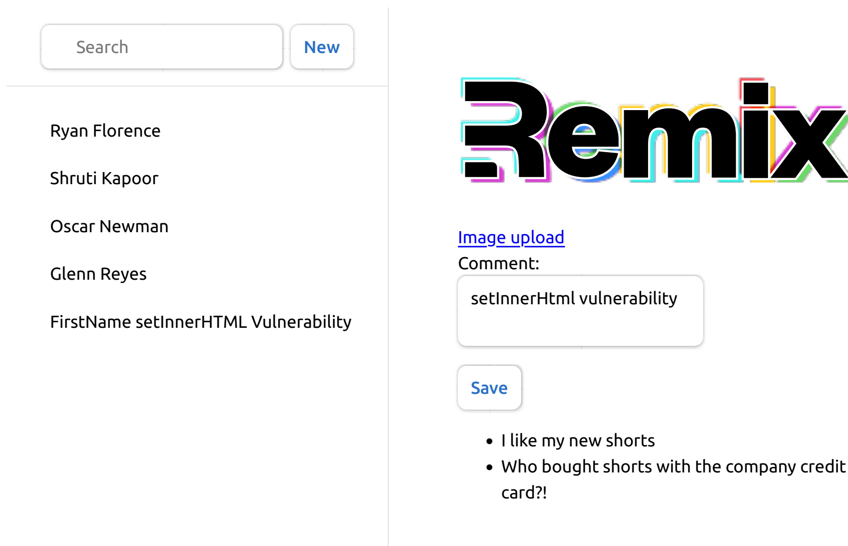
D

Appendix 4

This appendix contains screenshots from the test environment **Remix Contacts**. The vulnerable input fields are noted with text.

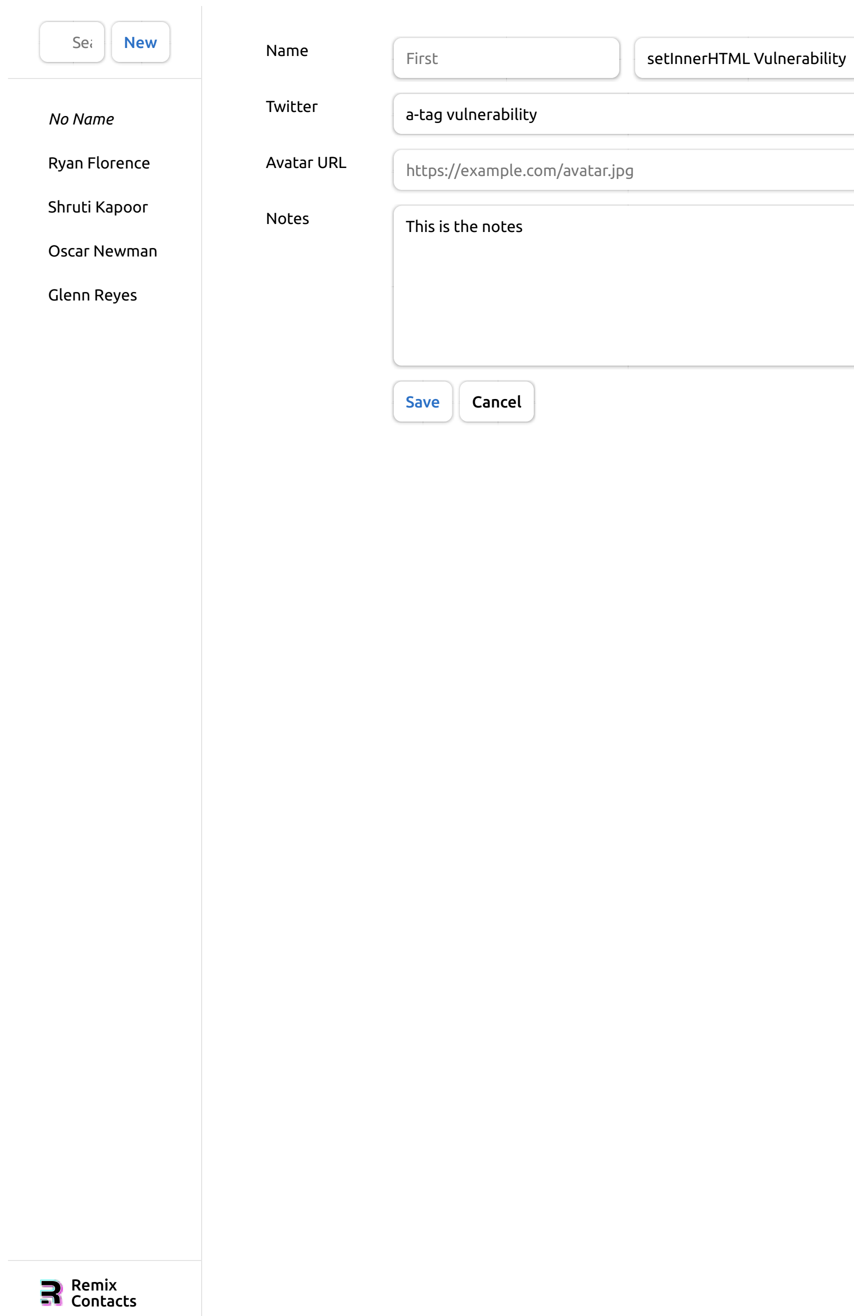
D.1 Remix Contacts: Main Page

This is the main page. It contains a comment field which has a **dangerously-SetInnerHtml** vulnerability.



D.2 Remix Contacts: Contact Create Page

This is the contact creation and edit page. No vulnerabilities are present on this page, though the input provided here are displayed in vulnerabilities on the contact pages.



D.3 Remix Contacts: Contact Page

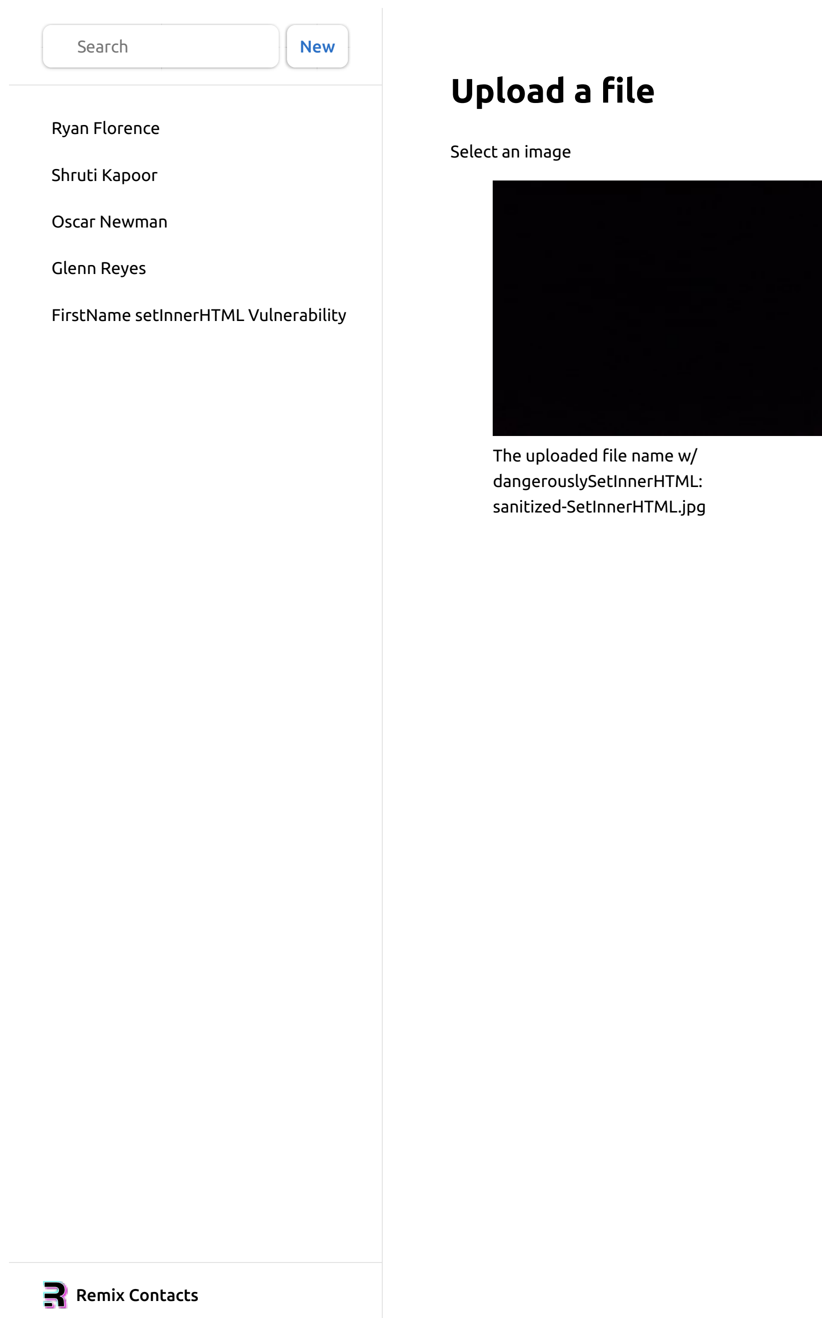
This is a page showing one contact. Here the input from the edit/creation page is displayed in both an a-tag vulnerability and in an **dangerouslySetInnerHtml** vulnerability.

Search
New
Ryan Florence
Shruti Kapoor
Oscar Newman
Glenn Reyes
FirstName setInnerHTML Vulnerability
Remix Contacts

FirstName setInnerHTML setInnerHTML Vulnerability Vulnerability ☆
a-tag vulnerability
This is the notes
Edit
Delete

D.4 Remix Contacts Image Upload Page

This is a page allowing for image uploads. The image name is sanitised then passed on to a **dangerouslySetInnerHtml** attribute. This page is made to test for false positives, it is not vulnerable.



E

Appendix 5

This appendix contains screenshots from the test environment **Remix Real time App**. The vulnerable fields are noted with text.

E.1 Remix Real Time App: Post Edit

This is the page where posts are edited and displayed in detail. This is where the dangersoulySetInnerHtml vulnerability is present and where the input for it is submitted. They exist on the same page due to the Real Time App nature of the website. As soon as a successful payload is entered and the user stop using the edit box the user is exposed to the payload.

E.2 Remix Real Time App: Main Page

This is the main page of the website, showing an overview of all tasks. There is no vulnerability present on this page. Even though the input from the page shown above is visible here it is handled in a safe way, demonstrating the difference between sanitised and unsanitised input.

All issues

ID	Title	Date
REM-1	setInnerHTMLVulnerability	Oct 10
REM-2	Provide `matches` to loader/action/meta args [TODO: flesh th...	Oct 10
REM-3	ci - update RR workflow to comment again	Oct 10
REM-4	Write decision doc on replacing `@remix-run/dev` with `remix`	Oct 10
REM-5	Make a list of new docs/guides that we want to write	Oct 10
REM-6	React Router + Webpack migration guide using BYOC	Oct 10
REM-7	Fetcher API Updates Decision Doc	Oct 10
REM-8	Add `createCloudflareDurableObjectSessionStorage`	Oct 10
REM-9	Add CF workers ESM request handler	Oct 10
REM-10	bring React 18 fixtures/templates in line with indie-stack upda...	Oct 10
REM-11	Back navigation from ErrorBoundary (without `<Scripts>`) doe...	Oct 10
REM-12	An error in MetaFunction or LinkFunction is not caught by Erro...	Oct 10
REM-13	Fix rendered href for createHashRouter links	Oct 10
REM-14	Optional Route Segments Internal RFC	Oct 10
REM-15	bug: Respect basename in RR useFormAction	Oct 10
REM-16	UMD Build for remix-run/router	Oct 10
REM-17	Bug: Form action for pathless layout routes	Oct 10
REM-18	React Router: Optional Params	Oct 10
REM-19	RouterProvider unit test refactor	Oct 10
REM-20	Fetcher API updates Implementation	Oct 10
REM-21	Move dom utils from react-router to @remix-run/router	Oct 10

F

Appendix 6

This appendix contains screenshots from the from the test environment **NextJS Dashboard**. The vulnerable fields are noted with text.

F.1 NextJS Dashboard: Comment Publishing Page

This is a page to write posts for a comment section. This page contains no vulnerabilities, though the inputs provided here will be passed on to a vulnerable page.

Comments / Create Comment

Enter link then comment

a-tag vulnerability | setInnerHtml Vulnerability | Submit

lorem ipsum dolar sit amet

F.2 NextJS Dashboard: Main Page

This is the main page, it contains the comment field and has both the a-tag and the dangerouslySetInnerHtml vulnerabilities.

Create New Comment and Link

Click Here

Comments

setInnerHtml Vulnerability

User Links

a-tag vulnerability

G

Appendix 7

This appendix contains the screenshot from the from the test environment **JQuery Input**. The vulnerable fields are noted with text.

G.1 JQuery input: Main Page

This is the main page of the JQuery test environment. It contains an input box and the setInnerHtml vulnerability. The input from the box is passed to the vulnerability on submit.

jQuery+RequireJS Sample Page

Look at source or inspect the DOM to see how it works.

Input please

setInnerHTML Vulnerability Submit
setInnerHTML Vulnerability

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se