Universität des Saarlandes
MI Fakultät für Mathematik und Informatik
Department of Computer Science

Bachelor Thesis

Do you Trust your Types? A Qualitative Study on the Usability of Trusted Types to Mitigate Client-Side XSS Vulnerabilities

submitted by

Philipp Baus

on July 15, 2022

Reviewers

Dr. Ben Stock

Dr. Katharina Krombholz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, July 15, 2022,

(Philipp Baus)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, July 15, 2022,

(Philipp Baus)

Abstract

Cross-site scripting (XSS) is a web vulnerability that allows attackers to execute arbitrary JavaScript code in a victim's browser. Although a lot of time has passed since the discovery in 1999, XSS is still a huge problem for websites on the internet nowadays. With the current trend of shifting the code of web applications to the client-side and the rising complexity of client-side code, the prevalence of client-side XSS vulnerabilities is also getting more severe.

To mitigate these vulnerabilities, Google recently introduced a new web API, called *Trusted Types*. Trusted Types eliminate the root causes of client-side XSS vulnerabilities by locking dangerous DOM and JavaScript API functions to only allow input in the form of a Trusted Types object. However, from the top websites at the time only Facebook and Google are actively using Trusted Types to protect their services against client-side XSS vulnerabilities.

Therefore, this thesis aims to find common roadblocks for web developers when it comes to the implementation and the understanding of Trusted Types. To achieve this goal, we conducted a qualitative study on the usability of Trusted Types for web developers.

Acknowledgements

I would like to express my deepest appreciation to my advisor Sebastian Roth who continously supported me during the writing of this thesis. I am very thankful for his reviews on the early versions of this thesis and the weekly meetings in which he always tried to answer my questions.

Secondly, I am grateful to all participants that took part in the user study as this thesis would not have been possible without them.

I would also like to thank to my godfather Werner Recktenwald for his effort in proofreading this thesis. Finally, I would be remiss to not mentioning my parents, Vera Baus and Volker Baus, for supporting me in what I was doing since the beginning of my studies.

In the end, I also want to thank the reviewers Dr. Ben Stock and Dr. Katharina Krombholz for spending their time on reading and grading this thesis.

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Outline . . . 3

2 Background and Related Work **5**
2.1 HTML . . . 5
2.2 Document Object Model . . . 6
2.3 JavaScript . . . 7
2.4 Client-side XSS . . . 7
2.5 Trusted Types . . . 9
2.6 Qualitative Studies . . . 13

3 Methodology **15**
3.1 Problem Statement . . . 15
3.2 Research Goals . . . 16
3.3 Study Structure . . . 17
3.3.1 Interview Session . . . 18
3.3.2 Coding Task . . . 20
3.3.3 Transcripts . . . 27
3.3.4 Open Coding . . . 28
3.3.5 Recruitment . . . 28

4 Findings **31**
4.1 Study Results . . . 31
4.1.1 Participant Demography . . . 32
4.1.2 Mental Model . . . 32
4.1.3 Perceptions and Discourages . . . 35
4.1.4 Deployment Strategies . . . 38
4.1.5 Roadblocks . . . 41
4.1.6 Improvements . . . 43
4.2 Trusted Types Implementation Guideline . . . 44
4.3 Limitations . . . 47

5 Conclusion and Future Work **49**
5.1 Conclusion 49
5.2 Future Work 49

A Interview Guideline **51**

B Codebook **57**

C Recruitment Mail **65**

Abbreviations **65**

List of Figures **69**

List of Tables **71**

Bibliography **75**

Chapter 1

Introduction

During the early days of the internet web developers were only able to create static websites. As a result, it was difficult to change the content of a website without redirecting to another page. This changed in 1995, when the JavaScript language [1] was adopted by web browsers. Using JavaScript, developers may now dynamically add, remove, or update the content of a website. Therefore, JavaScript provided web developers with a plethora of new design options. Unfortunately, it also introduced a new type of web vulnerability known as *Cross-Site Scripting* or short *XSS*, which was first identified in 1999 by Microsoft [2].

An attacker can use XSS flaws to insert arbitrary JavaScript code into a website. The user's browser will then run the injected code in the page's context. XSS vulnerabilities can further be classified into two kinds. We're talking about a server-side XSS vulnerability if the code injection occurs on the server. Instead, if the injection occurs with the help of JavaScript in the user's browser, we're dealing with a client-side XSS vulnerability. Over the years JavaScript got more and more popular what led to a rising number of statements in external JavaScript files that are also often loaded from remote domains [3]. Therefore, the general complexity of JavaScript code has risen, resulting in an increase of XSS vulnerabilities on the client side that are now on the verge of being the most common kind of XSS vulnerability [4].

Stamm et al. created a new browser security feature called *Content Security Policy* or short *CSP* to mitigate XSS vulnerabilities [5]. Developers can use it to define a policy that limits the execution of scripts and, as a result, the execution of injected code. However, it was discovered that the majority of the policies on the web were insecure, often even trivially bypassable and hence did not adequately guard against XSS vulnerabilities [6–10]. Developers for example often used the *unsafe-eval* CSP directive that allows the usage of *eval* and for this reason leaves the web page vulnerable to possible client-side

XSS vulnerabilities. On top of that, CSP only allows to completely allow or disallow the *eval* function meaning that a developer either has to lose the functionality or the protection against XSS vulnerabilities. Additionally, it is also possible to bypass a secure CSP with JSONP endpoints that are allowed in the CSP [7], script gadgets [11] and even open redirects [11]. Therefore, a new security mechanism was necessary to fully handle XSS vulnerabilities on the web, particularly client-side XSS vulnerabilities.

As a response, Google began developing a new browser API, called Trusted Types, to allow developers to secure their websites against client-side XSS. Trusted Types should, rather to other mitigation approaches, allow you to write XSS-free code instead of mitigating existing problems [12]. In advance to a CSP it also allows to have fine-grained control over the *eval* function. To achieve this, Trusted Types will only enable the assignment of safe Trusted Types objects to dangerous JavaScript functions that can lead to client-side XSS vulnerabilities.

Unfortunately, to the best of our knowledge, only *Facebook* [1] and *Google* [2] are already deploying Trusted Types to protect against client-side XSS vulnerabilities at this time. This brings up the question why this is the case and what can be done about it. To answer this question, we are looking for roadblocks of web developers in terms of not only understanding Trusted Types and client-side XSS but also in implementing the security mechanism. To attain this purpose, we will perform a qualitative study that is similar to the CSP study conducted by Roth et al. [8]. At the end of the study, we will also aim to provide an implementation guideline for developers to mitigate or circumvent the roadblocks of Trusted Types that we identified. However, due to the time limitations of a bachelor thesis we will only perform a prestudy that focusses on building a reliable guideline for possible follow-up studies.

1.1 Contributions

First of all, this thesis can help to improve the usage of Trusted Types on the web by raising awareness. But apart from that, this work might also be used to help improving the usability of Trusted Types for web developers in two different ways. The results obtained from the conducted study can be used to highlight problems that web developers encounter in the understanding and the implementation process of Trusted Types. These problems could then be used to change the Trusted Types standard in a way that it eliminates the understanding and implementation problems that web developers face. Secondly, we will also help developers to get rid of problems during the implementation

[1]`https://www.facebook.com/`
[2]`https://www.google.com/`

process by creating the implementation guideline that web developers can stick with to ensure that they are correctly implementing Trusted Types. By raising awareness of Trusted Types and by helping to improve the usability, the work also indirectly contributes to a better client-side XSS protection on the web as more web developers might deploy a secure Trusted Types implementation on their websites.

1.2 Outline

In Chapter 2 we will first discuss background information that is required to fully understand the topic of client-side XSS and Trusted Types. Furthermore, it also highlights the security implications of Trusted Types on web applications. In Chapter 3 we will proceed to explain the methodology of the user study that we conducted as part of the bachelor thesis. Following this chapter, we present our results in Chapter 4 together with their limitations. At the end, in Chapter 5, we finally evaluate the results that were collected during this thesis as well as future work that can be based on this thesis.

Chapter 2

Background and Related Work

In this chapter, we present the background information that is required to understand the Trusted Types security mechanism and the design of our study together with the related work in these topics. First, we go through the fundamentals of HTML and JavaScript. Following these sections, we will discuss how XSS vulnerabilities, particularly client-side XSS vulnerabilites, can occur in a web application. Furthermore, we will also explain how Trusted Types work and why they are able to guard against client-side XSS problems. At the end, we will also have a look at qualitative study designs.

2.1 HTML

HTML is the primary language used on the web to define the structure of a website and therefore builds the foundation of the current web.

HTML documents use the XML syntax to define the structure of a website. Therefore, HTML documents can contain different tags that are defined in the HTML standard [13]. Each tag can further contain attributes and attribute values to adjust the meaning of the tag. Every tag and attribute that is defined in the HTML standard has a specific functioning: The **<h1>**-tag for example marks the content of a tag as the heading/title of the website. However, the supported attributes can vary for different tags. The **alt**-attribute for instance is only supported in tags that are loading resources like images, videos or iframes. It assigns an alternative text to the tag, this text is then shown whenever the resource content can not be loaded.

HTML documents have the advantage of being very compatible with a wide range of devices. An HTML document can be viewed on both desktop computers, mobile devices and can even be presented with aural output by reading out the content of the website to the user. This allows developers to only develop one web application that is supported by every device and, therefore, is saving a lot of time and costs in the implementation process. But before an HTML document can be shown to the user, it first needs to be processed by the web browser. In this parsing process the browser will divide the different tags and create a data structure called the **Document Object Model** (DOM). In Section 2.2 we will have a deeper look into the concept of the DOM.

Over the course of time there were various HTML versions. Berners-Lee proposed and published the first version of HTML in 1993 [14]. The first version of HTML contained 18 tags that could only be used to create simple static websites containing text and links to other websites. Since then, the markup language has been updated several times. With HTML 4.0 a new big milestone was reached by adding the possibility to include scripts to HTML documents [15]. This allowed web developers to dynamically change the content of a website according to a user's actions by using scripts on the website. The scripting language that is supported by all popular browsers is called **JavaScript** and will be explained in Section 2.3. The current HTML version (HTML 5.0), however, will also be the last version as it is used as a living standard and will receive new features on a regular basis in the future [13].

2.2 Document Object Model

The Document Object Model (DOM) was defined by the W3C in 2000 as a "platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents with the goal of defining a programmatic interface for HTML documents" [16]. Web browsers implement this model and use it to store the structure of an HTML document. When an HTML document is loaded, the browser will parse the content and will then use this content to fill the DOM of the website. After that, the browser allows scripts that are contained on the website to access the DOM and change the content of the webpage. The data stored in the DOM is structured in a tree-like object where each node/object of the HTML document represents a node/object in the DOM tree [17]. However, the DOM does not only store content of the HTML document, it also stores data that is related to the current website like cookies or the URL [16].

2.3 JavaScript

The JavaScript language is a scripting language supported by most web browsers. In 1997 JavaScript became the ECMA-262 standard when it was developed for the Netscape 2 browser [18]. Multiple other browser developers followed the Netscape browser by implementing the JavaScript standard into their own browser in the following years. As a result, all popular browsers are supporting JavaScript nowadays. But JavaScript was and is, like HTML, still developed further to add new features in the future. The current version ES6 was released in 2015.

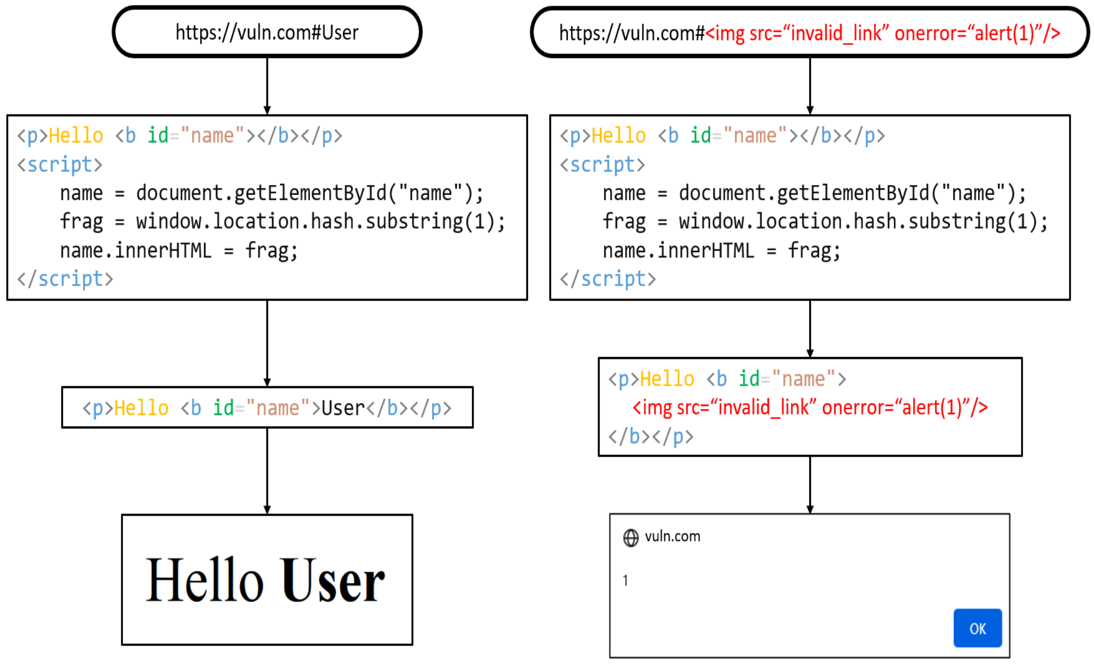
JavaScript enhances the user experience of websites by allowing developers to perform actions based on the user's interactions. This is possible as JavaScript code can change the content of a webpage by calling DOM API functions to modify the DOM of a website [19]. The *innerHTML* function can for example be called on a tag to change its content. Furthermore, it is also possible to change attribute values of specific tags. The *src* attribute of a script tag can be changed to control the URL where the script is loaded from. This is especially useful if the needed script is not known in beforehand and can therefore not be hardcoded in the HTML file. But there are also many other DOM API functions that can be used to change the DOM of a website to create powerful dynamic web applications. Besides the DOM API functions, JavaScript also provides APIs to execute JavaScript code directly from a string. Popular examples for such APIs are the functions: *eval*, *setTimeout* and *setInterval*. This enables the possibility for developers to dynamically generate and execute JavaScript code on the client-side.

But although these JavaScript functionalities create great usecases, they also have their downsides. Whenever untrusted user data is used in these APIs without sanitization, it leads to a vulnerability called client-side XSS. We will have a look at this type of vulnerability in Section 2.4.

2.4 Client-side XSS

Cross-Site Scripting is an attack in which an attacker is able to inject arbitrary JavaScript code into a web application [20]. However, XSS vulnerabilities can further be divided into two categories, server-side XSS and client-side XSS. In a client-side XSS vulnerability the code injection happens on the client-side in the user's browser through JavaScript code.

As mentioned before in Section 2.3, client-side XSS vulnerabilities occur when untrusted and unsanitized user data is used in different JavaScript APIs. These can be DOM APIs that allow to change the website content and JavaScript APIs that directly execute code


https://vuln.com#User
https://vuln.com#<img src="invalid_link" onerror="alert(1)"/>
Hello User
vuln.com
1
OK

```
<p>Hello <b id="name"></b></p>
<script>
    name = document.getElementById("name");
    frag = window.location.hash.substring(1);
    name.innerHTML = frag;
</script>
```

```
<p>Hello <b id="name">User</b></p>
```

```
<p>Hello <b id="name">
   <img src="invalid_link" onerror="alert(1)"/>
</b></p>
```

Figure 2.1: Client-Side XSS Vulnerability

from a string or set the source of a script tag. We will refer to these APIs in the following as "dangerous sinks" or "injection sinks". As a result, an attacker can craft a malicious payload that leads to the injection and execution of malicious JavaScript code in the user's browser when it is passed to a dangerous sink with insufficient sanitization. The malicious code injection can happen in three different ways depending on which sink is used for the injection:

- **HTML Sinks:** Malicious HTML code is written to the website eg. *innerHTML*.
- **JavaScript Sinks:** Malicious JavaScript code is executed from a string eg. *eval*.
- **ScriptURL Sinks:** The src attribute of a script tag is set to an URL containing a malicious script file.

Furthermore, we can also divide client-side XSS into two different categories according to the origin of the user data that is used in the dangerous sinks. If the user data is taken from a persistent storage, we speak of a persistent client-side XSS vulnerability. Persistent storages are for example the *localStorage* or the *cookie storage* of a website. In this case, the attacker would first need to infect the storage with a malicious payload that will trigger the XSS vulnerability when used in the dangerous sinks. However, once the victim's storage is infected, the victim will be attacked every time the vulnerable website is visited as the attacking payload is stored in a persistent storage. This is different for reflected client-side XSS attacks. In this case the user data is stored in the URL and reflected back to the victim. To attack a victim, we would therefore need to send an URL containing a malicious payload. When a victim is visiting the link, the malicious payload will trigger the XSS vulnerability and execute malicious code. But because of the fact that the malicious payload is contained in the URL the victim is only attacked if this specific URL is visited. An example for a reflected client-side XSS attack can be seen in Figure 2.1.

As the malicious JavaScript code that is injected by the attacker is also running in the context of the vulnerable webpage, the malicious code also has access to the storage API's of the website, like the *cookies API*. Therefore, an attacker could read confidential data from these storages and send it to its own endpoint. The data can then be used to hijack a victims's account or leaking private information about the user. Apart from that, there are even more possible attacks, a victim's actions on the vulnerable website could be recorded or the malicious code could even make requests to the target website without being blocked by CSRF [21] protection.

In 2019 Steffens et al. tried to study the threat of persistent client-side XSS to spot potentially malicious data flows to dangerous sinks [22]. After inspecting the top 5000 Alexa domains with a taint-tracking method, they discovered that 8% of these websites used unsanitized data in dangerous sinks, leaving the door open for a possible client-side XSS vulnerability. This shows that even the top websites on the internet are struggling to defend against client-side XSS vulnerabilities. For this reason, researchers tried to find possible ways to defend against these vulnerabilities and we saw multiple approaches in recent years [23–25]. The methods proposed in these works ranged from taint-tracking approaches to stop XSS payloads, over using a web proxy to identify and filter malicious JavaScript code, to sanitizing data before using them in dangerous sinks. However, none of these approaches tried to tackle the initial cause of client-side XSS vulnerabilities, the usage of untrusted data in dangerous sinks.

2.5 Trusted Types

To protect against client-side XSS vulnerabilities, a new browser security mechanism, called **Trusted Types**, was proposed by Google security engineers [26]. Trusted Types attempt to protect websites against client-side XSS vulnerabilities by making the dangerous sinks that lead to client-side XSS vulnerabilties secure by default. This is achieved by only allowing special Trusted Types objects to flow into these dangerous sinks. So, instead of trying to mitigate existing client-side XSS vulnerabilities, Trusted Types "direct developers to write XSS-free code in the first place" [4].

```
var defaultPolicy = trustedTypes.createPolicy('default', {
  createHTML: string => DOMPurify.sanitize(string),
});
```

Listing 2.1: Trusted Types Default Policy

When Trusted Types are enforced, developers are able to declare Trusted Type policies that are used to sanitize user data. To create a Trusted Types policy, the API function *createPolicy* must be called. The function must be given a unique name that identifies the policy and a JavaScript object. The JavaScript object can contain up to three sanitization functions with predefined names. These functions are used to create different Trusted Type objects, one for every type of injection sink [27]:

- **createHTML:** Returns a *TrustedHTML object* that can be safely used in sinks adding HTML code to the website.
- **createScript:** Returns a *TrustedScript object* that can be safely used in JavaScript sinks that might lead to the execution of a script from a string.
- **createScriptURL:** Returns a *TrustedScriptURL object* that can be safely used in a ScriptURL sink that will parse it as a URL of an external script resource.

It is the developer's task to ensure that a policy correctly sanitizes user data so that no malicious data can end up in a dangerous sink. If a policy is implemented insecurely, malicious data might still be able to flow into a sink. Once a Trusted Types policy is implemented, it can be used to create the correct Trusted Type object for a given sink by calling the coresponding function on the policy object.

There is also a special policy that can be implemented, the *default policy.* It is declared by assigning the name "default" to the policy and can be described as a fallback policy. An example implementation of a default policy can be seen in Listing 2.1. Whenever a pure string is passed to a sink, this policy will be implicitly called by the user agent to create the required Trusted Types object [27]. A default policy can be very useful, especially when third party libraries are included in the document. In this case the default policy is automatically used whenever the third party code calls a dangerous sink. This basically removes the need of rewriting third party code to explicitly create Trusted Types objects before assigning data to a dangerous sink.

To enforce Trusted Types developers must use the CSP. There are two different CSP directives that must be used to declare the enforcement rules for Trusted Types. The first directive **require-trusted-types-for** tells the browser to enforce Trusted Types for the current website by only allowing Trusted Type objects in dangerous sinks [28].

```
require-trusted-types-for 'script';
```

Listing 2.2: Trusted Types Enfocement

```
trusted-types default 'allow-duplicates';
```

Listing 2.3: Trusted Types Policy Restriction

The argument of the directive must always be set to *'script'*. The CSP directive can be seen in Listing 2.2. The second directive restricts the creation of Trusted Type policies [29]. The directive is called **trusted-types** and its arguments describe the names that may be assigned to a policy, all other names are disallowed. So, if the directive is absent, no policies may be created. There is also another possible argument, *'allow-duplicates'*. If this argument is set, it is allowed to recreate a policy after it was initially created. The directive can be seen in Listing 2.3. When Trusted Types are enforced and a pure string is passed to a sink without a default policy being defined, the browser will throw a PolicyViolation error. The same happens when the wrong Trusted Types object is assigned to a sink, for instance when a TrustedHTML object is assigned to a sink requiring a TrustedScript object and whenever a policy is created with a name that is not listed in the *trusted-types* CSP directive.

Although Trusted Types provide adequate protection against client-side XSS vulnerabilities, developers must exercise caution during the implementation process. First of all, Trusted Types are currently not supported by all popular web browsers. At this time, only browsers based on Chromium are supporting the CSP directives to enable Trusted Types. This means that Firefox as well as other popular non-Chromium web browsers are not supporting Trusted Types yet. Table 2.1 shows the compatibility of the different Trusted Types CSP directives for various popular browsers. The Trusted Types code, when executed, will thus generate an error on the web browsers that have not yet implemented the security feature. To address this issue, developers can use JavaScript code to determine if the user's web browser supports Trusted Types. This can be accomplished by verifying whether the *trustedTypes* variable is defined in the current window and whether it contains the function *createPolicy*. Whenever these conditions are not met, the Trusted Types code shall not be executed. Listing 2.4 illustrates one possible way to implement the condition checks. Secondly, web developers must ensure that their website is provided through HTTPS by the time it is operational. Even if the CSP directives are correctly configured, whenever the website is delivered through HTTP with an origin other than *localhost*, Trusted Types will not be available in the browser [30]. That means, if developers choose to run their website over HTTP, they cannot use the advantages of Trusted Types.

trusted-types		
Browser Name	Support	Version
Chrome	✓	**83**
Edge	✓	**83**
Firefox	×	×
Internet Explorer	×	×
Opera	✓	**69**
Safari	×	×
WebView Android	✓	**83**
Chrome Android	✓	**83**
Firefox Android	×	×
Opera Android	×	×
Safari IOS	×	×
Samsung Internet	✓	**13.0**

require-trusted-types-for		
Browser Name	Support	Version
Chrome	✓	**83**
Edge	✓	**83**
Firefox	×	×
Internet Explorer	×	×
Opera	✓	**69**
Safari	×	×
WebView Android	✓	**83**
Chrome Android	✓	**83**
Firefox Android	×	×
Opera Android	✓	**59**
Safari IOS	×	×
Samsung Internet	✓	**13.0**

Table 2.1: Trusted Types Browser Compatibility [28, 29]

```
if (window.trustedTypes && trustedTypes.createPolicy) {
    var escapePolicy = trustedTypes.createPolicy('default', {
      createHTML: string => string,
      createScript: string => string,
      createScriptURL: string => string
    });
}
```

Listing 2.4: Trusted Types Compatibility Check

In 2021 the Google employee Krzysztof Kotowicz released a report on the state of Trusted Types [31]. In the report the researcher anaylzed the state of Trusted Types in the year 2021. Therefore, Google applications and numerous libraries and frameworks were examined. During the examination of Google applications it was found that 61% of client-side XSS vulnerabilities that were not recognized by static analysis were prevented by the usage of Trusted Types. Also no client-side XSS vulnerability was reported after Trusted Types were enforced on these applications. This shows that Trusted Types are highly effective in defending against client-side XSS vulnerabilities. Furthermore, Trusted Types are also already supported in a few popular frameworks, like *Django* [32] and *Angular* [4]. In the same year, Wang et al. attempted to assess the effort involved in refactoring existing code bases to use Trusted Types [12]. To do so, they conducted a case study on the adoption of Trusted Types in the web framework *Angular*. The goal was to help other developers in the process of adopting Trusted Types to other frameworks by sharing the problems and solutions they encountered during the adoption process. At the end of their study, they fully adopted Angular to Trusted Types allowing web developers to make use of Trusted Types in their Angular projects. Furthermore, they also fixed a new Angular vulnerability, that they discovered during the adoption process.

2.6 Qualitative Studies

There are two different kinds of approaches that can be used to conduct a study, qualitative and quantitative approaches. These different approaches are used for different tasks. Whenever the goal is to confirm a specific hypothesis, a quantitative approach should be used. The empirical data collected throughout the quantitative study can be used to confirm whether a hypothesis holds or not [33]. However, in our case we have the exactly opposite case, we have no hypothesis that we would like to confirm, we would like to reveal new theories about the usability of Trusted Types for web developers. Qualitative approaches are often used to specify new hypotheses or theories in areas where not much is known [33]. Therefore, this approach is more suited for our study due to the small amount of information known about the usability of Trusted Types. One of the most popular methods to perform a qualitative study is conducting interviews with participants. There are two different interviewing techniques, structured interviews that follow a strict guideline, while unstructured interviews do not have a fixed set of questions and order. However, often a mixture of these interview technqiues is used, a so called semi-structured interview. A semi-structured interview can help to gain more insights into the concepts of the participants by asking follow-up questions on interesting concepts, while it also allows to still follow a specific question guideline. The technique was also used in recent publications in which the researchers were able to use a semi-structured interview to gain insights into topics that are related to online privacy concerns [34, 35], mental models of the HTTPS protocol [36] as well as developer work habits [37] and the misconceptions about CSP [8]. As these are also topics that are connected to secure web development, we will also use a semi-structured interview to gain insights into the usability of Trusted Types.

As mentioned before, Roth et al. conducted a qualitative study to reveal misconceptions of web developers regarding the CSP [8]. To achieve this goal, they interviewed 12 web developers from different countries. The interview was splitted into three different parts, a semi-structured interview, a drawing task and a coding task. Because the study goals were similar to our study and because Trusted Types also use the CSP to get enforced, we decided to adopt a similar study approach for our Trusted Types study.

Chapter 3

Methodology

3.1 Problem Statement

Although a lot of time has passed since the initial discovery of XSS, it is still a huge problem for a lot of sites on the web nowadays. In 2021 OWASP classified XSS as the third most prevalent vulnerability on the web [38]. Also, almost 14.2 billion dollars were paid to bug bounty hunters that found XSS vulnerabilities on websites of several companies in the same year [39]. Although this appears to be a large sum of money, the average bug bounty was only around 501 dollars. This means that while there were many reported cases of XSS, only a minimal amount of money was awarded to bug bounty hunters for the reporting of these vulnerabilities. This shows that companies do not put enough effort and resources into fixing XSS vulnerabilities. However, with a severe vulnerability like XSS, this can have serious consequences for the website users.

As the client-side code of web applications has risen in the recent time, we could also see a rise in client-side XSS vulnerabilities [4]. Unfortunately, these client-side XSS vulnerabilities are pretty hard to detect for tools as the reasons of these vulnerabilities lie in the dynamic client-side code of complex web applications that often programmatically add HTML code by using dangerous injection sinks [20]. This means that tools cannot find XSS vulnerabilities with a "high degree of confidence" [4]. Therefore, even if companies focused on finding client-side XSS vulnerabilities in their web applications, it would be far from sure that they are able to find all of them. So, to protect users against client-side XSS attacks, we need a security mechanism to secure websites against client-side XSS vulnerabilities. The Content Security Policy already was a step in the right direction, but as mentioned in Chapter 1, with most of the CSPs on the web being trivially bypassable [8], we need another security mechanism that completely removes the possibility for the existence of client-side XSS vulnerabilities. Trusted Types offer such protection by making dangerous injection sinks secure by default. Furthermore, the security mechanism has another advantage to using a CSP, it allows to implement a fine-grained control for *eval* by defining a sanitization policy that is called before data is used in the sink. With a CSP instead, it is only possible to completely disallow the usage of the *eval* function.

But despite the fact that Trusted Types are already implemented in popular web browsers such as Chrome, Opera, and Microsoft Edge, they are still not widely used on the web. To the best of our knowledge, only a small amount of websites, including *Facebook* and *Google*, is already widely utilizing the security mechanism to safeguard their web applications against client-side XSS vulnerabilities. Also, as indicated in Chapter 2, the research community has primarily focused on the adoption of Trusted Types into popular frameworks [4] and the evaluation of efficiency [26]. But apart from that, no research has been undertaken on the usability of Trusted Types for web developers who need to implement the security mechanism in their applications. This, however, is just as important as the research on popular framework adoption or efficiency measurements in our opinion. If developers struggle to correctly deploy Trusted Types in their web applications, these applications will still be vulnerable to client-side XSS vulnerabilities and will expose their users to the risks of an XSS attack.

Furthermore, there are currently only a few available sources of information for developers that allow them to fully comprehend Trusted Types, how they work, and how to implement them. The small amount of information that currently exists is often also incomplete and does not really provide information on how to correctly implement Trusted Types. Even popular documentations on the web, like the *Mozilla Developer Network* [1], only provide basic information on the usage of Trusted Types.

3.2 Research Goals

To improve the usability of Trusted Types on the web, this thesis aims to find common roadblocks for web developers when it comes to the usage of Trusted Types by answering the **following research questions**:

1. **Do developers understand the threat model that is covered by Trusted Types?**
2. **What are the roadblocks for web developers to correctly deploy Trusted Types on their webpages?**

[1]`https://developer.mozilla.org/de/`

To answer these research questions, we will conduct a qualitative study. The structure of the study will be explained in Section 3.3.

At the end of the thesis we will also create a **development guideline** for web developers that can help to correctly implement Trusted Types in their web applications. The development guideline can be found in Section 4.2.

3.3 Study Structure

We decided to conduct a **qualitative study** in order to answer the research questions from Section 3.2. Qualitative methods "are appropriate to uncover and understand what lies behind any phenomenon that is poorly understood" [40]. And this is exactly what we want to achieve for Trusted Types: Uncover and understand whatever roadblocks web developers face in the context of Trusted Types.

The qualitative study was divided into two different parts: A first semi-structured interview with the participant and a coding task. Both parts of the study were undertaken in a meeting with the participant that lasted about one hour. Each participant was interviewed seperately and all of the participants were recruited ahead of time. Section 3.3.5 contains more information about the recruitment procedure. Ahead of the interview, we required the participants to read an article about client-side XSS and Trusted Types [30]. This ensured that they had the required knowledge in these topics to complete all tasks of the study.

Due to the Corona pandemic, the study was conducted online. The participants could suggest an online meeting tool of their choice for the interview. However, if they did not have any preference we would suggest to use Zoom for the interview because it offers a built-in screen sharing method. If participants selected to utilize a video conference tool that does not support screen sharing, they would have to ensure that they could use other programs that allowed us to share our screen with them and the other way arround as it was needed for the coding task. The video conference tool was chosen together with the particpants prior to the interview together with the date.

The interview session and the coding task were also recorded so that we could subsequently transcribe both parts of the study to analyze it later on. The recording program depended on the video conference tool that was chosen for the meeting with the participant. When a participant decided to use a video conference tool that supports the recoding of meetings, we used the built-in recording function. Otherwise, we used OBS to tape the meeting by recording the screen on which the video conference window was located. However, before we started with the recording of the meeting, we asked the participants for their consent to be recorded.

3.3.1 Interview Session

Because our study used a qualitative approach, the interview session with the participant was conducted in the style of a **semi-strucured interview**. Therefore, like in a structured interview, we first created an interview guideline where we established all the questions that we wanted to ask the participants during the interview. However, in a semi-structured interview, the order of the questions can be freely changed during the interview and we could also ask follow-up questions based on the participants' responses. This basically allowed us to go further into detail and to better understand the participants' views on the topics covered by the questions. In the course of the study, the interview guideline was changed as we found a few inconsistencies in it. The final interview guideline that was used at the end of our study is appended at the end of the thesis in Appendix A. Unlike this thesis, the interview guideline was written in German because the participants were also interviewed in German. This decision was made since all of the participants spoke German as their native language and, therefore, it was more intuitive to conduct the study in German. The interview quideline was divided into four different sections:

(1) General Questions on the Participant:
First of all, we started by asking general questions about the participant's area of work or study and previous web development and IT security experience. Furthermore, the particpants were also told to describe the implementation steps that they follow when adding new features to their projects. These questions were considered introductory questions and aimed to get to know the participant. But they can also provide some valuable insight into the individual's web development skills and IT security knowledge. Nevertheless, this section may also reveal information about the participant's attitude towards software security, such as whether it is taken seriously or not.

(2) Cross-Site Scripting:
In the second part of the guideline, we discussed the notion of Cross-Site Scripting. This part was mainly used to reveal the participant's understanding of XSS vulnerabilities and their mitigation approaches. To accomplish this, we first started off with a few questions about the concept of XSS vulnerabilites. Following these questions, we asked the particpants to name different types of XSS vulnerabilities that can occur on a website, along with the dissimilarities between them. We chose

to include these questions because the knowledge of client-side XSS vulnerabilities is crucial for developers to completely understand why and how Trusted Types can help protecting against them. To conclude this section, we also examined the participants' understanding of XSS security mechanisms. Therefore, we interviewed the participants on the security mechanisms they have heard of or even already deployed previously.

(3) Trusted Types:
Following the XSS section, we then tried to reveal the participants' mental model of Trusted Types by interviewing them about the concept of Trusted Types and their perception of the security mechanism. In this regard, we first asked the participants to explain what Trusted Types was and how the mechanism was able to protect web applications against client-side XSS vulnerabilities. Additionaly, we requested the participants to identify any potential issues that could arise from the use of Trusted Types. Following that, we determined if the participants had previously utilized Trusted Types, and if so, why. If they answered this question with "No", we then continued by asking them whether they would use Trusted Types for their next web project after reading the given article about Trusted Types. To be able to answer the last questions we then explained the coding task to the participants and showed them a small code snippet of the coding task that was containing an insecure Trusted Types default policy. We then proceeded to ask the participants to comment on the security of this code snippet and the characteristics of Trusted Types default policies. The code snippet that was shown to the participants can be seen in Listing 3.1.

(4) Final Questions:
The final part of the interview followed immediately after the end of the coding task. First, we asked the participants to state whether they would use Trusted Types after having worked on the coding task. We also raised this question in the previous section before the participants started with the coding task. This allowed us to draw a comparison between the participants' perception of Trusted Types before and after the coding task. Furthermore, we also requested the participants to evaluate the coding task that they previously worked on. The evaluation intended to reveal whatever challenges the participants encountered during the coding task as well as which parts of the coding task they were able to solve without major problems. The main reason why we came up with these questions was to discover further roadblocks for web developers during the implementation process that might not have been revealed during the coding task. To conclude the interview, we then questioned the participants about problems they currently see with Trusted Types, whether they would change the mechanism and how an ideal mechanism would look

```
var escapePolicy = trustedTypes.createPolicy('default', {
  createHTML: string => string,
  createScript: string => string,
  createScriptURL: string => string
});
```

Listing 3.1: Insecure Trusted Types Default Policy (trusted-types.js)

like for them. The information gathered by these questions could then be utilized to further understand how Trusted Types might be improved for a better usability.

The first three parts of the interview session which took place before the coding task were originally scheduled to last about 20 minutes. But depending on the participant, the used time varied a little bit. As an example, one of the participants was not able to talk much about the given questions due to the lack of knowledge and for this reason, the session was already finished after not more than 15 minutes. On the other hand, another participant discussed the questions for a longer period of time than anticipated and as a result, stretched this part of the study to arround 30 minutes. However, if the participants deviated too far from the original questions and hence, taking too much time, we would have asked them to move on to the next question.

3.3.2 Coding Task

In the second part of the study, the participants were instructed to solve a coding task with the goal of securing a web application against client-side XSS vulnerabilities. However, there were two restrictions. The participants needed to secure the web aplication by using the Trusted Types API while the functionality of the web application also needed to stay intact. The main purpose of this coding task was to reveal roadblocks that web developers experience when integrating Trusted Types into their web applications.

For the coding task, we tried to simulate an environment that was as similar as possible to how the participants would normally approach the implementation of web applications so that the results would be representative for other web developers. For that reason, we decided to allow the participants to use every source of information that they could find on the internet as well as copying code from these sources. This also included coding forums like "Stack Overflow" [2], Trusted Types documentation pages like the W3C Trusted Types standard [3] and version control systems like "GitHub" [4]. To prevent misunderstandings concerning the web application and Django in general we also detailedly explained the

[2]`https://stackoverflow.com/`
[3]`https://w3c.github.io/webappsec-trusted-types/dist/spec/`
[4]`https://github.com/`

```
window.addEventListener('load', function () {
    let inj = unescape(location.hash.slice(1));
    if (inj.length === 0)
        return
    if (inj.startsWith('http')) {
        let s = document.createElement('script');
        s.src = escapePolicy.createScriptURL(inj);
        document.body.appendChild(s);
    } else {
        if (inj.startsWith('<')) {
            let d = document.createElement('div');
            d.innerHTML = escapePolicy.createHTML(inj);
            document.body.appendChild(d);
        } else {
            eval(escapePolicy.createScript(inj));
        }
    }
});
```

Listing 3.2: Vulnerability Script (vulnerabilities.js)

```
def response(request, template, context=None):
    if context is not None:
        r = render(request, template, context)
    else:
        r = render(request, template)
    # r.headers['content-security-policy'] = \
    #     "trusted-types default; " \
    #     "require-trusted-types-for 'script';"
    return r
```

Listing 3.3: Response Function (views.py)

web application to the participants in beforehand of the coding task. In the course of the coding task, the participants were encouraged to explain the steps that they were taking during the implementation process. Furthermore, we also asked them to think aloud as their thoughts could provide us with more valuable information that might help us in revealing additional roadblocks.

3.3.2.1 Structure

The web application for the coding task was provided by the web security chair in advance of the study. It was written in *Python* and utilized the *Django* library [5], a high-level web framework. It was decided to provide the coding task only in one programming language and framework because the majority of code that needed to be implemented throughout the coding task was JavaScript code. So that being the case, no language- or framework-specific code was required to complete the coding task.

[5] https://djangoproject.com/

Django projects always have a defined structure. When a Django project is initialized by the developer, the framework will automatically create a project directory. This project directory already includes a subdirectory, called "python", that stores the project configuration files. In addition, it also contains a Python script for managing and executing the Django project. This script can then be used to create applications inside the Django project. When an application is created, a subdirectory with the name of the application is added to the project directory. This directory contains a set of files that must be used to implement the application. Furthermore, the developer can also store the static files and HTML templates in the directory of the application.

For the coding task an application with the name "app" was created. The file structure of the application is shown in Figure 3.1. Every Django application follows the Model-View-Controller pattern, so this is also the case with our application. In Django the model is implemented in the *model.py* file of the application. The model describes how the data that is shown on the website is stored in the database. In our case the only model that needed to be stored in the database was a Note object that users can create in one of the modules of the application.

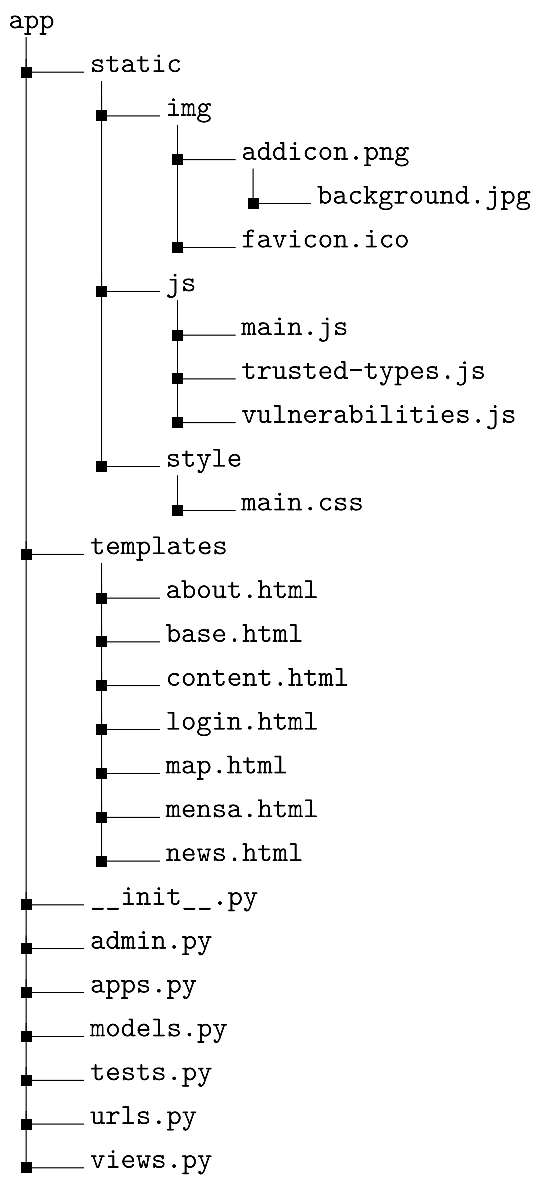
Next, let us have a look at the views of the application. A view is used for presenting the application to the user, so basically the views of a web application are the documents that are sent back to the user and shown in the user's browser. In our case, the views are available as HTML templates in the *templates* directory. The templates will be rendered and filled with data according to the user's request before being sent to the user. Our application has seven different templates. The first template is a base template that includes all functionalities that are needed by every part of the application. Furthermore, there are six more templates that extend the base template and are used when the corresponding application path is requested by the user. There is one template for the login screen, another one for an about page and four more templates for the modules that are needed to be fixed by the participants during the coding task.

In the web application's controller, the templates are rendered and returned to the user. The functionality of the controller is contained in two different files: In the *urls.py* file, the paths of the web application are registered and each path is associated with a function that is called whenver a user requests this path. The implementation of these function is contained in the *views.py* file. In this file the participants would also need to set the CSP in the response to the user to enforce Trusted Types during the coding task. To help the participants, there is already a function containing a code snippet to set a CSP that is commented out at the beginnig of the coding task. The function can be seen in Listing 3.3.

The application also contains a directory with static files that are loaded in the different modules of the application. Two of these files are especially important for the coding task. The first one is the *trusted-types.js* file. It is loaded in the base template and already contains an insecure implementation of a Trusted Types default policy that was already shown in Listing 3.1 in the previous section. This will also be the file in which the participants will later implement the Trusted Types policy. To test their implementation during the coding task, the *vulnerabilities.js* file is also loaded in the base template. The file adds a client-side XSS vulnerability to every module and, therefore, allows the participants to check the security of their Trusted Types code by trying to exploit the vulnerability. The file is presented in Listing 3.2.

In total, there are four different modules that the participants need to secure against client-side XSS during the coding task. But because the time of the coding task was limited to aproximately 30-40 minutes, the participants only needed to secure one of the modules that was chosen for them ahead of the coding task. Every module contains different widgets that are often used in other web applications across the web and that could lead to problems when used in combination with Trusted Types. The content of the modules can be seen in Figure 3.2. In the following we will explain the content of the modules in detail and explain which problems might arise when they are used in combination with Trusted Types:

(1) **Notes Module:**
In the notes module, there are two different elements that are embedded on the page. First of all, users can create notes that are stored on the server. However, these notes are already inserted into the document when the template of the module is rendered on the server side. Therefore, this part of the website cannot be vulnerable to client-side XSS. But apart from the notes widget, the module is also loading a script from Twitter that is appending a tweet timeline of the account *Saar_Uni* in a same-origin iframe to the document. To achieve this, the script dynamically adds the content in the iframe by calling dangerous sinks. But as the CSP of a webpage will be inherited to all same-origin iframes, Trusted Types will also be enforced in this iframe. The problem that the participants would face in this case is that the dangerous sink calls would fail in the iframe as no Trusted Types objects are used when cdangerous sinks inside the iframe are called. Therefore, the browser will not append the content to the iframe and, as a result, nothing will appear, leaving the functionality of the module broken. To solve this problem, participants would need to inject a Trusted Types default policy into the iframe so that the browser would again add the tweet timeline content into the iframe.

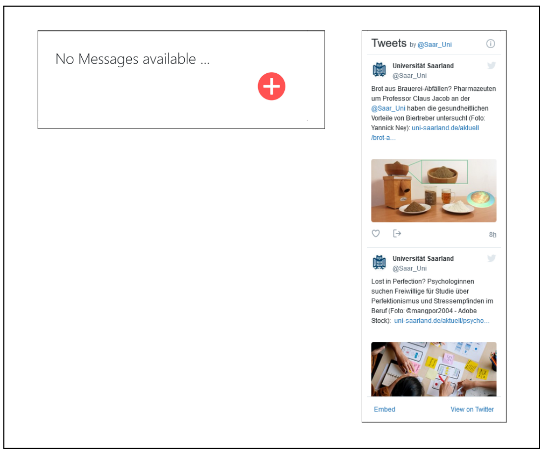


```
app
├── static
│   ├── img
│   │   ├── addicon.png
│   │   │   └── background.jpg
│   │   └── favicon.ico
│   ├── js
│   │   ├── main.js
│   │   ├── trusted-types.js
│   │   └── vulnerabilities.js
│   └── style
│       └── main.css
├── templates
│   ├── about.html
│   ├── base.html
│   ├── content.html
│   ├── login.html
│   ├── map.html
│   ├── mensa.html
│   └── news.html
├── __init__.py
├── admin.py
├── apps.py
├── models.py
├── tests.py
├── urls.py
└── views.py
```

Figure 3.1: File Structure of the Django Application

(2) News Module:

The news module contains a news window that shows latest news about the web application. The content is hardcoded and, therefore, is not prone to XSS vulnerabilities. Nevertheless, there is also another element on the website, a *Disqus* iframe that loads a discussion thread for users to discuss the latest news directly on the page. The *Disqus* iframe is not appended as a same-origin iframe and, therefore, does not have the same problem as the Twitter timeline iframe. Still, the iframe is added with the help of a script that is loaded from *Disqus* by dynamically adding it to the document with the *script.src* sink. Therefore, if a participant creates a policy that is too restrictive, the widget might not load and the functionality would be broken.

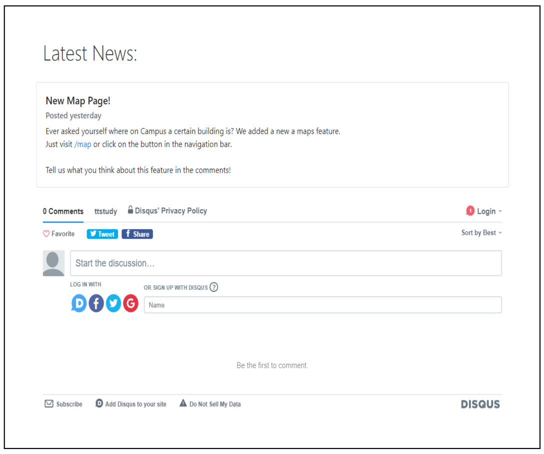

No Messages available ...

Tweets by @Saar_Uni

Universität Saarland
@Saar_Uni
Brot aus Brauerei-Abfällen? Pharmazeuten um Professor Claus Jacob an der @Saar_Uni haben die gesundheitlichen Vorteile von Biertreber untersucht (Foto: Yannick Ney): uni-saarland.de/aktuell/brot-a...
8h

Universität Saarland
@Saar_Uni
Lost in Perfection? Psychologinnen suchen Freiwillige für Studie über Perfektionismus und Stressempfinden im Beruf (Foto: ©mangpor2004 - Adobe Stock): uni-saarland.de/aktuell/psycho...

(a) Notes Module

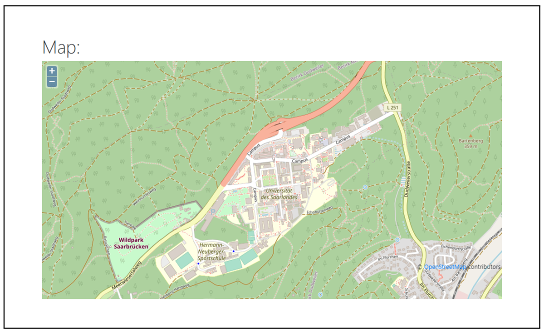

Latest News:

New Map Page!
Posted yesterday
Ever asked yourself where on Campus a certain building is? We added a new a maps feature.
Just visit /map or click on the button in the navigation bar.

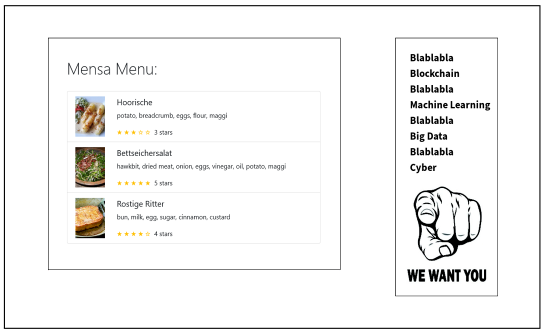
Tell us what you think about this feature in the comments!

0 Comments ttstudy Disqus' Privacy Policy Login
Favorite Tweet Share Sort by Best
Start the discussion…
LOG IN WITH
OR SIGN UP WITH DISQUS
Name
Be the first to comment.
Subscribe Add Disqus to your site Do Not Sell My Data DISQUS

(b) News Module


Map:

(c) Map Module


Mensa Menu:

Hoorische
potato, breadcrumb, eggs, flour, maggi
★★★☆☆ 3 stars

Bettseichersalat
hawkbit, dried meat, onion, eggs, vinegar, oil, potato, maggi
★★★★★ 5 stars

Rostige Ritter
bun, milk, egg, sugar, cinnamon, custard
★★★★☆ 4 stars

Blablabla
Blockchain
Blablabla
Machine Learning
Blablabla
Big Data
Blablabla
Cyber

(d) Mensa Module

Figure 3.2: Content of Coding Task Modules

(3) Map Module:

The map module basically loads an *OpenStreetMap* of the Saarland University. Just like the *Disqus* widget, the *OpenStreetMap* is also loaded via a script with the difference that the script is not dynamically loaded with JavaScript code. Also, the loaded script does not include an iframe containing the content of the widget but instead adds the content of the map directly to the document by using dangerous sinks. The problem that the participants might face when trying to secure the module is that the *OpenStreetMap* code is not controllable and, therefore, does not allow the participants to change the content of the script to use their Trusted Types policy. For this reason, the participants would need to use a default policy that would then be utilized in the third party *OpenStreetMap* script. But also when this is done, participants again could still make the policy too restrictive, so that the map would not be embedded correctly into the page.

(4) Mensa Module:
The last module that might need to be secured against client-side XSS by the participants is the mensa module. In this module there are again two different elements. First of all, there is a mensa menu that is loaded with the help of a script. This script first loads a file that contains the mensa menu data in the form of a JavaScript string that, when executed, adds the menu to the document. The script executes this JavaScript string by using the *eval* sink. Next to the mensa menu there is also an ad that is loaded from an external source. The ad is loaded in the form of a script that dynamically adds HTML by creating an image tag that loads the image by setting the *src* attribute to the base64 encoded image data. Here again, the participants have the challenge of not implementing the policy too restrictive as it might break the functionality.

Furthermore, all of the different modules are loading a *Google Analytics* script as a lot of developers use this tool to analyze the traffic on their websites. However, this should also not cause any problems for the participants during the coding task because no dangerous sinks are used in that script to add content to the document.

3.3.2.2 Distribution

We provided three different ways for the participants to retrieve and solve the coding task:

(1) Remote Control:
First of all, we gave the participants the possibility to solve the coding task on our machine through remote control. Therefore, we installed all required Python libraries together with a set of popular IDEs on our machine ahead of time. To perform the remote control, the participants could decide between TeamViewer and the built-in Zoom remote control. The Zoom remote control, however, was only available when the meeting was also held via Zoom.

(2) Docker Container:
The second option that we gave the participants was a Docker container. We, therefore, sent the participants a zip file containing a *Dockerfile*, a *docker-compose* file and the source code of the coding task. To start the docker container, the participants simply needed to run the *docker-compose up* command. As soon as the command was executed, the docker container was created and the source code directory was mounted into the docker container to run the web application. All of the needed libraries were also installed during the creation of the container. To

work on the coding task, the participants simplpy needed to edit the files on their own machine with the IDE of their choice as the changes were also reflected in the docker container due to the directory mount.

(3) VirtualBox VM:
Additionally, we also gave the participants the option of using a VirtualBox VM for the coding task. Therefore, we created a virtual machine in VirtualBox and again installed all required libraries as well as popular IDEs that the participant might use to edit the source code. After everything was set up correctly, we started to export the virtual machine to a file. This file was sent to the participants once they decided to use the virtual machine for the coding task. The file then needed to be imported in VirtualBox. Once this process was finished, the participants were able to start the VM and could work on the coding task.

3.3.3 Transcripts

To be able to analyze all the data that we collected during the interviews, we transcribed both, the interview and the coding task, with the help of the recordings afterwards. In order to capture every detail, we chose to use the true verbatim transcription sytle. Thus, we also transcribed word fillers that were used by the participants as well as sounds that they made, such as laughter [41]. This specific transcription style was chosen because it is also displaying how a particpant has said something as this can change the meaning of a statement given by the participant.

As a first step, we gathered all of the information from the audio recording by transforming the conversation into a written text. This process was done manually by listening to the captured recordings. However, to speed up the process, we used the transcription tool *OTranscibe* [6]. Although the tool is only available online, the site stores all data in the *localStorage* of the browser, meaning that no recordings or transcripts left the browser to be shared with the website.

In a second step, we also had a look at the video recordings of the coding task. The reason for this choice was that some information could not be extracted from the audio recording. This could include actions taken by the participant during the coding task that were not described, or problems experienced by the participant that were not expressed in the audio recordings. On top of that, the video recordings also showed the sources that the participant used during the coding task. In the transcriptions we marked the observations that we retrieved from the video recordings by enclosing it into square brackets.

[6] `https://otranscribe.com/`

3.3.4 Open Coding

After we finished with the transcription of an interview, we began with the open-coding process to analyze the information contained inside it. The main purpose of this process is "to build concepts from a textual data source" and to "expose the meaning, idea and thoughts in it" [42]. Often this data analysis approach is utilized in order to anaylze qualitative data.

During the open-coding process we created codes and assigned them to the statements in order to analyze the data and to derive concepts from it. We decided to group the codes into different categories depending on what concept they describe. For example, we grouped all of the codes that described roadblocks during the development of Trusted Types in one category. In the beginning, we created a draft of the codebook by assigning codes during the first reading of the transcriptions. This initial codebook then was further improved until we thought that the codes were meaningful and unique. Our final codebook is listed in Appendix B.

We also used the open coding process to determine the end of the study. We decided to end the interviews whenever we encountered no new concepts after assigning codes to transcript. This was the case after a total amount of five interviews.

3.3.5 Recruitment

In general, we approached the recruitment of participants like this:

(1) We reached out to possible participants by advertising the study on different platforms.

(2) When people declared interest in participating, we

- decided on a date for the interview.
- decided on a meeting tool for meeting up with the participant.

(3) We asked the participants to read an article about client-side XSS and Trusted Types in beforehand of the study.

We decided not to require the participants to have worked in web development before, because it was complicated to find enough participants for the study in the time frame of the thesis. At the beginning of the study, we started to recruit participants from a German IT company. This decision was made because we were employed as working students at the company during the time of the study. To contact the employees for advertising our study, we used the internal company communication platform. But although the company employs a lot of web developers, we struggled to find enough employees that were interested in participating in the study. Also, the ones that were interested did not work in the web development department. After we conducted the study with these participants, we were still in need for more participants.

As a result of their small experience in web development, we decided to further recruit students from *Saarland University* who were participating in the *Foundations of Web Security* course. The students were contacted with a message in the content management system of the course. This message also triggered an email containing the message that was sent to every student that took the course. The content of the message can be seen in Appendix C.

Chapter 4

Findings

In this chapter we will present the results of the qualitative study. This includes the demography of the participants that were interviewed for the study and the results of our analysis. The analysis revealed not only the participants' mental models of XSS and Trusted Types but also their perceptions of the mechanism. Furthermore, we were also able to shed light on the roadblocks that developers encounter during the implementation process. We also captured the participants' ideas on how to improve Trusted Types to make the mechanism more usable for web developers in their opinion. Furthermore, we also present an implementation guideline for web developers to securely implement Trusted Types. At the end, we also state the limitations that might have influenced the findings of our study.

4.1 Study Results

In this section we will summarize the findings of our study. First, we will start by presenting the demographics of our participants. After that, we will proceed with the mental model of the participants regarding XSS vulnerabilities and the Trusted Types mechanism. Following this section, we also present the perceptions of Trusted Types and the discourages to use the security mechanism that the participants came up with during the interview. In the final two sections we will finally state the deployment strategies that were used by the participants during the coding task as well as the roadblocks that we discovered by conducting the study.

4.1.1 Participant Demography

In total, we recruited five persons to take part in the study. Three of them worked for a German IT company, while the other two were *Saarland University* students who took the *Foundations of Web Security* course. Our study's population consisted solely of male persons ranging in age from 21 to 25 years. None of the participants was working in web development at the time of the study. However, with the exception of one participant, all individuals had implemented at least tiny websites or worked on minor web projects prior to the study. All of the participants were still students at *HTW Saar* or *Saarland University* that either studied *Cybersecurity* or *Computer Science.* Two had already completed their bachelor's degrees and were working on their master's degrees, while the other three were still in their bachelor's. Along with their studies, all of the participants worked as students. Four of the participants worked in the field of IT security, with the final person working in DevOps. However, all of the participants had a background in IT security, either through their studies or through their day-to-day work in their working student job. When it comes to security awareness during implementation, one participant stated that security is secondary during implementation and that code is only evaluated for vulnerabilities before deploying an application.

4.1.2 Mental Model

In the first part of the interview with the participant we tried to reveal the participants' mental model of XSS and Trusted Types. The questions about XSS mainly focussed on how the vulnerabilities can occur, which types of XSS exist and, furthermore, how to protect against them. For Trusted Types, we wanted to know from the participants how the mechanism works and why it helps to protect against client-side XSS. However, we did not only use the answers from the participants during the first part of interview, we also used the comments that were made during the coding task to learn about the participants' understanding of the topics.

XSS:

(1) **XSS Vulnerabilities:**
First of all, the participants had a good understanding of XSS vulnerabilities in general. They stated that the vulnerability can be used to inject malicious code into a website but also that this happens because of the confusion of text as code, when appended to the HTML document and, therefore, is interpreted and executed as JavaScript code. When the participants were asked to describe different attack scenarios that can be used to exploit a XSS vulnerability, the participants

mentioned a variety of different attacks. First, the participants stated that a XSS vulnerability can be used to execute arbitrary JavaScript code allowing an attacker to do everything that is possible with JavaScript. When asked about specific scenarios, two participants mentioned that it is possible to intercept data from the victim like username and password. One of these participants also mentioned more attack scenarios, like installing keyloggers to record the victim's keystrokes, stealing cookies and hijacking user sessions.

(2) Types of XSS:
Furthermore, all of the participants were able to name the two types of XSS vulnerabilities: client-side and server-side XSS. Four of the five participants did even explain the difference between reflected and persistent XSS, although we did not explicitly ask for this. However, when going into detail, some participants had a wrong understanding of client-side, server-side XSS or both. One of the participants, for example, mentioned that in a server-side XSS attack the code injection is triggering functions on the server. This is wrong as the injected code is executed in the user's browser and not on the server. Another participant instead did not know that there are other dangerous sinks than HTML sinks that can lead to client-side XSS vulnerablities, indicating a lack of knowledge when it comes to client-side XSS. There was also one participant that could not explain any difference between the two types of XSS at all. These results are interesting as they slightly differ from the results that were retrieved by Roth et al. in their CSP study [8]. In the paper it was mentioned that the participants often struggled when it came to client-side XSS.

(3) Mitigation Mechanisms:
When it comes to the security mechanisms to protect against XSS vulnerabilities, every participant, except of one, already heard of at least one mitigation technique ahead of the interview. While CSP was known to everyone of these four participants, input sanitization was only mentioned by three of them. The participants that knew CSP at the time of the study also used it before and, therefore, had a pretty good understanding. One of them also knew that CSP does not protect against XSS vulnerabilities in the first place but instead only mitigates existing XSS vulnerabilities. Another participant explained how CSP works: "You can use a CSP to restrict the execution of scripts and, with the help of certain attributes in different levels, restrict exactly which scripts can be executed and from where the scripts can originate, for example". But although most of the participants already used a mitigation mechanism against XSS, no participant had heard of Trusted Types before the interview.

Trusted Types:

(1) Trusted Types Mechanism:
After we finished the questions regarding XSS, we started with questions about the Trusted Types mechanism. All of the participants were able to explain that Trusted Types are protecting websites against client-side XSS vulnerabilities by locking the dangerous sinks. However, a few participants mentioned that they did not yet completely understand the security mechanism. Also, some participants had a wrong understanding of the way how Trusted Types work. One of the participants, for instance, explained: "Trusted Types ensure that no malicious code is included in the input to these potentially dangerous sinks". The statement showed that this participant did not completely understand the mechanism because Trusted Types only ensure that the code is sanitized before being used in the dangerous sinks, meaning that if the sanitization policy is insecure, malicious code may still end up in the dangerous sinks.

(2) Trusted Types Enforcement:
Furthermore, some participants' lack of knowledge related not only to the Trusted Types mechanism but also to the enforcement of Trusted Types. One of the participants that also had a wrong understanding of the Trusted Types mechanism stated that Trusted Types are enforced over a HTTP header that is similar to the CSP, although the correct answer would have been that the mechanism is enforced over the CSP. Another participant that also mentioned to not completely understand Trusted Types, did not know that if Trusted Types are correctly enforced, the browser will not allow any strings to be passed to dangerous sinks. The other participants, on the other hand, had a good comprehension on how to enforce the mechanism and could also explain how the browser enforces the use of the mechanism.

(3) Trusted Types Policies:
At the end, we also tried to reveal the participants' knowledge of Trusted Types policies. First and foremost, all participants were aware of the fact that a Trusted Types policy must be used to sanitize input before it is passed to a dangerous sink. Furthermore, the participants also comprehended that developers can make mistakes during the implementation of a policy when they were presented with an insecure Trusted Types policy implementation. They further mentioned that this might lead to the website still being prone to client-side XSS vulnerabilities, "The way I see it, the same vulnerabilities would exist as before. So it would make no difference". But although participants knew about the purpose of a Trusted Types policy, only two participants understood the difference between a custom

and a default policy. They described the default policy as a fallback policy that is called if no Trusted Types object was explcitly used in a dangerous sink. The other participants either had a wrong understanding of a default policy or could not think of any possible difference between the two types of policies. For instance, one participant thought that the default policy might already include some basic sanitization rules. But despite that two participants were able to correctly explain the specific features of a default policy, none of these individuals was able to identify the advantage of such a default policy of being able to use third party code without rewriting it to use Trusted Types. Finally, let us also have a look at the understanding when it comes to the different sanitization functions of Trusted Types policies. When asked to explain the differences between these functions that can be implemented in a Trusted Types policy, only one of the participants gave a satisfying explanation: It was mentioned that each sanitization function is used for a specific dangerous sink, *createHTML* for HTML sinks, *createScript* for JavaScript sinks and *createScriptURL* for sinks loading a script from an URL. Nonetheless, just like all the other participants, this individual was unsure of what was needed to be done differently and assumed that all of the different sanitizer functions could be implemented the same way.

4.1.3 Perceptions and Discourages

During the interview, the participants were asked multiple questions that focused on revealing their perception of Trusted Types as well as why they might be discouraged from using Trusted Types. We will summarize the findings that resulted from these questions in this section. We decided to group the different perceptions of the participants in four different groups: Security, Design Problems, Usage and Awareness. In Figure 4.1 and Figure 4.2 the distribution of perceptions and discourages among the participants is presented

Perceptions:

(1) **Security:**
In the first part of the interview, participants mentioned several security perceptions of Trusted Types. To begin with, all participants agreed that Trusted Types provide **enhanced security** for web applications when properly applied. One of the participants justified this with the argument that Trusted Types allow to completely protect against client-side XSS vulnerabilities when the mechanism is properly deployed. All of the other participants could also imagine that Trusted Types might help to increase the security of their web applications, however, they

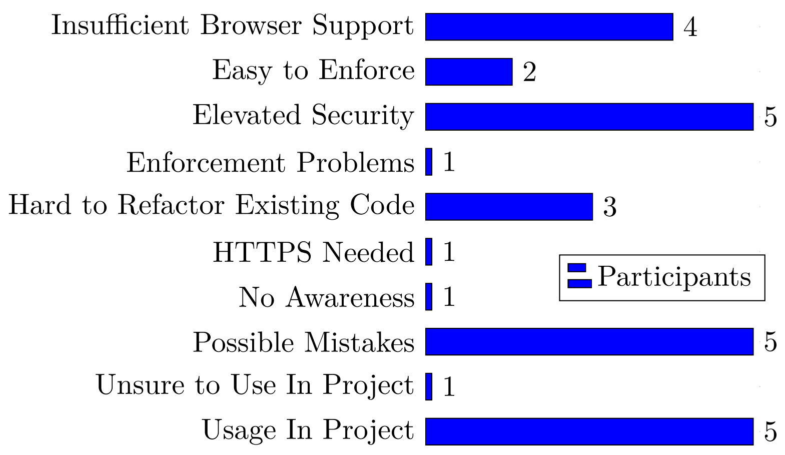


Figure 4.1: Perception Distribution

did not state any reason for their perception. But although the participants perceived Trusted Types to enhance the security of web appications, all of them also stated that developers might still be able to make **mistakes during the implementation process** that could impact the security of the web application against client-side XSS. One of the participants, for instance, also expressed: "I would guess that I would forget something or somehow, there always is some way of loophole". This also demonstrates that the participants not only considered potential implementation errors but also the possibility to overlook steps required to ensure the enforcement of Trusted Types, such as setting the correct CSP headers and that there might even be **loopholes that undermine the security implications of Trusted Types**.

(2) Design Problems:
Aside from the security perceptions of Trusted Types, participants moreover discussed the issues they see with the current design of the security mechanism. The first problem that was mentioned by one of the participants was the fact that **Trusted Types only work over HTTPS**. In the opinion of the participant this could prevent or disencourage web developers from deploying Trusted Types as not every website posesses a certificate to encrypt the connections to its users. Secondly, three participants also observed that it might be **hard to refactor existing code of web applications** to use the Trusted Types mechanism. As a result, one of the participants perceived that this might encourage web developers to create an exception for parts of the code that might be hard to migrate what could leave parts of the code vulnerable to client-side XSS vulnerabilities. Another person also mentioned that long time of the refactoring process might lead to companies disencouraging their web developers to implement Trusted Types because it might

not be useful to invest this time. However, this perception was only mentioned by participants that did not have a good understanding of default policies and for this reason did not know that such policies allow to use Trusted Types without refactoring existing code bases. Last but not least, with the exception of one person, all of the participants criticized that Trusted Types are currently **only supported by browsers that are based on the chromium project**. In their opinion, "this must work for all browsers, or at least there must be no error or disadvantage if you use a browser that does not support it". Although this is not really a design problem of Trusted Types but rather of different web browsers that decided not yet to adopt the security mechanism.

(3) Usage:

Although the participants mentioned a few design problems, most of the participants stated that they would consider **using Trusted Types in their next web project**. We first asked the participants right before the coding task on whether they would use Trusted Types. Additionaly, we raised the same question again after the participants had finished the coding task. Four of the five participants mentioned in both parts, before and after the coding task, that they would use Trusted Types in their next web project. However, there was also one person that was **unsure after the coding task**, although the individual considered using the mechanism earlier in the interview. The fact that so many persons decided to give the security mechanism a try is surprising as most of the participants struggled during the coding task to deploy a secure Trusted Types implementation. In the course of the interview, the participants also mentioned some resasons why they might consider the usage of Trusted Types in their next project. One of the participants, for example, believed that it was **easier to use Trusted Types to enforce sanitization of malicious input** before assigning it to dangerous sinks than to manually find and guard every sink by sanitizing the input in beforehand. Another person also mentioned that it is **easy to enforce Trusted Types** via the CSP. Nevertheless, there was also one participant that stated the exact contrary. The person observed that web developers often **struggle to deploy a sane CSP** to protect against XSS vulnerabilities and, therefore, had the opinion that a lot of web developers would fail to correctly use the security mechanism because of a wrongly configured CSP.

(4) Awareness:

Finally, one participant perceived **Trusted Types to have an awareness problem**, which might be one of the reasons Trusted Types are still rarely used on the web according to the individual. The person stated that the awareness for vulnerabilities in general, including XSS vulnerabilities, is not really given "in the

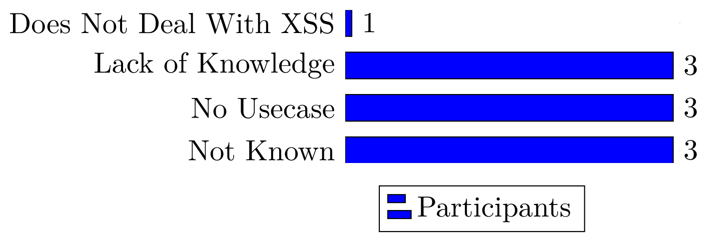


Figure 4.2: Discourage Distribution

developers' everyday life". For this reason, the participant hypothesized that this might also lead to a low awareness of the security mechanisms that could be used to protect against XSS vulnerabilities and, furthermore, the low usage of Trusted Types on the web.

Discourages:

In addition to the perceptions of the participants, the interview also revelad certain points that discourage the participants in their next project or discouraged them in the past to use Trusted Types. First, three of the five participants mentioned that they had **no usecase** in their projects for using XSS mitigation techniques, especially client-side mitigations. The reason for this was that most of the participants only worked on small projects in the private environment and did not dynamically add any code to the website in these projects. One participant also **did not deal with XSS** vulnerabilities in general before the study and, therefore, did not think about using any mitigation mechanisms to protect against XSS vulnerabilities. Furthermore, a lot of participants also stated that they **never heard of Trusted Types** before the study and for this reason could not use the security mechanism previously, even if they wanted to. Last but not least, three participants also mentioned **a lack of knowledge to be responsible for their discourage** to use Trusted Types, "I miss the knowledge [...] for the correct usage of Trusted Types". The main reason for this was that the participants did not deal with Trusted Types before.

4.1.4 Deployment Strategies

During the coding task, the participants used different approaches to solve the assignment. We will summarize the steps and strategies that the developers took to deploy the Trusted Types mechanism as well as the information sources and tools that were utilized by the participants during the coding task. We categorized the strategies in three different

TT Implementation	General Strategies	Information Sources
No Scripts Dynamically Added	Copy Code	Blogs
PolicyViolations Event Listener	Developer Console	GitHub
Same Code for all Sanitization Functions		Library Docs
Set CSP		Mozilla Developer Network
Using Sanitization Library		Trusted Types Article
Test Exploitability		Trusted Types W3C standard

Table 4.1: Categories of Deployment Strategies

categories that are shown in Table 4.1: Trusted Types implementation, general strategies and information sources:

(1) **Trusted Types Implementation:**
Throughout the interview, participants used different strategies for trying to deploy a secure Trusted Types implementation. The workflow of the participants can be seen in Figure 4.3. All participants except of one started off by **trying to set a CSP**. However, one of these participants struggled to do so and was not able to complete this step. In the next step, three of these four participants decided to **use a sanitization library** to implement the *createHTML* Trusted Types policy, while the other individual instead had the idea of **implementing a PolicyViolation Listener** to later spot Trusted Types violations in the implementation process. Nevertheless, after finishing these steps, the indiviudal that was implementing the PolicyViolation Listener as well as one of the participants that had the idea of using a sanitization library stopped after this step and did not proceed any further due to the lack of knowledge. The other two participants again followed different approaches to continue the implementation. The first individual started to **test the exploitability of the implemented Trusted Types policy** with the help of the vulnerability script, while the other person decided to copy the *createHTML* code into the other sanitization functions of the Trusted Types policy. As we mentioned before, there is also one participants that used a different starting approach than all other participants. This person decided to first test the exploitability of the

Trusted Types code that was already implemented at the start of the coding task and then decided to follow a radical approach by **allowing no scripts to be added dynamically to the website**. Furthermore, this participant also used the **same code for all sanitization functions of the Trusted Types policy**. As a final step, the person again tested the exploitability of the web application to check whether the implemented Trusted Types code secures against client-side XSS vulnerabilities.

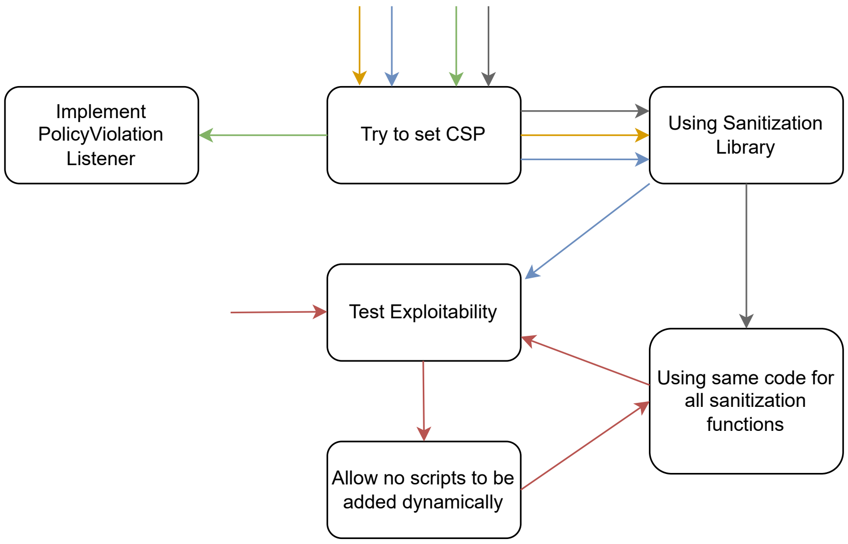

Implement PolicyViolation Listener
Try to set CSP
Using Sanitization Library
Test Exploitability
Using same code for all sanitization functions
Allow no scripts to be added dynamically

Figure 4.3: Workflow of the Participants

(2) General Strategies:

To complete the steps that the participants took to implement a Trusted Types policy that was secure in their opinion they used two different strategies. First of all, the participants **copied code from various information sources on the web**. Most of the participants simply copied the code from websites one by one without changing a single statement. One participant even copied a code snippet containing a comment that warned about using the snippet due to possible client-side XSS vulnerabilities. Furthermore, the participants also used the browser console for different purposes. At the beginning, the participants used the **browser console** to check whether a CSP is set for the current document. However, later they also used the console to get information about the errors that occured in their code. But not all participants used the console for this purpose as some of them did not realize that their code threw an error.

(3) Information Sources:

Furthermore, the participants also used various information sources throughout the coding task to gather the necessary information for a secure Trusted Types

Application Roadblocks	Knowledge Gaps	Coding Strategies
3rd Party Code	Bad Information Sources	Copying Insecure Code
Little Framework Support	False Security	Focussing on Specific Vulnerability
Website Breaks	Identical Policy Functions	
	Insufficient JS Knowledge	
	No CSP	
	Unsure where to Start	

Table 4.2: Categories of Roadblocks

implementation. The first source that was used to acquire information about Trusted Types, was **GitHub**. On GitHub, two participants had a look at the *DOMPurify library* that they thought of using in their implementation. Next, the participants also used **blog articles about Trusted Types** to learn more about how to implement Trusted Types. One of the blogs that was used by the participants was the one that they needed to read before the study. However, they also used the **Mozilla Developer Network** as well as the **Trusted Types standard** that was published by the W3C.

4.1.5 Roadblocks

During the coding task, we could reveal several roadblocks that the participants encountered. We again tried to categorize them in different categories and at the end came up with three different categories that can be seen in Table 4.2: Application roadblocks, knowledge gaps and coding strategies. The distribution of the roadblocks among participants is also shown in Figure 4.4.

(1) **Application Roadblocks:**
Throughout the study, our participants encountered multiple technical roadblocks. First, one of the roadblocks was mentioned by one of the participants after the coding task when the individual was asked what was hard for him during the coding task. The participant stated that the **missing framework support** for Trusted Types made it hard for him to correctly implement the security mechanism. In the opinion of this person, this might also be a roadblock for other web developers that

only use frameworks in their everyday work and, therefore, are not familiar with plain JavaScript code. Another roadblock that was encountered during the coding task was the **use of third party code**. One participant struggled to understand how to implement Trusted Types when third party code using dangerous sinks was included on the website. The other technical roadblocks were, however, revealed during the coding task. First of all, one participant implemented a Trusted Types policy that led to the **functionality being broken** as long as the policy was enforced. The participant did also not understand why the website broke and how to change the Trusted Types code for the web application to work again. The participant then tried to recover the initial functionality of the web application by loosening the Trusted Types sanitization policies. However, the individual failed and was not able to get the web application running correctly again.

(2) Knowledge Gaps:
In comparison to the application roadblocks, the participants encountered more roadblocks that were related to different knowledge gaps. The first roadblock was encountered by two participants. They did not know the differences between the sanitization functions of Trusted Types policies. Due to this lack of knowledge they decided to use the **same code for every sanitization function** in their Trusted Types policy and for this reason were not able to adequately protect the web application against client-side XSS. Also two of the participants **did not set a CSP to enforce Trusted Types** but still thought that their Trusted Types code would be secure without it. These last two roadblocks also show that the participants had a **false sense of security** as they implemented an insecure Trusted Types implementation that was considered secure in their opinion. However, this means that the web application is still vulnerable to client-side XSS and, therefore, can be exploited by an attacker. Later in the interview participants also mentioned that the **information sources about Trusted Types are not very good** to understand Trusted Types and are "only suitable for beginning". Therefore, participants were not able to inform on how to implement Trusted Types for specific cases. Due to this roadblock, participants often also **did not know where to start implementing** due to their lack of knowledge regarding Trusted Type policies. Finally, some participants also mentioned that they only used web frameworks to implement web applications and, therefore, had a **lack of knowledge when it came to JavaScript** what prevented them from correctly implementing Trusted Types.

(3) Coding Strategies:
Apart from the knowledge gaps of the participants, they also encountered roadblocks due to their coding strategies. First of all, some participants **copied insecure**

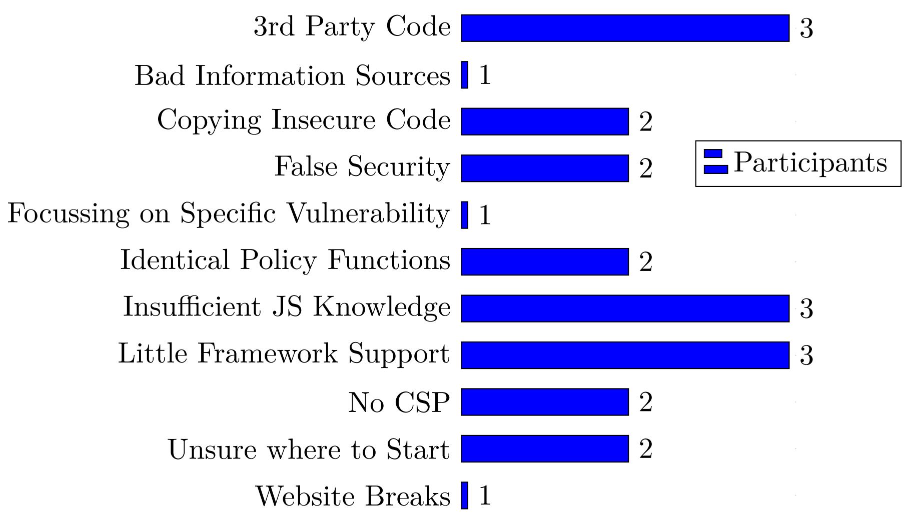


Figure 4.4: Roadblock Distribution

code from the internet and used it in their Trusted Types implementation. One of the participants, for example, copied a piece of code that was annotated with a warning that the code might be vulnerable to XSS. The participant even was aware of that but still insisted on using this code snippet. Secondly, another participant also **focussed on trying to fix a specific vulnerability** from the vulnerability script although the task was to secure the application in general against client-side XSS. Therefore, only this specific vulnerability was fixed by adapting the Trusted Types code to this vulnerability. However, the whole rest of the application was still prone to client-side XSS in this case. This shows that developers might not always be aware of possible client-side XSS vulnerabilities and, therefore, simply try to fix known vulnerabilities with the help of Trusted Types.

4.1.6 Improvements

After the coding task, the participants were also questioned about how they would improve the Trusted Types mechanism and how a perfect mechanism would look like for them. We will summarize the ideas of the participants in this section.

To begin with, one participant mentioned that Trusted Types **need to be supported by all browsers**. In the opinion of the participant, the fact that not all browsers are supporting Trusted Types requires web developers to put more time into defending against client-side XSS vulnerabilities. This is because they need to implement further security mechanisms apart from Trusted Types because otherwise, users of browsers that do not support Trusted Types would not be protected against client-side XSS. The time factor also played a role in the second improvement proposal that was mentioned by the participants. One individual **perceived the refactoring process to be time consuming and stated that this needed to be changed** as otherwise existing web applications would not migrate to Trusted Types. However, the participant did not know how the proposal could be realised and also did not know that a default policy can be used for this purpose. Another participant also mentioned that the **Trusted Types mechanism had to be easier to use**. Nevertheless, this participant also did not know how to change the Trusted Types standard in order to realise this. Last but not least, two of the participants also mentioned that they prefered using frameworks to implement web applications and, for this reason, it would be easier for developers if the security mechanism was **integrated in more frameworks**. One of these persons also stated that frameworks could also convince developers to use Trusted Types by making it mandatory to implement Trusted Types.

4.2 Trusted Types Implementation Guideline

As previously stated, we did not only conduct the study but also tried to create an implementation guideline to help developers in deploying a secure Trusted Types implementation. The goal was to find a guideline that required a minimal amount of work for developers while still protecting websites from client-side XSS if correctly followed. We started by creating a first draft and, from there on, gradually improved the guideline. After multiple improvements, we ended up with the following guideline that is also depicted in Figure 4.5:

(1) First and foremost, developers should determine whether the document they are working on, or any third-party code embedded in the document, utilizes dangerous sinks with untrusted data. If this is not the case, the developer does not need to deploy Trusted Types because the document should not contain any client-side XSS vulnerabilities.

(2) After developers determined that the current document is using dangerous sinks, the following step requires to implement a Trusted Types default policy. The implementation of the policy depends on the type of sinks that are used either in the document or by third party code included in it. If one type of sink is not used within the webpage, the corresponding sanitization function must not be implemented in the default policy. To implement the sanitization functions of the policy, the developer can follow these approaches:

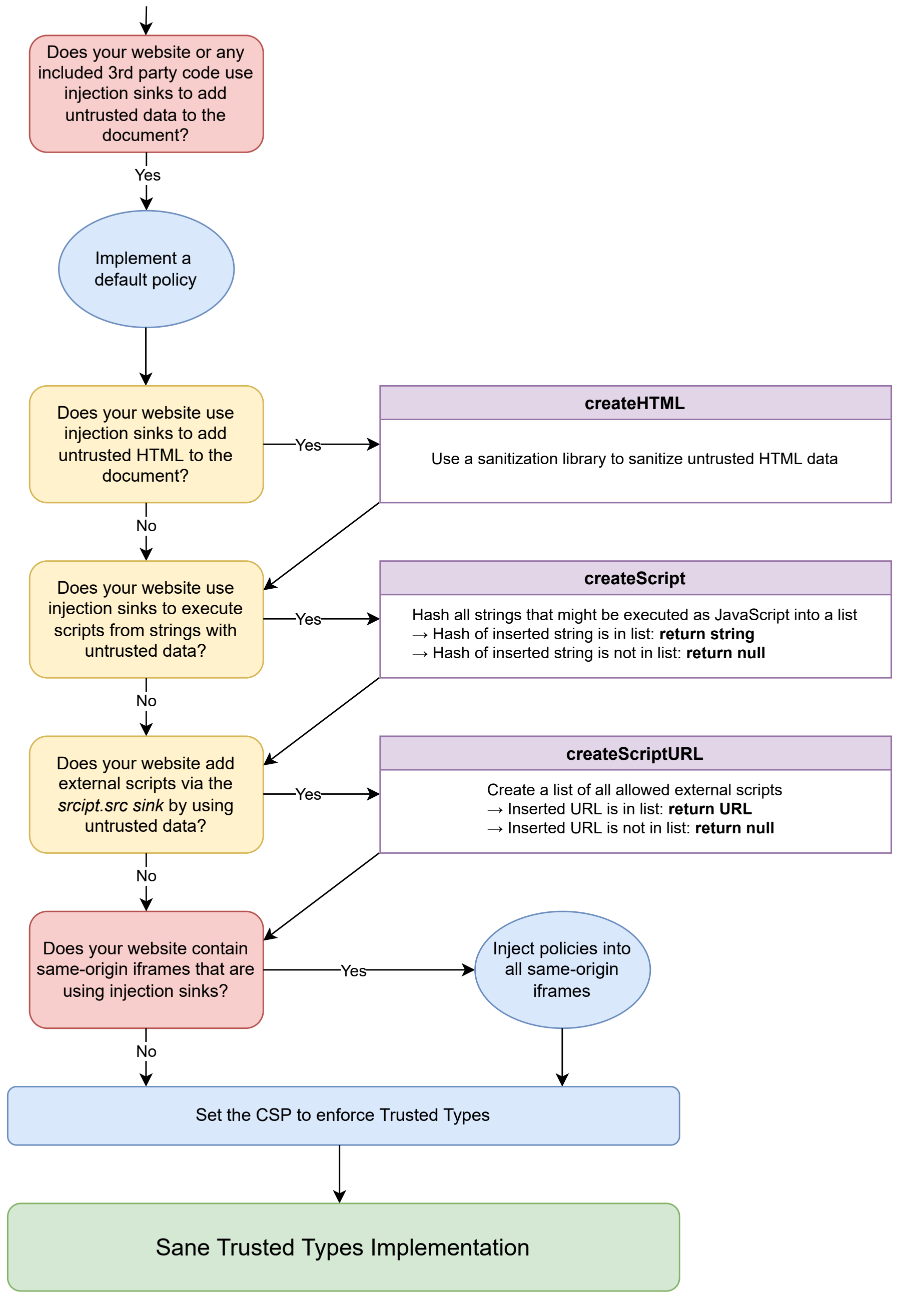

Does your website or any included 3rd party code use injection sinks to add untrusted data to the document?
Yes
Implement a default policy
Does your website use injection sinks to add untrusted HTML to the document?
Yes
createHTML
Use a sanitization library to sanitize untrusted HTML data
No
Does your website use injection sinks to execute scripts from strings with untrusted data?
Yes
createScript
Hash all strings that might be executed as JavaScript into a list
→ Hash of inserted string is in list: **return string**
→ Hash of inserted string is not in list: **return null**
No
Does your website add external scripts via the *srcipt.src sink* by using untrusted data?
Yes
createScriptURL
Create a list of all allowed external scripts
→ Inserted URL is in list: **return URL**
→ Inserted URL is not in list: **return null**
No
Does your website contain same-origin iframes that are using injection sinks?
Yes
Inject policies into all same-origin iframes
No
Set the CSP to enforce Trusted Types
Sane Trusted Types Implementation

Figure 4.5: Trusted Types Implementation Guideline for Web Developers

- **createHTML:** To implement the *createHTML* sanitization function, developers should stick to a sanitization library that allows them to securely sanitize HTML data as it is hard to build a reliable sanitization function on your own. There are multiple possible libraries that can be used for this purpose with the most popular library on the web being *DOMPurify*. In the future there might also be other options available that will already be built into the browser, like the Sanitization API that is currently developed as an experimental feature for *chromium*. In some cases developers might also want to parse HTML without adding the content to the document. However, by using the DOMParser API that also makes use of dangerous HTML sinks the input will automatically be sanitized by our default policy. As we do not want the input to be sanitized, the developer needs to create a custom policy in which the *createHTML* function simply returns the original string. This policy should then only be used explicitly to create a Trusted Types object for the DOMParser API to prevent sanitization.

- **createScript:** To sanitize data that might be executed as a script, the developer needs to collect all strings that should be allowed to be executed in a JavaScript sink. After this step is done, the developer needs to hash all of these strings and store the hashes in a list. When the sanitization function is called, it should hash the input that should be inserted into the JavaScript sink and should compare it to the list of allowed hashes. If the input hash is contained in the list, the function shall return the input string. In all other cases the function should return *null* as then a PolicyViolation error is thrown by the browser and no string is passed to the JavaScript sink. However, to compute the hash of the function input, the developer must again stick to an external library. There is also a built-in Crypto API in popular browsers but the API functionality is only provided with asynchronous functions and, therefore, cannot be called in the synchronous sanitization functions of a Trusted Types policy.
- **createScriptURL:** The sanitization process that should be implemented here is similar to the one in *createScript*. First of all, the developer should create a list of all external scripts that will be loaded dynamically with the help of a dangerous sink. When the function is called, it should check whether the URL of the external script that should be loaded matches one of the URLs in the list. If this is the case, the function should return the URL, in every other case we do not want to add the script and, for this reason, return *null* to prevent the browser from passing any data to the sink.

(3) Once this step is completed, the developer once again has to inspect the document to determine whether the document includes same-origin iframes on the webpage. The reason for this is that the CSP gets inherited to all same-origin iframes. This means that when Trusted Types are enforced in the document, this is also the case for all same-origin iframes. However, despite the fact that the CSP is inherited by JavaScript, variables defined in the document, including all Trusted Types policies, are not exposed to same-origin iframes. Therefore, if no Trusted Types code is implemented in the same-origin iframes, all calls to dangerous sinks will fail as Trusted Types are enforced. To circumvent this problem, developers must inject the default policy into all same-origin iframes.

(4) Finally, after these steps only the CSP is left. Developers now should set the CSP to enforce the usage of Trusted Types. This step can either be done via the CSP header, but it is also possible to include the CSP directly in the HTML document in the form of a meta tag. When this step is finished, the developer has a sane Trusted Types implementation that should secure the webpage against client-side XSS vulnerabilities, assumed that all steps were followed correctly.

4.3 Limitations

Although we revealed a number of roadblocks throughout the study, there are some limitations to the study design that may have influenced the results. To begin with, we only interviewed a small number of participants because the study had to be completed within the time range of this thesis and we, therefore, immediately stopped the interviews when we did not encounter any new concepts in an interview. This might have led to certain roadblocks not being discovered throughout the study. Additionally, we only interviewed young male persons that were still studying and for this reason did not yet have a large amount of work experience. This could have influenced the results in a way that they are not representative for professional or female web developers. Nevertheless, there were also limitations to the data analysis approach that we used. In Section 3.3.4 we explained that we used open coding on the transcripts to analyze them. This process was performed only by one person in order to meet the criteria of a bachelor thesis. However, open coding should normally be performed by at least two independent coders to discuss, compare and resolve conflicts in the codebook. For this reason, we were also not able to calculate the inter-coder agreement. The absence of a second coder might have had influence on the satisfaction of the codebook. Finally, we also encountered limitations of the coding task. Some participants did not set a CSP to enforce Trusted Types in the course of the coding task and, therefore, were not able to encounter any Trusted Types related roadblocks. For example, the two participants that needed to secure the notes module did not set a CSP and, for this reason, were not able to encounter any roadblocks concerning the breakage of the modules. This could be seen for the notes module as every participant working on it forgot to set a CSP and, therefore, did not encounter any roadblocks concerning same-origin iframes.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis was used to conduct a prestudy on the usability of Trusted Types to reveal the mental model of web developers regarding XSS and Trusted Types as well as roadblocks that web developers face when trying to implement the security mechanism. After finishing the study, we can say that developers often struggle to understand the security mechanism but also fail to correctly implement a secure Trusted Types code to guard web applications against client-side XSS vulnerabilities. The reasons for this varied with the most common roadblocks being third party code usage, insufficient JavaScript knowledge and no framework support for Trusted Types. Concluding, our work shows that the Trusted Types standard currently does not provide a good usability for web developers in terms of both understanding and usage of the security mechanism. Therefore, the Trusted Types standard might need to be adjusted in the future to improve the usability by allowing web developers to fully understand and correctly implement the security mechanism. In order to help developers in correctly deploying Trusted Types until that point is reached, we also created a development guideline that will allow them to create a sane Trusted Types implementation if followed correctly. All in all, our work can help to improve the usability of Trusted Types in the future and that it can contribute to better client-side XSS protection on the web.

5.2 Future Work

As we mentioned in Chapter 1, the main purpose of this work was to conduct a prestudy. This means that the thesis focussed on building a reliable interview guideline that can be used by researchers to perform a follow-up study. Researchers that are conducting a possible follow-up study cannot only use our improved guideline, they can also use the limitations that we encountered in our work to ensure that they are not occuring again. However, despite the limitations that are mentioned in Section 4.3, our results should still be representative for amateur web developers as the participants already had a few experiences with web development. Therefore, the roadblocks as well as the improvement proposals of the participants could already be used by researchers to improve the usability of Trusted Types by changing the Trusted Types standard in order to remove possible roadblocks for the developers. In a follow-up study, researchers could also test the effectiveness of our implementation guideline by dividing the participants in two groups, allowing one group of participants to read the guideline in beforehand of a coding task, while the other group does not get to see it.

Appendix A

Interview Guideline

Hallo $Name,
mein Name ist Philipp Baus und ich werde heute mit Ihnen das Interview für unsere Studie über Trusted Types durchführen. Erst einmal danke, dass Sie an dieser Studie über die Erforschung der Nutzbarkeit von Trusted Types in der Webentwicklung teilnehmen. Ich hoffe, dass wir mit der Studie die Nutzbarkeit von Trusted Types und somit auch die Sicherheit von Webanwendungen verbessern können. Erst einmal, darf ich Sie denn dutzen? Ich bewerte übrigens nicht die Korrektheit Ihrer Aussagen während des Interviews, sondern versuche nur, Probleme ausfindig zu machen, mit denen Webentwickler beim Anwenden von Trusted Types zu kämpfen haben. Das Interview wird zu Analysezwecken außerdem aufgezeichnet werden. Wenn Sie damit einverstanden sind, würde ich Sie bitten, dies jetzt zu Protokoll zu geben.

Als Erstes würde ich mit ein paar Fragen über ihr Arbeitsumfeld, Cross-Site Scripting und Trusted Types starten. Dies sollte ungefähr 20 Minuten dauern. Wenn Sie dabei eine Frage nicht beantworten wollen, können Sie mir dies gerne sagen, dann werden wir die Frage überspringen.

Danach sollen Sie eine kleine Coding Aufgabe lösen, bei der Sie eine Django Web Applikation gegen eine client-seitige XSS Schwachstelle absichern müssen. Dabei sollen Sie Trusted Types nutzen. Die Coding Aufgabe kann entweder per Fernsteuerung auf meinem PC, als VirutalBox VM oder als Docker Container zur Verfügung gestellt werden. Was würden Sie denn bevorzugen? Haben Sie irgendwelche Fragen bisher?

Generelle Fragen zur Person:

- In welchem **Bereich** arbeiten/studieren Sie?
- Sehen Sie sich selbst als **Webentwickler**?
  - Gibt es einen **Grund** dafür?
  - Haben Sie schon an **größeren Webprojekten** mitgearbeitet? Welche Aufgaben haben Sie dort übernommen?
- Haben Sie schon **Erfahrung mit IT-Security** gemacht?
  - In welchem **Bereich** haben Sie schon Erfahrungen gemacht?
  - Haben Sie auch schon Erfahrung mit **Websicherheit**?
- Gibt es in Ihrer Firma Programmier-Richtlinien? / Wie ist ihr Arbeitsablauf, wenn Sie ein neues Feature einbauen?
  - Legen Sie viel Wert auf die Sicherheit bei der Implementierung?
  - Gibt es Gründe, warum Sie darauf (nicht) viel Wert legen?

Cross-Site Scripting:

- Können Sie mir erklären, was eine XSS Schwachstelle ist und welche Folgen eine solche Schwachstelle haben kann?
  - Wodurch entsteht eine XSS Schwachstelle?
  - Wie kann ein Angreifer die Schwachstelle ausnutzen?
  - Was kann ein Angreifer mit einer XSS Attacke erreichen?
  - Wissen Sie in welchem Kontext der injizierte Schadcode ausgeführt wird und welche Folgen dies für den User hat?
- Es gibt verschiedene **Arten von XSS**? Welche fallen Ihnen denn ein?
  - Können Sie mir den **Unterschied** zwischen client- und serverseitiger XSS Schwachstelle erklären?
  - Haben Sie **vor dem Lesen** des Artikels gewusst, dass es zwei verschiedene **Arten von XSS** gibt?

- Haben Sie schon einmal **Sicherheitsmechanismen** zum Schutz gegen XSS **angewandt**?
    - **Welche Sicherheitsmechanismen** haben Sie angewandt?
    - Wie **verhindern** diese Sicherheitsmechanismen eine **XSS Attacke**?

Trusted Types:

- Können Sie kurz erläutern was Trusted Types ist?
- Wissen Sie wie Trusted Types eine Website gegen client-side XSS Schwachstellen absichern?
    - Wie genau verhindert Trusted Types die Injizierung von Schadcode?
    - Wie werden Trusted Types im Browser enforced?
    - Wie werden Trusted Types Objekte erstellt?
- Welche Probleme könnten Ihrer Meinung nach bei der Nutzung von Trusted Types auftreten?
- Nachdem Sie den Artikel gelesen haben, würden Sie Trusted Types **bei Ihrem nächsten Projekt nutzen** um sich gegen client-seitige XXS Schwachstellen zu schützen?
    - Welche **Gründe** haben Sie dafür?
    - Denken Sie, dass Trusted Types Ihnen **helfen werden** sich gegen client-side XSS zu schützen?
- Gibt es einen Grund warum Sie bisher (keine) Trusted Types **für ihre Projekte genutzt haben**?
    - Haben Sie vorher überhaupt einmal **von Trusted Types gehört**?

[Web Applikation erklären]

- Denken Sie, dass der aktuelle Trusted Types **Code sicher** ist?
  - **Warum** vermuten Sie, dass der Code unsicher ist?
  - Ist ihr Code **automatisch sicher gegen XSS** sobald eine Trusted Type Policy implementiert ist? Oder denken Sie, dass ein Entwickler trotzdem Fehler machen kann? Welche wären das?
- Hier im Code wird eine **default Policy** genutzt, haben Sie eine Idee was das Besondere an einer default Policy ist?
  - Können Sie vermuten welche **Vorteile** eine solche default Policy mit sich bringen kann?

Coding Task:

Ihre Aufgabe ist es die Django Web Applikation gegen eine client-seitige XSS Schwachstelle abzusichern ohne dass die Funktionalität des Moduls leidet. Dafür sollten Sie Trusted Types nutzen. Um zu testen, ob ihre Implementierung korrekt ist, lädt jedes Modul ein JS File, dass die App gegen eine client-side XSS Schwachstelle anfällig macht.
Ich würde Sie bitten, während der Coding Aufgabe ihren Bildschirm zu teilen, laut zu denken und alle ihre Schritte so detailliert wie möglich zu erklären, sodass wir später Ihren Gedankengang und Ihr Vorgehen während der Coding Aufgabe analysieren können, um Probleme bei der Implementierung von Trusted Types ableiten zu können. Während der Coding Task dürfen Sie übrigens alle Quellen nutzen, die Sie normalerweise auch beim Implementieren nutzen würden, beispielsweise online nach Lösungen suchen usw.

Abschließende Fragen:

- Nachdem Sie die Coding Aufgabe gelöst haben, würden Sie **(immer noch)** Trusted Types **nutzen** um sich gegen client-seitige XSS Schwachstellen abzusichern?
  - Können Sie mir detaillierter beschreiben, was Ihnen **Probleme bereitet hat** während der Coding Aufgabe?
  - Gab es Sachen, die Ihnen in der Coding Aufgabe **sehr leicht gefallen** sind?

- Was sind Ihrer Meinung die Probleme von Trusted Types?
  - Sehen Sie auch **Probleme im Zusammenhang mit Browsern**?
  - Denken Sie diese Probleme sind der Grund warum Trusted Types aktuell noch nicht oft genutzt werden?
- Was müsste sich ändern, damit Sie den Mechanismus benutzen könnten?
- Denken Sie, Trusted Types ist ein geeigneter Mechanismus um sich gegen client-side XSS Schwachstellen zu schützen?
- Wie würde ein idealer Mechanismus für Sie aussehen?

Appendix B

Codebook

Demography

- ◇ **Experience IT-Security**

  Talks about his experience in IT-Security.

- ◇ **Programming Policy**

  Follows a specific programming policy when implementing.

- ◇ **Security Awareness**

  Has good security awareness.

- ◇ **Security Secondary**

  Does not focus on security during implementation.

- ◇ **Web Development Experience**

  Talks about his web development experience.

- ◇ **Work/Study Area**

  Talks about his work or studies.

Discourage

- ◇ **Does not Deal with XSS**

  Does not come in contact with XSS and therefore does not need to deploy Trusted Types.

- ◇ **Lack of Knowledge**

  Has a lack of knowledge when it comes to Trusted Types and therefore is disincented to use Trusted Types.

- ◇ **No Usecase**

  Did not yet have a usecase where it would be useful to deploy Trusted Types.

- ◇ **Not Known**

  Did not know Trusted Types before.

Improvements

- ◇ **Better Framework Support**

  Suggests to improve the framework support for Trusted Types.

- ◇ **Easier to Use**

  Suggests to make Trusted Types easier to use for developers.

- ◇ **Full Browser Support**

  Suggests to support Trusted Types in all browsers.

- ◇ **Less Refactoring Work**

  Suggests to change Trusted Types so that the refactoring of existing applications takes less work.

Information Source

- ◇ **Blogs**

  Uses blog articles as an information source for the coding task.

- ◇ **GitHub**

  Uses GitHub as an information source for the coding task.

- ◇ **Library Docs**

  Uses documentation pages from various libraries as an information source for the coding task.

- ◇ **Mozilla Developer Network**

  Uses the Mozilla Developer Network as an information source for the coding task.

◇ **Trusted Types Article**

Uses the given Trusted Types article as an information source for the coding task.

◇ **Trusted Types W3C Standard**

Uses the Trusted Types W3C standard to get information about Trusted Types.

Knowledge Gap

◇ **Client-Side XSS**

Has a lack of knowledge when it comes to client-side XSS.

◇ **Server-Side XSS**

Has a lack of knowledge when it comes to server-side XSS.

◇ **TT Default Policy**

Has a lack of knowledge when it comes to the Trusted Types default policy.

◇ **TT Enforcement**

Does not know how to enforce Trusted Types.

◇ **TT Insecure Policy**

Does not know that the TT example policy is completely insecure.

◇ **TT Mitigation Technique**

Has a lack of knowledge when it comes to the mitigation technique that Trusted Types use to protect against client-side XSS.

◇ **TT Object**

Does not know how a browser knows that an object is a Trusted Types object.

◇ **TT Policies**

Has a lack of knowledge when it comes to Trusted Types policies.

◇ **XSS Mitigations**

Has a lack of knowledge when it comes to XSS mitiagtions.

Perception Trusted Types

- ◇ **Insufficient Browser Support**

  Thinks that the browser support for Trusted Types is insufficient to be able to efficiently use the security mechanism.

- ◇ **Easy to Enforce**

  Thinks that web developers do not have problems when setting the CSP to enforce Trusted Types and that it is easier than enforcing sanitizationon its own.

- ◇ **Elevated Security**

  Thinks that Trusted Types help to improve the security of websites against client-side XSS.

- ◇ **Enforcement Problems**

  Sees a problem in writing a correct CSP header to enforce Trusted Types.

- ◇ **Hard to Refactor Existing Code**

  Thinks that it is hard to refactor existing web applications.

- ◇ **HTTPS needed**

  Thinks that developers might not be able to use Trusted Types as it requires HTTPS.

- ◇ **No Awareness**

  Thinks that there is no awareness of Trusted Types.

- ◇ **Possible Mistakes**

  Thinks that developer can make mistakes during the implementation of Trusted Types policies, resulting in an insecure policy.

- ◇ **Unsure to Use in Project**

  Is unsure whether to use Trusted Types in the next web project.

- ◇ **Usage in Project**

  Considers using Trusted Types in the next web project.

Roadblocks

◇ **3rd Party Code**

Struggles with third-party code during the coding task.

◇ **Bad Information Sources**

Thinks that information sources on the web are bad.

◇ **Copying Insecure Code**

Copies code that is declared as insecure on the webpage.

◇ **False Security**

Thinks that the code is secure, although it is insecure.

◇ **Focussing on Specific Vulnerability**

Focuses on a specific vulnerability instead of trying to fix client-side XSS in general.

◇ **Identical Policy Functions**

Implements an insecure Trusted Types policy because the same code is used for all sanitizer functions of the policy.

◇ **Insufficient JavaScript Knowledge**

Lacks knowledge of JavaScript to correctly deploy Trusted Types.

◇ **Little Framework Support**

Only a small number of frameworks support Trusted Types.

◇ **No CSP**

CSP is not set to enforce Trusted Types.

◇ **Unsure where to Start**

Has no idea where to start with the implementation of Trusted Types.

◇ **Website Breaks**

Implemented Trusted Types breaks the functionality of the website.

Strategy

◇ **Copy Code**

Copies code from online resources to use in the coding task.

◇ **Developer Console**

Uses the developer console in the browser to gather information about the web application.

◇ **No Scripts Dynamically Added**

Tries to disallow that any scripts are dynamically added/executed.

◇ **PolicyViolations Event Listener**

Registers the PolicyViolations Event Listener to be notified when JavaScript code violates the Trusted Types CSP header.

◇ **Same Code for all Sanitization Functions**

Wants to use same code for all sanitizer functions in the Trusted Types policy.

◇ **Set CSP**

Tries to set the CSP to enforce Trusted Types.

◇ **Vulnerability Script**

Tests the Trusted Types implementation by trying to exploit the vulnerability file.

◇ **Using Sanitization Library**

Uses a sanitization library to implement the Trusted Types policy.

Trusted Types

◇ **Default Policy**

Knows the special properties of a default policy.

◇ **Difference between Sanitizer Functions**

Knows the difference between the different sanitizer functions.

◇ **Enforcement**

Knows how Trusted Types need to be enforced.

◇ **Insecure Policy**

Knows that the shown example policy is insecure.

◇ **Mechanism**

Knows how the Trusted Types API works.

◇ **Mitigation Technique**

Knows how Trusted Types can protect against client-side XSS.

XSS

◇ **Attack Scenarios**

Can name possible attack scenarios that result from an XSS vulnerability.

◇ **Client-Side**

Has good knowledge of client-side XSS vulnerabilities.

◇ **Mitigation Mechanisms**

Knows XSS mitigation techniques and how they work.

◇ **Persistent**

Has good knowledge of persistent XSS vulnerabilities.

◇ **Reflected**

Has good knowledge of reflected XSS vulnerabilities.

◇ **Server-Side**

Has good knowledge of server-side XSS vulnerabilities.

◇ **Vulnerability Details**

Knows what am XSS vulnerabilities is.

Appendix C

Recruitment Mail

Dear former WebSecurity students,

Our Bachelor student Philipp is currently writing his thesis about Trusted Types. As part of this thesis, he is conducting a study on the usability of Trusted Types.
The study consists of two parts: First, an interview about XSS and Trusted Types. And afterwards, a coding task where Trusted Types sanitizer functions need to be created.
The whole process will last approximately 60 minutes, and you will be compensated with a 25€ Amazon voucher.
So if you are interested or have further questions, do not hesitate to write him an email.

Abbreviations

Acronym	**Meaning**
API	**A**pplication **P**rogramming **I**nterface
CSRF	**C**ross **S**ite **R**equest **F**orgery
CSP	**C**ontent **S**ecurity **P**olicy
DevOps	**Dev**elopment **Op**eration**s**
DOM	**D**ocument **O**bject **M**odel
HTML	**H**yper**t**ext **M**arkup **L**anguage
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
HTTPS	**H**yper**T**ext **T**ransfer **P**rotocol - **S**ecure
HTW	**H**ochschule für **T**echnik und **W**irtschaft
IDE	**I**ntegrated **D**evelopment **E**nvironment
IT	**I**nformation **T**echnology
JS	**J**ava**S**cript
JSON	**J**ava**S**cript **O**bject **N**otation
OBS	**O**pen **B**roadcaster **S**oftware
OWASP	**O**pen **W**eb **A**pplication **S**ecurity **P**roject
MDN	**M**ozilla **D**eveloper **N**etwork
URL	**U**niform **R**esource **L**ocator
VM	**V**irtual **M**achine
TT	**T**rusted **T**ypes
XML	e**X**tensible **M**arkup **L**anguage
XSS	**C**ross-**S**ite **S**cripting

List of Figures

2.1 Client-Side XSS Vulnerability 8

3.1 File Structure of the Django Application 24
3.2 Content of Coding Task Modules 25

4.1 Perception Distribution 36
4.2 Discourage Distribution 38
4.3 Workflow of the Participants 40
4.4 Roadblock Distribution 43
4.5 Trusted Types Implementation Guideline for Web Developers 45

List of Tables

2.1 Trusted Types Browser Compatibility [28, 29] 12

4.1 Categories of Deployment Strategies 39
4.2 Categories of Roadblocks 41

Listings

2.1 Trusted Types Default Policy 10
2.2 Trusted Types Enfocement 11
2.3 Trusted Types Policy Restriction 11
2.4 Trusted Types Compatibility Check 12
3.1 Insecure Trusted Types Default Policy (trusted-types.js) 20
3.2 Vulnerability Script (vulnerabilities.js) 21
3.3 Response Function (views.py) 21

Bibliography

[1] A. Wirfs-Brock and B. Eich, “JavaScript: The First 20 Years,” *ACM on Programming Languages*, 2020.

[2] J. Grossman, S. Fogie, R. Hansen, A. Rager, and P. D. Petkov, *XSS attacks: Cross Site Scripting Exploits and Defense.* Syngress, 2007.

[3] B. Stock, M. Johns, M. Steffens, and M. Backes, “How the Web Tangled Itself: Uncovering the History of Client-Side Web (In)Security,” in *USENIX Security Symposium*, 2017.

[4] P. Wang, B. Á. Guðmundsson, and K. Kotowicz, “Adopting Trusted Types in ProductionWeb Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study,” in *IEEE European Symposium on Security and Privacy Workshops.* IEEE, 2021.

[5] S. Stamm, B. Sterne, and G. Markham, “Reining in the Web with Content Security Policy,” in *International conference on World wide web*, 2010.

[6] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, “Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies,” in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[7] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, “CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[8] S. Roth, L. Gröber, M. Backes, K. Krombholz, and B. Stock, “12 Angry Developers-A Qualitative Study on Developers’ Struggles with CSP,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[9] S. Calzavara, A. Rabitti, and M. Bugliesi, “Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild,” in *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[10] K. Patil and B. Frederik, “A Measurement Study of the Content Security Policy on Real-World Applications,” *Int. J. Netw. Secur.*, 2016.

[11] S. Roth, M. Backes, and B. Stock, “Assessing the Impact of Script Gadgets on CSP at Scale,” in *ACM Asia Conference on Computer and Communications Security*, 2020.

[12] P. Wang, B. Á. Guðmundsson, and K. Kotowicz, “Adopting Trusted Types in ProductionWeb Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study,” in *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2021.

[13] Whatwg, “HTML Standard,” *https://html.spec.whatwg.org/*, [Online; accessed 23-April-2022].

[14] T. Berners Lee, “Information Management: A Proposal. Technical Report,” https://cds.cern.ch/record/369245/files/dd-89-001.pdf, 1989, [Online; accessed 23-April-2022].

[15] D. Raggett, A. Le Hors, I. Jacobs *et al.*, “HTML 4.01 Specification,” *W3C recommendation*, 1999.

[16] L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor *et al.*, “Document Object Model (DOM) Level 1 Specification,” *W3C recommendation*, 1998.

[17] Mozilla, “Introduction to the DOM,” *https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction*, [Online; Accessed: 23-April-2022].

[18] W3Schools, “JavaScript History,” *https://www.w3schools.com/js/js_history.asp*, [Online; Accessed: 23-April-2022].

[19] W3C, “Document Object Model (DOM) Level 1 Specification (Second Edition),” *https://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/DOM.pdf*, [Online; Accessed: 23-April-2022].

[20] S. Lekies, B. Stock, and M. Johns, “25 Million Flows Later - Large-scale Detection of DOM-based XSS,” in *ACM SIGSAC conference on Computer & communications security*, 2013.

[21] R. D. Kombade and B. Meshram, “CSRF Vulnerabilities and Defensive Techniques,” *International Journal of Computer Network and Information Security*, 2012.

[22] M. Steffens, C. Rossow, M. Johns, and B. Stock, “Don’t Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild,” *NDSS Network and Distributed System Security Symposium*, 2019.

[23] M. Heiderich, C. Späth, and J. Schwenk, “Dompurify: Client-Side Protection against XSS and Markup Injection,” in *European Symposium on Research in Computer Security*, 2017.

[24] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna, “Client-Side Cross-Site Scripting Protection,” *Computers & Security*, 2009.

[25] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, “Precise Client-Side Protection against DOM-based Cross-Site Scripting,” in *USENIX Security Symposium*, 2014.

[26] auth0, “Securing SPAs with Trusted Types,” *https://auth0.com/blog/securing-spa-with-trusted-types/*, [Online; Accessed: 23-April-2022].

[27] W3C, “Trusted Types Standard,” *https://w3c.github.io/webappsec-trusted-types/dist/spec/*, [Online; Accessed: 04-May-2022].

[28] Mozilla, “CSP: require-trusted-types-for,” *https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-trusted-types-for*, [Online; Accessed: 04-May-2022].

[29] ——, “CSP: trusted-types,” *https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/trusted-types*, [Online; Accessed: 04-May-2022].

[30] K. Kotowicz, “Prevent DOM-based Cross-Site Scripting Vulnerabilities with Trusted Types,” *https://web.dev/trusted-types/*, 2020, [Online; Accessed: 25-May-2022].

[31] ——, “Trusted Types-Mid 2021 Report,” 2021, [Online; Accessed: 23-April-2022].

[32] Django-CSP, "Implementing Trusted Types with CSP," *https://django-csp-test.readthedocs.io/en/latest/trusted_types.html*, [Online; Accessed: 14-July-2022].

[33] R. B. Johnson and L. Christensen, *Educational Research: Quantitative, Qualitative, and Mixed Approaches.* SAGE Publications, 2019.

[34] R. Kang, L. Dabbish, N. Fruchter, and S. Kiesler, ""My Data Just Goes Everywhere:" User Mental Models of the Internet and Implications for Privacy and Security," in *Symposium on Usable Privacy and Security (SOUPS 2015)*, 2015.

[35] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun, "Home is Safer than the Cloud! Privacy Concerns for Consumer Cloud Storage," in *Symposium on Usable Privacy and Security*, 2011.

[36] K. Krombholz, K. Busse, K. Pfeffer, M. Smith, and E. Von Zezschwitz, ""If HTTPS Were Secure, I Wouldn't Need 2FA" - End User and Administrator Mental Models of HTTPS," in *IEEE Symposium on Security and Privacy*, 2019.

[37] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *International Conference on Software Engineering*, 2006.

[38] OWASP, "OWASP Top 10," *https://owasp.org/Top10/*, [Online; Accessed: 14-July-2022].

[39] Securityaffairs, "Companies Paid $4.2M Bug Bounties for XSS Flaws in 2020," *https://securityaffairs.co/wordpress/110223/reports/xss-top-bug-bounty.html*, 2020, [Online; Accessed: 12-May-2022].

[40] A. Strauss and J. Corbin, *Grounded Theory: Grundlagen Qualitativer Sozialforschung.* Beltz, Psychologie-Verlag-Union, 1996.

[41] IndianScribes, "3 Examples of Transcribed Interviews," *https://www.indianscribes.com/example-transcript/*, 2018, [Online; Accessed: 06-June-2022].

[42] S. H. Khandkar, "Open Coding," *University of Calgary*, 2009.