Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor Thesis

SynthTT: Jamming Client-Side XSS with synthesized TrustedTypes sanitizers

submitted by

Daniel Emmel

on March 07, 2021

Reviewers

Dr.-Ing. Ben Stock

Dr. Giancarlo Pellegrino

Disclaimer

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, der 07. März, 2021,

(Daniel Emmel)

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, der 07. März, 2021,

(Daniel Emmel)

Acknowledgements

First, I would like to thank Dr. Ben Stock for providing me with the unique chance to write my thesis in the Secure Web Applications Group as well as to work as a research assistant in the area of Web Security. Also, I want to give thanks for his valuable advice and feedback that he provided me with during the working on my thesis.

Furthermore, I want to thank Marius Steffens for his advice on some of the technical problems that I faced and general feedback considering the writeup of my thesis, as well as proofreading. Additional thanks for proofreading my thesis also go to my peer Moritz Wilhelm.

Lastly, I would like to give thanks to all my family members and friends who supported me during the stressful time I had with my thesis and my studies in general, especially my mother Astrid, my brother Marius, and my best friends Tobi und Robin who in particular took their time to support me during my working on this thesis.

Abstract

With the great importance and popularity of the web to today's world, it is an unfortunate truth that there exist many malicious entities who try to take advantage of common web vulnerabilities to make profit from damaging innocent users.

One of the most common web vulnerabilities is client-side Cross-site-scripting (XSS), which can occur when a website's code uses dangerous, JavaScript executing sinks with user-controllable input, as an attacker may also be able to manipulate the executed value. This vulnerability in particular can be devastating to the user, as it allows an attacker to execute arbitrary JavaScript in the authorization context of the user, potentially allowing leakage of sensitive user data.

One new proposed mitigation to this problem is called `Trusted Types`. Its main idea is that each input that a sink is called with must pass through a `Trusted Types` sanitizer, where it can be filtered to no longer be problematic. This idea, however, raises the immediate challenge that third-party content in the wild will often write a lot of dynamic HTML, JavaScript and URLs into the main site's document via the problematic sinks. Still, since this usage is benign, the `Trusted Types` sanitizer should allow it, while non-intended sink inputs should be prohibited to prevent an attacker from executing JavaScript at their will.

In this thesis, we enforce the principle of least privilege on third-party scripts by auto-generating policies for them based on their previously observed benign sink usage behavior. In particular, we collected sink input data from the top 100 Tranco domains and generated configs that describe what sink inputs should be allowed in the future. Further, we built a library to enforce these configs on a given site and evaluated its functionality-preservation as well as its overhead. We found that over a period of 10 days, functionality was preserved well, never going below 94%, and that the runtime overhead exhibited was generally low, being less than 4% for the average load time of a site. Overall, we show that `Trusted Types` is indeed feasibly usable in an automated manner to reduce the attack surface exposed by third-party code on a site.

Contents

Chapter 1

Introduction

In today's time, the web has become extremely important in our everyday life. It provides users with many different services, some of which are extremely sensitive, for instance online banking applications. Oftentimes, these modern applications try to be as feature-rich as possible, making them increasingly complex [42]. To this end, they make use of the scripting language JavaScript, which allows for much more dynamic interactions with users. However, the existence of such sensitive applications and the data of users that they contain naturally provides an interesting incentive for malicious entities to make profit from. In order to protect a site from such malicious accesses, browsers feature a security mechanism called the Same-Origin-Policy (SOP). However, if the JavaScript running in a site makes use of one of the APIs that allow for executing strings as code, called sinks, and such a sink input is attacker-controllable, the SOP can be circumvented since that code is executed in the site's own origin. This yields an attacker the possibility to execute arbitrary code on the vulnerable site. The described vulnerability is one of the two major classes of Cross-Site Scripting (XSS), called *client-side XSS* [30].

Additionally to the threat of an application's own code being vulnerable, modern applications also often include third-party script content [34] to provide more functionality to the user or display advertisements to gain profit for the developer. However, since this code also runs in the same origin as any of the site's own native code, it is not restricted by the SOP. Therefore, a client-side XSS vulnerability present within any of the third-party code can lead to compromise of the application even if its own code is completely safe. Unfortunately, simply attempting to sandbox the third-party code like prior work attempted [43] [44] [28] [45], or restricting its

ability to add new scripting content [33] often interferes with the code's functionality. For instance, advertisements may wish to interact with the site's Document Object Model (DOM) to verify that they are actually being displayed properly, which may not work if the code is sandboxed. Another possible mitigation mechanism that could be considered is the Content Security Policy (CSP), which allows specifying the sources from which JavaScript is allowed to be executed on the site, as well as the allowed sources for other content. However, many real-world CSPs are insecure due to the site's reliance on third-party script content [36] [46].

Another novel mitigation mechanism proposed by Google is `Trusted Types` [26]. The main idea behind this mechanism is that the dangerous sink APIs can no longer be called simply with plain strings. Instead, the inputs must first pass through a `Trusted Types` policy, in which they can be arbitrarily sanitized or even rejected completely. Thus, the problem of defending against client-side XSS boils down to constructing secure but functionality-preserving sanitizers. However, these sink inputs may depend on dynamic parameters, and thus it is hard for the first-party site to predict what inputs the third-party code might call sinks with.

In this work, we attempt to automatically synthesize secure and functionality-preserving `Trusted Types` policies based on observed benign sink inputs on a site. In particular, we use the benign inputs to extrapolate a set of rules that inputs for a given party must fulfill, storing these rules in so-called *configs*, and then construct a client-side library which enforces the rules from the configs on any new sink access on the site. We thus aim to reduce the attack surface as much as possible by enforcing the principle of *least privilege* on the third-party code while preserving as much functionality as possible. Apart from implementing the described infrastructure, we also perform an evaluation of its preservation of code functionality, initially and over time, as well as the overhead that the client-side library introduces on a site when present using data collected from the top 100 sites of the Tranco list [17]. Additionally, we provide insights about the collected data and the generated configs and discuss the achieved reduction in attack surface.

1.1 Related Work

Client-side XSS is not a new problem, but its origin rather reaches back to about 2005 [30]. Hence, much research on the vulnerability and potential mitigations has already been done. In the following, we explore some interesting work related to the relevance of XSS in general and how often third-party code was found to be the origin of a flaw specifically, as well as some strategies existing work proposed to mitigate the threat of client-side XSS and if appropriate compare them to our general idea. Additionally, we present work related to problems with allowlisting benign JavaScript samples in a secure way, as these problems and solutions are also very important for our own work. For any unclarities concerning technical terms, we refer to chapter 2.

1.1.1 Relevance of client-side XSS

Many different researches have previously shown that client-side XSS is a dangerous threat introduced by flaws in first-party and third-party code alike. Stock et al. [40] gathered 1273 real-world vulnerabilities from the top 10k Alexa domains. By gathering information about the JavaScript that introduced the vulnerabilities, they were able to measure the complexity of each individual flaw. They found that while about two-thirds of the flaws were very simple and likely accountable to insufficient security-awareness of the developers, about 15% were much more complex, indicating that some vulnerabilities are introduced due to code complexity rather than missing awareness. Furthermore, 273 vulnerabilities originated from third-party code, while another 165 came from a combination of first-party and third-party code, further displaying the significance of third-party code in introducing client-side XSS vulnerabilities.

Steffens et al. [38] specifically investigated the prevalence of persistent client-side XSS, a subclass of client-side XSS which abuses flows from persistent client-side storage, such as cookies or the `localstorage`, to sinks to repeatedly trigger the payload after it has infected the client-side storage once. They performed an analysis on the top 5000 Alexa domains using a taint-aware browser and found that more than 8% of these sites made use of flows from vulnerable storage to dangerous code-execution sinks, out of which they found 70% to be exploitable. Furthermore, they found that 21% of the considered sites made use of data from

client-side storage in their code, which indicates that developers assume a certain trust from saved data even though it may be malicious or compromised during storage.

Lekies et al. [31] used a fully-automated approach to collect 6167 unique client-side vulnerabilities distributed across 480 of the top 5000 Alexa domains. In particular, this means that at the time of their study, almost 10% of these domains showed some kind of client-side XSS vulnerability. Five years later, Melicher et al. [32] improved upon the methodology of Lekies et al. [31], being able to find 83% more flaws on the same dataset. They found that in comparison to previous results, there were actually *more* vulnerable flows, implying that the impact of client-side XSS becomes higher rather than lower over time. Further, they determined that 83% of flaws are introduced by code from analytics or advertising domains, and that flaws are generally concentrated on a small number of script hosting sites and iframe owners, which again emphasizes the impact of third-parties on client-side XSS flaws.

1.1.2 Defending against Vulnerabilities in Third-Party Code

Since the vulnerabilities present in an application are oftentimes not their own fault but rather that of the included third-party code, many ideas to reduce the privileges of third-party code, some of which are similar to our approach, have been proposed.

Musch et al. [33] implemented a tool named `ScriptProtect` assuming a similar threat model as our work, with the goal to restrict included third-party code in its ability to access the dangerous JavaScript executing sinks while still trying to preserve functionality as best as possible. In particular, their tool acted as a reference monitor between code and sinks by instrumenting all dangerous APIs and then sanitizing any input coming from a third-party source. They also compared their approach to the `Trusted Types` mechanism, which we make use of in this work. They pointed out that usage of `Trusted Types` would necessitate rewriting third-party code. However, due to the recently introduced `default` policy which automatically intercepts all calls to the sink APIs, this is no longer the case. The biggest difference of our approach to `ScriptProtect` is the following: `ScriptProtect` determines whether or not a sink call occurred from a third-party, and if so, simply sanitizes the input of any additional script content. In contrast

to that, we first determine via the construction of configs from gathered data what script content a party should be allowed to be adding to the document, meaning that we preserve the functionality of the third-party sink accesses while `ScriptProtect` by its design does not.

Stock et al. [39] proposed to combat client-side XSS flaws in the browser's JavaScript execution mechanisms themselves. In particular, they implemented an enhanced version of the JavaScript engine that possesses the ability to track vulnerable code injection flows, as well as a modified version of the Hypertext Markup Language (HTML) and JavaScript parsers that are able to identify code that originates from such problematic flows. While their implementation showed a false positive rate of below 2%, it also showed a non-negligible performance overhead of between 7% and 17%, though they emphasize that a potential efficient native implementation could significantly lower this overhead. In contrast to this approach, our design does not need to modify the basic workings of any of the browser's own parsers, but we rather use the specially designed `Trusted Types` mechanism to disrupt any malicious flows.

1.1.3 Content Security Policy

CSP is a XSS mitigation mechanism well-known across researchers. However, CSPs deployed in the real world are often found to be lacking in security. Roth et al. [36] collected deployed CSPs from 10000 highly-ranked domains over seven years obtained from the Internet Archive [1], finding that most CSPs provide insufficient protection against script injection and are next to trivially bypassable. They additionally performed surveys with developers, finding that most of them are discouraged from using CSP since it appears too complex to them. They concluded that CSP as a mitigation to XSS largely fails in the real world.

Weichselbaum et al. [46] also performed a study on real-world CSPs, but on an even larger scale. They found about 26000 unique policies across about 1.6 million hosts that used a policy. Overall, they tested about 100 billion sites, showing that only a small portion deployed a CSP at all. Out of the collected policies, over 94% were found to be trivially bypassable. Additionally, about 75% of all policies were found containing a host-based allowlist with at least one vulnerable host.

1.1.4 Constructing allowlists for benign JavaScript samples

Using observed benign sink API inputs to construct appropriate allowlists for future sink accesses is a non-trivial task, since the lists must be sufficiently strict to provide security but also permissive enough that as many new benign inputs as possible are not accidentally blocked. Hence, simply constructing a list out of all observed inputs and then just allowing these is insufficient, as the sink inputs may be generated dynamically and not every possible variation may have been observed. Instead, one might consider the notion of *structural hashes*, where the pure syntactical structure of the inputs is allowlisted and any new input matching one of these structures is allowed. While good for preserving functionality, the security provided by this mechanism is flawed. Fass et al. [29] showed that a generic attack against such syntactical allowlisting can be constructed. In particular, given a set of allowed Abstract Syntax Trees, their system was able to rewrite malicious samples in such a way that they produced semantically equivalent samples that possessed one of the allowed syntactical structures. They were able to perform the rewriting in practice for over 91000 malicious scripts, reproducing benign ASTs from the top 10000 Alexa pages. Additionally, their approach showed a very low detection rate, making it effective. Given the possibility of such a generic camouflaging attack, it becomes clear that allowlisting relying on structure alone is not sufficient.

Instead, Stock et al. [41] used a different approach. Their goal was to generate reliable anti-malware signatures from given malicious JavaScript samples that would allow identification of newly found malicious samples. To this end, they clustered the samples by their token structure and searched clusters for common token substrings. From these substrings, they generated regexes that matched the observed token values as well as variations of them if different values for a particular token were observed across the common tokens. They showed that the false positive rate of this approach was well below 1%, while the false negative rate was below 5% as well. This implies that the approach is well-suited for the problem of classifying new JavaScript samples from observed ones. Hence, we decided to implement a similar approach for deciding whether or not a new JavaScript input is benign based on previously seen benign ones.

Chapter 2

Background

In the following chapter, we explain all the basics necessary to understand the rest of our work. In particular, we present basic web technologies, as well as the most basic security mechanism in browsers called Same-Origin-Policy, and the threat of client-side Cross-Site Scripting which circumvents this mechanism. Further, we present all details about the `Trusted Types` mechanism necessary to understand our infrastructure.

2.1 Web Basics

First, we want to clarify some of the most basic workings of the web to establish appropriate background knowledge.
Let us consider that we are using any browser of our choice, and we wish to visit `https://example.com`. When navigating to that site, our browser will make a Hypertext Transfer Protocol [35] request for us, requesting the site's content from the *server* that is hosting the said content of `https://example.com`. This Hypertext Transfer Protocol (HTTP) request may contain arbitrary metadata, given in *headers*. One of the most important headers is the `cookie` [27] header, with which so-called *cookies* are sent. These are set in the browser using the value from the `set-cookie` header of a server's response after a user has successfully logged in. When a user then visits that site again, the browser will attach any cookies it has for that site to the request, which will automatically re-authenticate the user such they won't have to input their credentials again.
After the server has processed our request, it will return a HTTP response which

includes the actual content of the site in Hypertext Markup Language [25] format. The browser will then render the HTML response and display it to the user. Such HTML files are structured in so-called *elements*. Each element essentially represents one entity within the site. This may be anything from simple text, to images, or even videos and *iframes*, which are used to display the content of another site within a part of our own site. These elements consist of their actual content, as well as possibly many different attributes. An element can have other elements as children (and thus also as parents), which causes HTML documents to be displayed as a tree in memory, the so-called Document Object Model.
In order to enable websites to dynamically interact with the user, browsers allow usage of a scripting language called *JavaScript* [23]. Scripts of this language can interact with the DOM to for instance react to user input in a dynamic manner. In general, it can arbitrarily react to events happening in a site as well as modify the appearance of the site's context that it is running in.

```
<img src="https://site.com/img/logo.png" id=42
onload="alert('Image loaded!')">
```

Listing 1: A simple HTML example snippet

Listing 1 provides a small example of an HTML element's definition. In particular, it defines an `image` element, which will attempt to load the image hosted at `https://site.com/img/logo.png` and then display it on our current site. Further, we have given the image element the id 42, which makes it easy to access it via JavaScript later. We further observe an `onload` attribute, which is an example of a so-called event handler. Once the image has finished loading, the JavaScript that is assigned to that attribute is executed. In this case, an alert box will pop up, containing the text `'Image loaded!'`. Such event handlers are one way to execute JavaScript in our site. Additional ones are displayed in Listing 2.

Within Listing 2, we define an `anchor` element. Such elements represent a link to some site which when clicked will redirect to the URL given in the `href` attribute. Should this attribute contain a `javascript:` URL, however, it will instead execute the specified JavaScript. The listing also shows a `script` tag with a `src` attribute that contains a `data:` URL. Such URLs represent inline data to be used in a context, in this particular case, in the context of a script. Thus, the content of the `data:` URL, which may optionally be base64 encoded [5], is processed and in this case executed as a script.

```
<!-- inline script tags -->
<script>
alert("I am a script tag!");
</script>
<!-- external scripts -->
<script src="https://site.com/js/script.js"></script>
<!-- JavaScript URLs -->
<a href="javascript:alert('I am a JavaScript URL')">
<!-- data URLs as a script's attribute -->
<script src="data:text/javascript,alert('I was in a data URL')"></script>
```

Listing 2: Examples of how to execute JavaScript within HTML

Now, recall that many sensitive applications run on the web, and JavaScript can perform essentially arbitrary operations on a website. Therefore, it might be possible for an attacker to include their target site in their own website as an `iframe`, and then attempt to, for instance, install a keylogger in that site via JavaScript to steal the user's credentials for that site. However, that is not trivially possible due to the most basic security policy present in all web browsers: The Same-Origin-Policy. This policy dictates that one document may only directly read or write content from another if they are of the same *origin*. Here, origin means having the same *protocol*, *hostname* and *port* in the URL. Let us again display this with a small example:

URL of site	Protocol	Hostname	Port
https://site.com/	https	site.com	443 (default https port)
https://attacker.com/	https	attacker.com	443

Table 2.1: A simple example of the Same-Origin-Policy

As table 2.1 shows, site one would not be able to access content on site two directly (and vice versa), since while they do share the same protocol and port, their hostnames differ, and thus the browser would refuse access via JavaScript.

2.2 Client-side XSS

While the SOP does prevent the direct access between two cross-origin documents, there is a scenario in which it can be bypassed. Let us consider the situation that the following JavaScript shown in Listing 3 is present within a given site.

```
let username = unescape(location.hash.slice(1));
document.write("Welcome back, " + username + "!");
```

Listing 3: An example of code vulnerable to client-side XSS

This could be part of a simple script which might be intended to greet the user after they have logged in again. The JavaScript would expect to read a username from the URL's fragment [1]. Now, let us consider the following two URLs presented in table 2.2 being visited by a browser, and the respective values that end up being written into the DOM.

URL being visited	Result added to site
https://vuln.com/user.html#Daniel	Welcome back, Daniel!
https://vuln.com/user.html# <script>alert("Evil code injected!")</script>	Welcome back, <script> alert("Evil code injected!")</script>!

Table 2.2: Benign vs vulnerable input into a XSS-vulnerable sink call

In the first case, the `document.write` would simply add the string to the HTML document and it would be displayed to the user. The problem with this code is that `document.write` can add *arbitrary* HTML to the site. So, if it is given a script tag as a string like in the last line of table 2.2, then that tag would be added to the document and executed in the site's context, with the same privileges as any other already existing script. In this particular case, we have simply inserted a script that opens another alert box. Naturally, this does not have to be a simple `alert` call, but can be any arbitrary JavaScript, thus having achieved *arbitrary code execution* in the vulnerable site. Since this JavaScript is now running in the site's *own origin*, it is not blocked by the SOP, and can do whatever it wants with the site.

To actually abuse this properly against a specific user, an attacker could send them a link like `https://vuln.com/user.html#<script>evilcodehere</script>`. Once clicked, it would cause the user to be directed to the vulnerable site, the attacker's code to be included in the site due to the vulnerable JavaScript code, and then executed. One may abuse this flaw to for instance install a *keylogger* to steal the user's inputs on the site, like credentials or other interesting information. In essence, the site's content can be almost arbitrarily modified and extracted.

[1] The fragment of a URL is the part after the hash symbol, e.g. for the URL `https://example.com#value`, the fragment would be `value`.

This class of vulnerability is called XSS, since it opens up the possibility to include attacker-controlled JavaScript not coming from the website itself, which would normally be blocked by the SOP. In particular, the example shown in table 2.2 is an instance of *Client-Side* XSS, since the vulnerable code that caused the attacker's code to actually be included in the site was part of the client-side JavaScript. This is the specific kind of XSS we will consider in this work, as `Trusted Types` only aims to mitigate this flavor of XSS [19]. The alternative is called server-side XSS, where the vulnerable server code (e.g. PHP code) includes attacker-controllable input parameters in the HTML that it will answer to a browser's request.

As previous work has found, this is a very critical vulnerability in the wild [31], so there have been quite a few ideas for mitigating this threat. In the following section, we have a look at the most important one for this work in detail: `Trusted Types`.

2.3 Mitigating client-side XSS with Trusted Types

`Trusted Types` [26] is a rather new mitigation mechanism for client-side XSS. It was proposed by koto [12] and is currently an editor's draft, though it is already implemented in Chrome as well as Edge since version 83, and Opera since version 69. [24]

The basic principle behind `Trusted Types` is as follows. There is a limited number of dangerous sinks in JavaScript that can lead to the inclusion of attacker-controllable inputs into the site and then executing them as JavaScript. One of those is the `document.write` API, of which a usage example can be seen in Listing 3. The idea of `Trusted Types` is that these dangerous sinks cannot be called directly with string inputs anymore. Instead, it becomes necessary for all these inputs to first be sanitized by `Trusted Types` sanitizers, which can change the input or reject it entirely. Thus, the problem of detecting malicious inputs becomes centralized in these sanitizers, and not every single possible dangerous sink input must be checked separately anymore. Therefore, assuming we have a secure sanitizer, our site will only allow non-malicious inputs to the dangerous sinks, and hence no malicious, attacker-controllable code is executed anymore.

2.3.1 Defining and Using a Basic Trusted Types Policy

In order to make use of `Trusted Types`, we must first define a *policy*, which then will contain the actual functions that are used to sanitize any given sink input. A simple policy may look like the following:

```
let policyName = 'dummyPolicy';
const policy = trustedTypes.createPolicy(policyName, {
  createHTML: function (input) {
    if(isAllowed(input))
      return input;
    else
      return null;
  }
});
```

Listing 4: Simple example of defining a policy

In order to define a policy, we simply call `trustedTypes.createPolicy` with two arguments: first, the name of the policy (here: `dummyPolicy`) and as the second value an object containing the sanitizer functions. Here, the only sanitizer function we use is `createHTML`. This is the function that we will call to sanitize any inputs that are later interpreted as HTML markup by the parsers, like a call to e.g. `document.write` causes it to occur. This function in particular will call another function called `isAllowed`, which by some rules will determine whether this particular input should be allowed to be used in the `document.write` call and if so, the policy returns said input. Otherwise, it will return `null`, which indicates to the `Trusted Types` implementation that the input was not allowed, and the sink call will fail.

Now, assuming we enforce `Trusted Types` in our site, any calls to dangerous sinks with normal string types are disallowed. Instead, one can only call sinks with values of one of the `Trusted Types`, which can only be obtained by calling the corresponding sanitizing function from a `Trusted Types` policy, like the one above.

Listing 5 gives a comparison between an allowed and a disallowed sink when enforcement of `Trusted Types` is active. The first call to `document.write` is not allowed, since we call the sink directly with just a normal string, and not a `Trusted Type`. Contrary, the second call is allowed since we invoked `createHTML` with the input and it was deemed allowed, yielding us an object of type `TrustedHTML`. We then call `document.write` with this object, which causes our input to be written

```
// Insecure/disallowed call:
document.write('I am not allowed');

// Allowed call (assume the policy allows this input):
const sanitized = policy.createHTML('I work fine');
// next line prints true
console.log(sanitized instanceof TrustedHTML);
document.write(sanitized);
});
```

Listing 5: Allowed vs disallowed sink inputs when enforcing a policy

to the DOM as normal. Thus, any sink calls present in an application must be rewritten to first sanitize the inputs with the policy and afterwards call the sink with the result, assuming the policy allowed the particular input.

2.3.2 Trusted Policy Subtypes

There are three different ways how a dangerous sink might lead to execution of its input as JavaScript. The scripting code may be part of HTML markup inserted into the site, or it may be executed directly, or the code might be loaded from an external resource. `Trusted Types` allows for separate handling of these three cases. When specifying a policy, one can define three different functions to sanitize inputs for each of the sink subtypes. We now present each of these sink subtypes and how to build a policy for them.

As displayed in Listing 3, a sink might add its input as HTML markup to the current document's DOM, and any script tags and event handlers added are then executed by the JavaScript engine. An example of a function which does this is `document.write`. If there is a `Trusted Types` policy active on our site and its use is enforced, we can only call these HTML writing sinks with objects of type `TrustedHTML`, which we may only get from a policy's `createHTML` function.

The second sink subtype directly executes its input as JavaScript without having to add it to the DOM. An example of this is the `eval` function. With the use of `Trusted Types` enforced, we may only call these sinks with objects of type `TrustedScript`, which are received from a policy's `createScript` function.

The last subtype programmatically loads JavaScript from an external resource, and causes this external code to be executed.

```
let newScript = document.createElement("script");
newScript.src = "https://site.com/js/script.js";
document.body.appendChild(newScript);
```

Listing 6: Creating a script tag via JavaScript

Listing 6 gives a small example in which JavaScript is used to programmatically generate a new script tag. The `src` attribute of this new script is set to `https://site.com/js/script.js`, meaning we tell this script tag to load an external script from that `src`. The last line adds this new script to the DOM to have it executed. So, we have *programmtically* included script content from another site in our site. Naturally, this can also be very dangerous, since an attacker could try to load a script hosted on his own site. Therefore, we may only set the `src` attribute of elements which can lead to execution of external JavaScript to objects of type `TrustedScriptURL`. In order to obtain an object of this type, we need to define a `createScriptURL` function in our policy and call this function with the desired input.

2.3.3 Enforcing Trusted Types on a Website

Next, we present how to enforce `Trusted Types` on a website to ensure that sinks cannot be called with non-sanitized inputs.

For its enforcement, `Trusted Types` makes use of another security mechanism of the web called Content Security Policy [22]. CSP is essentially a set of directives that a server can include in its response to tell the browser which resources a site is allowed to use. For instance, it is used as a mitigation to XSS, since it can restrict the sources of scripts. It may be configured in `report-only` mode, a mode in which violations of any given directive will not block the action, but only generate a log. `Trusted Types` makes use of a special CSP directive in its enforcement. In particular, there are two new directives to control usage of `Trusted Types` in a site. The first one is used to actually enforce the usage, like shown in Listing 7.

```
Content-Security-Policy: require-trusted-types-for 'script';
```

Listing 7: The basic `Trusted Types` enforcement CSP

With this header in place, the site must make use of a `Trusted Types` policy if it wishes to use the dangerous sinks discussed before. Additionally, it is possible

to restrict the allowed names of policies via the `trusted-types` CSP header, as shown in Listing 8.

```
// Allow only policy names foo and bar
// only allows ONE policy with these names
Content-Security-Policy: trusted-types 'foo' 'bar';

// Allow names foo and bar but arbitrarily often
Content-Security-Policy: trusted-types 'foo' 'bar' 'allow-duplicates';

// Allow NO policies
Content-Security-Policy: trusted-types 'none'; require-trusted-types-for
↪  'script';
```

Listing 8: Using CSP to restrict policy names

If we have `Trusted Types` enforced, and we allow no policy names, this means that we cannot define policies anymore, making it impossible to obtain any kind of `Trusted Type` object. However, since we are only allowed to call dangerous sinks with objects of said type, we *cannot* use sinks anymore. So, we can use this directive to block any kind of sink usage altogether.

There remains one large problem for the enforcement of `Trusted Types` on a real site. Many sites make use of *third-party scripts* [34] for things like advertisements, or better functionality.[2] Recall that we need to rewrite *all* usages of sinks, as those that we would not rewrite would simply not work anymore. This is problematic since third-party scripts are executed in the context of our own site, but their content is taken from an external source which we usually do not control. Hence we cannot rewrite their content such that they properly call a policy to sanitize their input, and thus `Trusted Types` will refuse any kind of sink usage of the third-party scripts, which means that likely, their functionality will break (assuming they do not make use of a `Trusted Types` policy, which we cannot universally assume). For the first-party site, that implies that e.g. no more advertisements will run, defeating the purpose of including the script content in the first place.

Next, we discuss a solution to the described problem.

[2]For instance, JQuery [9] is a JavaScript library which allows easier interaction with the DOM.

2.3.4 Making Third-Party Scripts Work Again: The Default Policy

There is a mechanism in `Trusted Types` which allows for automatic interception of all sink accesses, the *default policy*. Consider the following example of a default policy, as shown in Listing 9.

```
const policy = trustedTypes.createPolicy('default', {
createHTML: function (input, type) {
  if(isAllowed(input))
    return input;
  else
    return null;
}
});
```

Listing 9: A small example of the default policy

If we compare this to the simple policy shown at the beginning of section 2.3.1, then we notice two distinct but subtle differences. The *name* of this policy is *default*. That is the name any default policy must have. Furthermore, the sanitizing functions are not only passed the input to sanitize, but also the name of the `Trusted Type` they return. For example, `createHTML` is given `TrustedHTML` as an additional parameter.

Thus, if one deploys such a *default* policy, then there is no need to rewrite any other code that makes use of sinks, and in particular, no need to rewrite third-party scripts.

Chapter 3

Methodology

In the following chapter, we explore the methodology that we employed to enforce the principle of least privilege on third-party code to reduce the attack surface exposed by said code. In particular, we first have a deeper look at the attacker model and general assumptions, and then give an overview of our infrastructure, after which we explore each of its components in detail.

3.1 Attacker Assumptions

In this thesis, we assume that we are dealing with web applications that include third-party code that is *in itself benign but buggy*. Hence, they potentially contain code that calls some of the dangerous sinks that allow JavaScript code execution from attacker-controllable code but the third-party code does not perform any malicious actions by itself. Further, we expect that an attacker can control *all* inputs that third-party code calls sinks with, i.e. they can trigger client-side XSS in all sink accesses. Naturally, this is a complete overestimation, but it is safest to assume this for simplicity, as otherwise, we would need to determine which sink accesses are vulnerable and which are not, which is very much a non-trivial task. Furthermore, we assume that the attacker can only control *one consecutive sink access* at a time. This is also a simplifying assumption which we make as otherwise our design would become unnecessarily complicated for the context of this thesis.

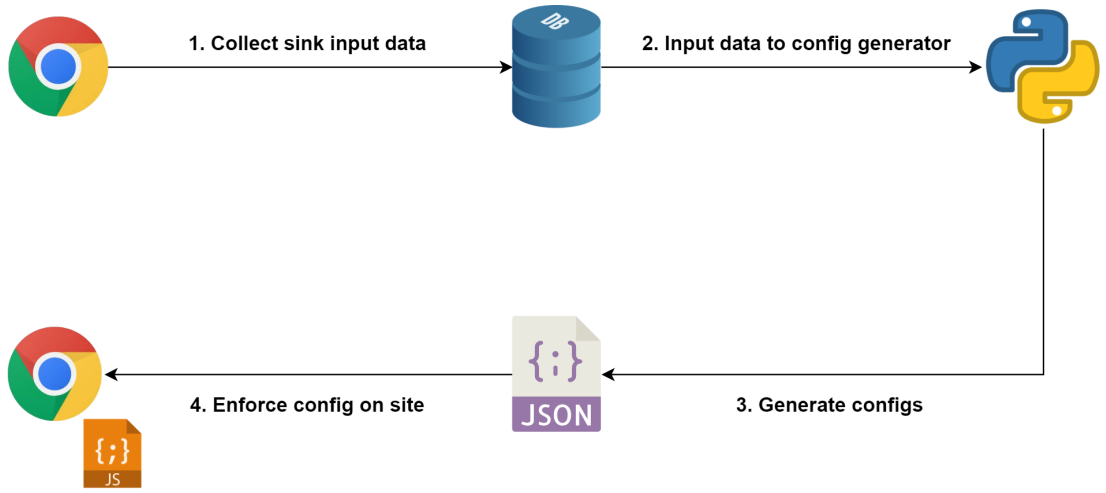

Figure 3.1: High-level overview of the infrastructure

3.2 Infrastructure Overview

A general flowchart-like overview of our infrastructure can be seen in figure 3.1. To reach our goal of enforcing least privilege on third-parties using `Trusted Types`, we use multiple building blocks. First, it is necessary to know what inputs the parties operating on a given site use to call the problematic sink APIs, as we must ensure that the generated policy will still allow all these benign inputs. Thus, we collect such inputs and store them in a database. In the next step, we use the collected data to generate a *config* for the given site. Such a config contains a mapping that specifies what inputs a given party is allowed to call a sink with and is generated per site. It can then be used as input to the `SanitizerLibrary`, which is a client-side library that we implemented to enforce configs on a given site using the `Trusted Types` default policy.

3.3 Data Collection

The data collection approach makes use of the crawling infrastructure provided by the `Secure Web Applications Group` [16]. A part of the infrastructure's codebase has since been made public [13] after its use in the `PMForce` paper [37]. Given a set of sites, a version of Chrome instrumented by the puppeteer framework [14] visits them one after another and executes the code of a pre-configured `crawler module` on them. This module may interact with the site and the browser in various ways, e.g. intercepting HTTP requests or injecting custom JavaScript into the site, as

the module can also make use of the Chrome Devtools Protocol [3]. In particular, it can do this *before* the request for the site's content is done. The crawler may potentially follow links present on a site up to a certain depth. This, as well as other crawling parameters, e.g. site load timeout, are determined via a config file. To determine the set of targets the crawler should visit, we chose to use the Tranco domain list [17], as it provides a reliable ranking of the most visited and thus most relevant webpages. Additionally, we use the `Trusted Types` mechanism itself in the data collection. Since each sink access triggers the default policy, we can use a simple no-op logging policy to collect the sink inputs.

3.3.1 Crawler module

Once the crawler has initiated visiting a site, our module will intercept the request for the actual content of the site. It then manually retrieves the response and modifies it: any CSP header present is removed and a new CSP is added instead:

```
Content-Security-Policy-Report-Only: require-trusted-types-for 'script';
```

Listing 10: The CSP used during data collection

This header ensures that `Trusted Types` is enforced on the site and each sink access triggers the default policy. Since it is configured in `report-only` mode, it additionally does not block functionality of the site should no policy be present.

Furthermore, the module injects a script tag containing a simple logging default policy into the site's content that allows any input and thus does not hinder the sink call execution. Its purpose is the collection of all sink inputs together with information about the context in which it was observed. In particular, this script tag containing the policy is added at the beginning of the `head` tag to ensure that it is executed before any other JavaScript code, and thus that the policy is present for all sink accesses on the site. However, this is not sufficient. An `iframe` may be constructed by the site whose `src` attribute is a `srcdoc` URL. Such an `iframe` inherits the CSP we injected [4], but since its content is determined by the code running on the visited site, it does not automatically have the logging policy present in its own code, even though these frames still belong to the context of the original site they run in and thus may contain interesting sink accesses. To mitigate the data loss occurring from this, we hooked any function that allows creation of an iframe and manually injected the logging policy.

Once the logging policy performs a log, it is intercepted by the crawler module, which then extrapolates data about the sink-calling party from the stacktrace, and proceeds to store the collected data in a database. In particular, the following information about the sink access is saved:

- The URL, origin and etld+1 of the site in which the sink input occurred
- origin and etld+1 of the party which called the sink
- the input given to the sink
- the `Trusted Type` which this input is associated with

3.4 Config Structure & Design

In the following, we explore the design of the configs that are used to specify which inputs a given party may access sinks with.

Any given config is a file in JavaScript Object Notation (JSON) [10] format. For a predetermined hostname, it contains a mapping of which party may call which sink with which inputs. An example of a config can be seen in figure 3.2. Apart from the aforementioned mapping, the config also contains an `ignoreList`, which is a list of origins that should be ignored when determining the calling party of a sink access. The other keys on the first level of the config are the origins of the third-party code running on the site. The choice of it being origins rather than hostnames or etld+1 is to keep the individual sink access privileges of each party as restrictive as possible. In this particular example, the party we define privileges for is `https://ads.com`. On the second level, each of the three `Trusted Types` is then mapped to the actual allowlists for this type. If a `Trusted Type` is not present, then the party is not allowed to call any sinks of this type. We now explain the configuration options for each `Trusted Type` using the example figure if appropriate.

3.4.1 TrustedScript

`TrustedScript` allows for three different directives. `Hashes` is an allowlist of SHA256 hash values of JavaScript code samples, while `regexes` provides a list of

```
{
  "https://ads.com": {
    "TrustedHTML": {
      "hashes": ["9ff943a7092302075e74c8ab7798903f74b6b911bc3acabcf11c666cb6ad7166"],
      "scripts": {
        "regexes": ["^window\\.[a-zA-Z0-9]{10,11}\\=\\=0$"],
        "hashes": ["7ac3c3feea64a7f3bf3ff1fa4aff22e5c616fb3e9c41d4896345001f8c541f4e"],
        "prefixes": ["https://example.com/script.js"]
      },
      "strict": false,
      "allow-any": false
    },
    "TrustedScriptURL": {
      "hashes": ["796485be79ab17af515a563cfccfc821cb92471060fb9d20725e3114bfc089da"],
      "origins": ["https://www.gmx.net"],
      "hosts": ["localhost"],
      "prefixes": ["https://example.de/scripts/script.js"],
      "eTLDs": ["example.com"],
      "dataHashes":["7ac3c3feea64a7f3bf3ff1fa4aff22e5c616fb3e9c41d4896345001f8c541f4e"],
      "allow-any": false
    },
    "TrustedScript": {
      "hashes": ["7ac3c3feea64a7f3bf3ff1fa4aff22e5c616fb3e9c41d4896345001f8c541f4e"],
      "regexes": ["^window\\.[a-zA-Z0-9]{10,11}\\=\\=0$"],
      "allow-any": false
    }
  },
  "ignoreList": []
}
```


Figure 3.2: An example config

regexes that any given input should be matched against to determine whether it should be allowed. If the `allow-any` flag is set, then the aforementioned allowlists are to be ignored and the party should be allowed to access the sinks corresponding to this `Trusted Type` with any input. This flag has the same meaning for the other two `Trusted Types`.

3.4.2 TrustedScriptURL

`TrustedScriptURL` contains many different directives for dealing with URLs. `Hashes` provides an allowlist of SHA256 hash values of complete URLs, making it the strictest directive to fulfill. `Origins`, `hosts` and `eTLDs` provides a list of allowed origins, hosts and eTLDs, respectively, meaning that if a URL's origin, hostname or eTLD is in one of these lists, then it should be allowed. `Prefixes`

gives a list of allowed prefixes that a given URL may start with, i.e. if the URL starts with one of these strings, it may contain any arbitrary content afterward. `DataHashes` exists specifically for `data:` URLs, which can also be used in a script source' context to provide the script's content. The directive states the SHA256 hashes of the allowed `data:` URLs content, ignoring any metadata of the URL. For instance, in the example, the SHA256 hash of `alert(42);` is given, so the URL `data:text/javascript, alert(42);` as well as the URL `data:, alert(42);` would be considered allowed, even though they have different metadata values.

3.4.3 TrustedHTML

The last `Trusted Type` provides again a `hashes` directive, which gives a list of SHA256 values that this party may call sinks with. However, since only specific attributes and elements allow execution of JavaScript, it is normally not necessary to allowlist entire HTML values. The `scripts` directive allows specifying what JavaScript may be contained in any HTML sink input via a combination of the three directives `regexes`, `hashes` and `prefixes`, which work exactly the same as their `TrustedScript` and `TrustedScriptURL` counterparts. The `prefixes` directive in particular deals with `script` tags that have a `src` attribute set, meaning external scripts or scripts loaded from a local resource. Lastly, the `strict` flag determines whether an input should be rejected entirely if a JavaScript executing tag or attribute is found that does not match the allowlist in `scripts`, or if such violating elements or attributes should simply be removed by the sanitizer and the rest of the input still be allowed.

3.5 Config Generation

Next, we explain the process of generating configs that are structured as described in section 3.4, which is also shown in a flowchart-like representation in figure 3.3. Generation is done via execution of a Python script, which fetches the data that was collected as described in section 3.3 from one or more given databases. In particular, a single config is generated per origin present in the data. For each such origin, it first fetches the origins of all parties that were observed performing sink accesses on that origin. For each of these parties it also retrieves a list of which of the three `Trusted Types` the inputs observed to have been written by this party

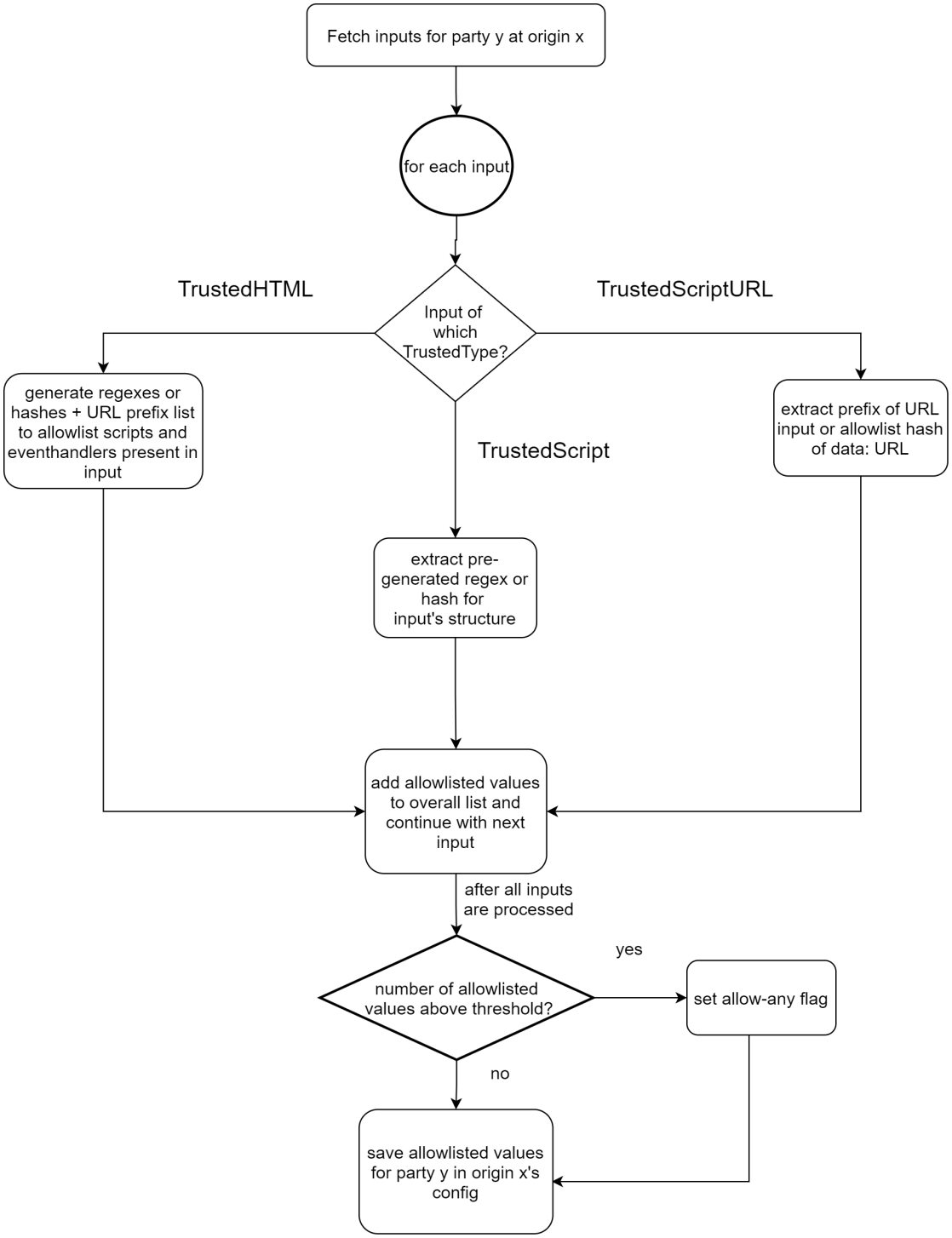

Figure 3.3: Flowchart displaying the process of generating configurations

belonged to. In particular, this means that if a party was not observed to access any sink belonging to `TrustedScriptURL`, then no entry for `TrustedScriptURL` is generated for this party in the config. Then, for each site origin and party origin combination, the generator fetches all the inputs observed by that party and proceeds differently depending on which kind of sink this input was observed to belong to. Should the overall number of values that need to be allowlisted for a certain `Trusted Type` of a party be above a pre-set value which can be specified during the generator's invocation, then the `allow-any` flag is set. Assuming the threshold for this decision is sound and high enough, a number of allowlisted values above this threshold implies that the inputs this party uses in sink calls are so diverse that the allowlists are likely missing possible benign inputs anyway and thus allowing simply any input is necessary to preserve functionality.

In the following sections, we discuss the individual handling for inputs belonging to each of the three `Trusted Types` as well as giving reasoning for the chosen directives in each case.

3.5.1 TrustedScript

The generation for this `Trusted Type` primarily makes use of the `regexes` directive. Simply using only hashes or structural hashes of benign inputs is not sufficient. First, just hashing all seen benign JavaScript code inputs and allowlisting these hashes may work for static inputs, but might not be sufficient for dynamically generated inputs. To illustrate the problem, observe the following simple listing of code that is assumed to run on some site:

```
function doExpensiveAction(value) {
  // do some expensive computation with the given value
}

for(var i = 0; i < 10000; i++) {
  eval('doExpensiveAction(' + i + ');');
}
```

Listing 11: An example of dynamic sink invocation

In this code, we perform a call to the JavaScript executing sink `eval` with a dynamic string that depends on the counter variable `i`. The function call to `doExpensiveAction` is assumed to perform an expensive computation. Depending on the time we actually spend observing the site's behavior, we would for instance

only see values of `i` of up to 1000 being used. If we were to now use hashes to allowlist these values that this code snippet calls `eval` with, we would *not* allowlist `doExpensiveAction(1001);` since we never observed this value. Thus, this input would be classified as not allowed later during enforcement, even though it clearly *should* be allowed.


```
[
    {
        "type": "Identifier",
        "value": "window"
    },
    {
        "type": "Punctuator",
        "value": "."
    },
    {
        "type": "Identifier",
        "value": "outerWidth"
    },
    {
        "type": "Punctuator",
        "value": "=="
    },
    {
        "type": "Numeric",
        "value": "0"
    }
]
```


Figure 3.4: An example of JavaScript tokenization

Using structural hashes provides a security problem, rather than one with functionality. They make use of the pure syntactical structure, i.e. the tokenization of a given JavaScript string. An example of such tokenization for the string `window.outerWidth == 0` can be seen in figure 3.4. When building a structural hash, one might simply chain all the token types together and hash this value. However, while this does preserve functionality well, Fass et al. [29] have shown that given a set of allowed structures, it is often possible to construct malicious JavaScript code that shares the same allowed structure and thus would be allowed.

Instead of using either of these two flawed approaches, we opt to use an approach heavily inspired by the work of Stock et al. [41]. The main idea is that we make use of the observed benign input's syntactical structure as well as the individual values of each token to generate regexes that match only benign variations of the observed inputs. In order to do so, we first perform clustering of all collected inputs. In particular, a given input's respective cluster is determined via the hostname of the site on which it was observed, the hostname of the party which wrote this input,

and the SHA256 hash of the input's pure syntactical structure, i.e. a structural hash of the input's tokenization, which is obtained using the esprima library [8]. We then traverse each of the resulting clusters and observe the number of their entries. Should only one input be assigned to a cluster, then this implies that the input is unique and we simply allowlist it explicitly using the `hashes` directive. Otherwise, we collect all inputs in the cluster and normalize them by removing any kinds of comments and whitespace present to keep the resulting regex as simple as possible. Said regex is then generated for the cluster by iterating through the inputs' respective token values. Should all token values be the same, the value is explicitly allowed in the regex. Otherwise, the length range between the different values is determined and a matching character range is bruteforced from a set of predefined choices. The regex generated by this description is additionally configured with a `caret` character (^) at the beginning and a `dollar` character ($) at the end to enforce that this regex matches the *entirety* of a given input. The generated regexes and hashes are then simply put at their respective place during config generation. Figure 3.5 gives a small example of the regex generation process's idea. The two displayed inputs display exactly the same token structure and token values, except for the third token, in which their respective values are `outerHeight` and `outerWidth`. Thus, to allow them both, we bruteforce a subregex for them, in this case, a alphanumerical value range suffices to allow both values.

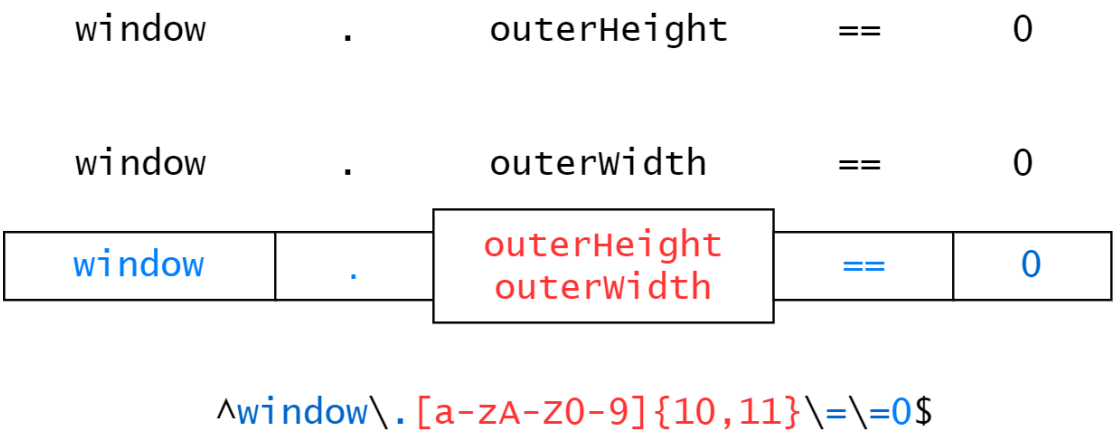

Figure 3.5: A simple example of regex generation

3.5.2 TrustedScriptURL

For this `Trusted Type`, we opt to primarily use the `prefixes` directive. In particular, we build prefixes consisting of the observed URL's protocol, hostname, port and path, but *without* any GET parameters, i.e. the query part of the URL. This decision comes from the fact that the query parameters for a script request can potentially be dynamic, which just like with dynamic JavaScript strings makes it hard to determine the set of benign query parameters from the observed benign ones alone. Furthermore, it is (normally) the origin and path of the URL rather than the query parameters which determine the actual script requested. Thus, it is in general sufficient to restrict origin and path combinations that can be loaded inside a script tag's `src` attribute while ignoring the query parameters to obtain a significant reduction of attack surface while not restricting functionality. We parse a given URL into its respective parts with the urlparse library. [21]

For URLs pointing to *local* resources, we simply allowlist the entire path with the `prefixes` directive, for the same reasons as for the URLs pointing to external sources. There are also URL inputs which have a `blob:` prefix before containing a normal `http:` protocol URL. Thus, this prefix is simply normalized away, and then the rest of the URL is treated like any other external URL. During our evaluation, we found that all `blob:` URLs contained in our data set indeed follow the described format, as described in section 4.1.3.

Lastly, we must also account for the usage of `data:` URLs. To also allowlist these, we simply opt to allowlist the SHA256 hash of the `data:` URL's content with the `dataHashes` directive. As described in section 4.1.3, we did not find any such `data:` URLs in the wild and thus only implemented this simple handling for them.

3.5.3 TrustedHTML

For a given benign HTML input, we use the `scripts` directive to allowlist all contained JavaScript. First, the input is parsed using the BeautifulSoup library [2], then all inline scripts, event handlers and external scripts are collected and their contents allowlisted via the `regexes` and `hashes` or the `prefixes` directive, in the exact same ways as for the other two `Trusted Types`. The `strict` flag is currently never set since it does not provide a significantly reduced attack surface while potentially hindering functionality. Lastly, it should be noted that we do *not*

consider `javascript:` URLs, even though they can be used to execute JavaScript. This is because the code contained `javascript:` URLs is always handled by the `createScript` function of the default policy when it is attempted to be executed anyway [18], so explicitly allowlisting them here would be redundant as they have already been considered in the `TrustedScript` config generation.

3.6 Config Enforcement

Next, we explain how the generated configs are enforced in a site. This enforcement is done via the so-called `SanitizerLibrary`, a client-side JavaScript library that we constructed exactly for this purpose. It is a simple JavaScript file that can be included in a site like any other script. Within it, the actual `SanitizerLibrary` class is defined, which, provided a config object in its constructor, can then be used to sanitize any given sink input based on its `Trusted Type` and the calling party. In particular, this is done by registering a default policy which simply calls the `SanitizerLibrary` and returns its output as follows:

```
trustedTypes.createPolicy('default', {
  createScriptURL: (s, type) => {
    return sanitizerLibrary.sanitizeInput(s, type);
  },
  createScript: (s, type) => {
    return sanitizerLibrary.sanitizeInput(s, type);
  },
  createHTML: (s, type) => {
    return sanitizerLibrary.sanitizeInput(s, type);
  }
});
```

Listing 12: Default policy used to make use of the `SanitizerLibrary`

The setup for the `SanitizerLibrary` and the registration of the shown default policy is done at the beginning of `head` tag of the document to make sure it is active for all sink accesses on the site.

Once called, the `SanitizerLibrary` first checks whether the given input is empty, as this is not problematic, and if so, simply allows the input. For any other input, the `SanitizerLibrary` will automatically determine the party that made the call to the sink. This is done by manually raising an exception and traversing that exception's stacktrace until a non-empty party not contained in the config's

`ignoreList` is found. It then further extracts the relevant portion of the config for this particular party by using the party's origin. Next, it determines whether or not, or in the case of a non-strict `TrustedHTML`, what part of the input is to be allowed, returning either a (potentially modified) input if it was a least partially allowed, or `null` if the input was *not* found to be allowed. Since a default policy returning `null` triggers a CSP violation [20], it thus prevents the sink from being accessed with the non-allowed input. In order to be considered allowed, a given input has to fulfill at least one of the directives in the corresponding config entry for the current party and `Trusted Type`. In the following, we provide details about how the `SanitizerLibrary` processes inputs for each `Trusted Type`.

3.6.1 TrustedScript

Given an input that originated from the `createScript` function of the default policy, the `TrustedScript` entry from the config of that party is used. The `SanitizerLibrary` simply checks whether the input matches any of the given hashes or regexes, and if so, declares the input allowed. Additionally, any input that can be parsed as `JSON` is always allowed regardless, since this implies that it is merely data, which is not problematic. This explicit handling is further justified by the fact that we found such inputs in the wild, as section 4.1.2 shows.

3.6.2 TrustedScriptURL

For inputs originating from the `createScriptURL` function of the default policy, the `SanitizerLibrary` simply checks whether it matches against any of the directives present in the corresponding `TrustedScriptURL` entry in a completely symmetrical way to how the entry was generated. That is, it checks whether its SHA256 hash is present in the `hashes` directive, or whether its domain/origin/etld is present in any of the corresponding directives, or if it starts with any of the allowlisted `prefixes`. `blob:` URLs are stripped of this prefix for the checking process, and the content of `data:` URLs is checked against the `dataHashes` directive, decoding the `base64` encoded content if necessary.

3.6.3 TrustedHTML

The `SanitizerLibrary` proceeds differently for a given HTML input depending on whether or not the given `scripts` directive contains any entries. If during the generation of this config entry no JavaScript was observed in any of the HTML sink inputs, then all of the allowlists in the `scripts` directive are empty. Thus, if no entries are found, the library assumes that no JavaScript should be allowed in the HTML and removes all of the potentially contained JavaScript using the DOMPurify library [7]. If that is not the case, then the `SanitizerLibrary` uses the DOMParser API [6] to parse the input into a document. Then, all scripts and event handlers are checked against the respective sub-directives to determine whether or not they are allowed. If a non-allowed entity is found, it is either simply removed if `strict` is not set in the config entry, otherwise the input is rejected entirely by returning `null`. Lastly, any `iframe` that uses a `data:` URL as its source has this URL's content sanitized using DOMPurify. This is done since we did find any occurrence of `data:` URLs being used in iframes that are written via HTML sinks, as described in section 4.1.4.

It should also be noted that using the DOMParser in itself is also a dangerous sink access. Thus, it would usually again trigger a default policy access, causing an endless loop. To circumvent this problem, we used a no-op policy called `dummy` that simply returns the input to pseudo-sanitize the input that is used in the DOMParser call.

Naturally, an input is also allowed in its entirety if the current `hashes` directive explicitly allowlists its SHA256 hash.

Chapter 4

Results

In this section, we present our findings concerning the data gathered from observed sink API calls, as well as the configs generated from this data. We further describe the measures used to evaluate the functionality penalty provided by enforcing the generated configs, as well as the performance overhead the enforcement exhibits on the web page. Lastly, we analyze how much attack surface reduction the enforcement of configs achieves in case of a compromise.

4.1 Data & Config Set Analysis

Category	Number of inputs	Percentage of total inputs
TrustedHTML	2900	54.92%
TrustedScript	1292	24.47%
TrustedScriptURL	1102	20.87%
Total JavaScript	1509	
Total	5280	

Table 4.1: Number of distinct inputs collected per Trusted Type

We performed the data collection as described in section 3.3 for the top 100 domains from the Tranco list [17], using a site loading timeout of 30 seconds and a module timeout of 10 seconds. Doing so on 13.02.2021 produced a data set containing 170 distinct URLs with 126 distinct origins. We chose this rather small dataset since it allowed for quick termination of all evaluation tools, while usage of bigger

datasets causes considerable evaluation time overhead that would have made a sensible evaluation infeasible in the given time constraints.

Table 4.1 shows the number of distinct inputs that were observed during collection for each of the three `Trusted Types`. In particular, we observed over 5000 distinct inputs, out of which more than half belonged to the `TrustedHTML` type. Inputs corresponding to the other two `Trusted Types` were observed about equally as often, though `TrustedScript` did noticeably occur a bit more often. Additionally, we extracted JavaScript samples from inputs corresponding to `TrustedHTML`, in particular, all inline scripts and event handlers, and calculated the set union of these samples with the JavaScript written to `TrustedScript` sinks. From this, we obtained 1509 unique JavaScript samples across all sink accesses.

We further used the previously described methodology from Section 3.4.3 to generate configs from the collected data. From the process we obtained 126 distinct config files, matching the number of observed origins. Altogether, the configs contained 506 non-empty allowlists across all directives, summing to a total of 1719 distinct allowlisted values. Out of these allowlists, 243 were found to only contain a single element, which amounts to almost half of the constructed allowlists. Additionally, the average length of all allowlists was 3, while the average length of all allowlists that contained more than one element was 6, implying that most of the allowlists are small in size. We further found that the maximum amount of elements in a generated allowlist was 105. In that particular case, this was due to many different hashes being generated for observed `TrustedScript` inputs, which were all syntactically distinct such that no regex could be generated for them.

Using the gained knowledge, we re-generated the configs with the `allow-any` directive enabled for a threshold of 10 values. We found that out of 380 sub-configs, this produced 50 with `allow-any` set, making a total of 13.16%. This further supports our finding that most allowlists are small in size.

Next, we present more specific findings in regards to the inputs that the sink APIs were called with during our collection process.

4.1.1 Input Clustering Analysis

As described in section 3.5.1, we cluster the observed JavaScript inputs based on the party which wrote them, the origin on which they were written and their syntactic

structure. After generation of configs from the observed inputs, we found that 239 distinct clusters had been generated for the overall sum of 1509 distinct JavaScript samples. Out of these clusters, 120 were found to contain exactly one element, and 119 were found to contain more than one element, which shows that there is an almost equal distribution of whether or not JavaScript inputs are structurally unique for a given site and party combination.

4.1.2 JSON inputs to TrustedScript sinks

```
res = JSON.parse('{"data": 42}');
// value assigned to res : {"data": 42}
res = eval('({"data": 42})');
// value assigned to res : {"data": 42}
```

Listing 13: Using eval to parse JSON from a string

In order to parse JSON data from a string, modern browsers provide the `JSON.stringify` API. However, older browsers do not support this API [11], and thus legacy code or developers that wish to support legacy browsers may instead use `eval` to obtain the data. Listing 13 provides an example of this usage. Since such a call to `eval` is merely data, it is not dangerous.

We analyzed our collected data to determine whether or not this is actually done in the wild. Indeed, out of the 1292 `TrustedScript` inputs, we found 18 inputs that are parsable to JSON objects. Still, this a rather negligible amount of below 2%.

4.1.3 Distribution of TrustedScriptURL Protocols

For all inputs corresponding to `TrustedScriptURL`, we determined the protocol used, or the absence of it, and grouped them. Figure 4.1 shows the results of this process. Above three-quarters of the URLs used a variation of the HTTP protocol, providing an absolute number of 829 URLs. Out of these, 820 used the HTTPS protocol, while only 9 made of plain HTTP. Additionally, almost one-fifth of URLs provided no protocol at all, instead using protocol-relative URLs [1]. URLs pointing to local scripts hosted by the site itself only made up 5% of the total. Furthermore, only 8 `blob:` URLs were found in total, though all of them were

[1] Protocol-relative URLs automatically determine whether to use HTTP or HTTPS based on the protocol used by the site in which they are constructed.

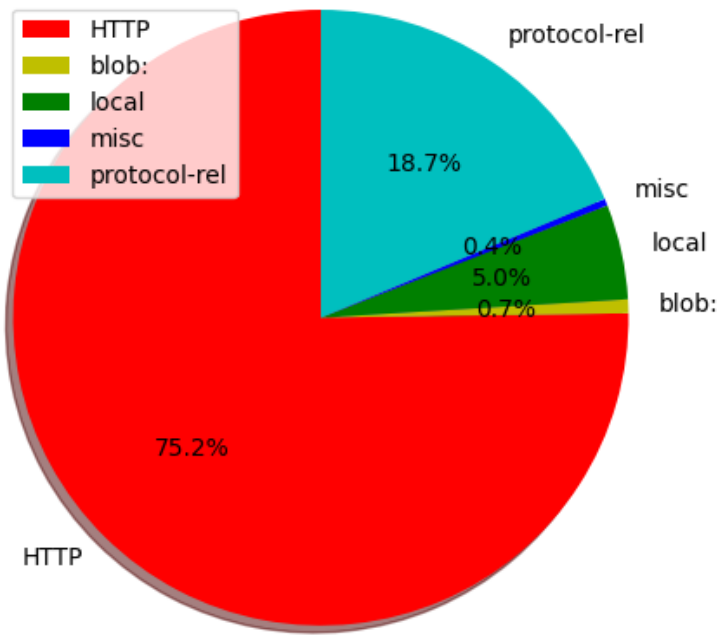

Figure 4.1: Distribution of URL protocols

prefixes to another HTTP URL. The category `misc` includes all inputs which could not be categorized by any of the other protocols. In particular, we found 4 such URLs, out of which 3 contained browser-specific protocols such as `edge://` and one was the JavaScript constant `undefined`. Lastly, `data:` URLs were not found, making it seem as though this mechanism is generally not used in the wild for script content inlining. In conclusion, almost all scripts queried in the wild seem to originate from external resources.

4.1.4 Presence of JavaScript in TrustedHTML inputs

For the inputs corresponding to `TrustedHTML`, we analyzed the occurrence of JavaScript within them as well as the configs generated from them. To this end, we searched all inputs for the occurrence of script tags as well as inline event handlers. We found that out of the 2900 collected `TrustedHTML` inputs, only 205 contained some kind of JavaScript, making up less than 10%. Additionally, during analysis of the generated configs, we found 235 `TrustedHTML` subconfigs. Furthermore, 173 of these subconfigs contained only empty allowlists within the `scripts` directive,

meaning that the parties' inputs to HTML sinks did not contain any JavaScript. In total, this means that 73.62% of HTML subconfigs originate from only non-code markup. We also did not find any instance of an iframe that makes use of a `data:` URL in its `src` attribute, which can also potentially be used to execute code. Lastly, 8 of the 126 configs were found to contain *only* empty `TrustedHTML` entries, showing that on these origins all parties only wrote non-code HTML markup.

4.1.5 Summary

Throughout this section, we made a few key observations about the sink usage of parties in the wild. We found that HTML sinks are used in over 90% of cases to only write non-code markup. In terms of HTML subconfigs, almost three-quarters of them were built from non-code markup. Thus, for a significant majority of HTML sink accesses, no JavaScript should ever be included in a benign use case. We further found that almost all script content that is loaded via a `TrustedScriptURL` sink is loaded from an external resource rather than a local one. Furthermore, we saw by our clustering that the syntactical structure of `TrustedScript` inputs across all site/party combinations was about as often unique as it was not unique, implying that inputs could be grouped together by structure in about half of the cases. Lastly, we observed that the inputs for a given site/party combination were often of rather low diversity, making only a low amount of values necessary to be allowlisted in the config to account for all observed possibilities.

4.2 Functionality Evaluation

As one of our primary goals is to preserve functionality while the generated configs are enforced, we performed a study as to how well functionality is preserved. To this end, we repeated the data collection as described in section 4.1 10 times in the time span from 13.02.2021 to 22.02.2021, collecting once each day at approximately 7 pm. We then generated configs using the snapshot from the first day and an additional set of configs using the first two snapshots. To determine how much the enforcement of the configs would hinder functionality, we simulated the enforcement by instantiating the `SanitizerLibrary` with the config for each corresponding origin and then invoking the library with each input from a data set, collecting the result of the invocation and checking whether it is the same

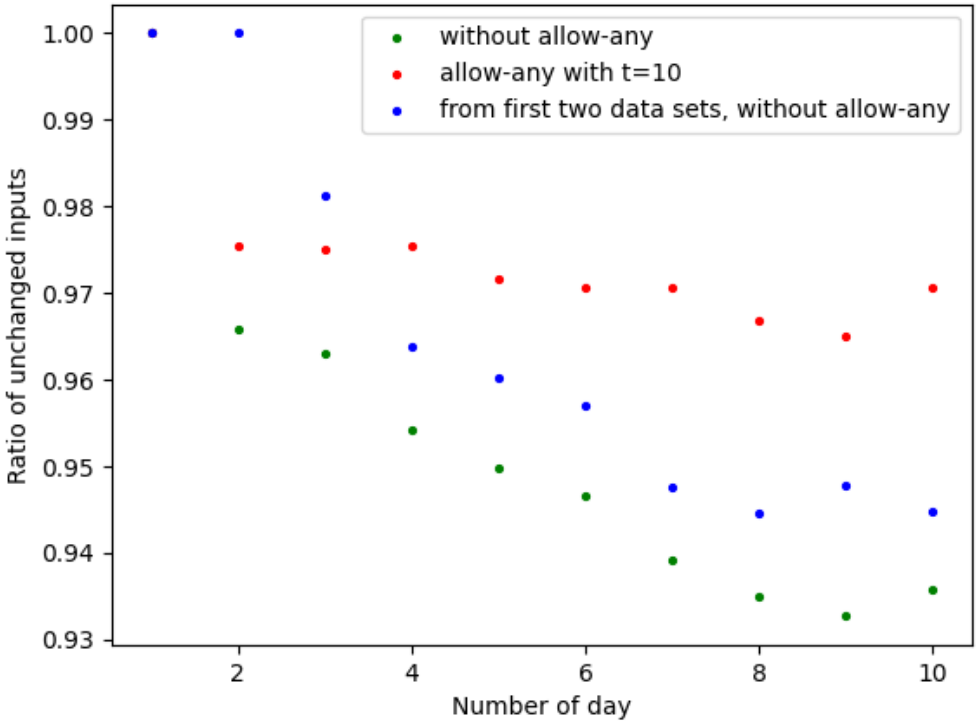

Figure 4.2: Functionality preservation during config enforcement

as before the sanitization. If the input was not changed, then this implies that the `SanitizerLibrary` correctly identified it as being benign. For each data set obtained during the 10 days span, we executed the described experiment and counted the number of unchanged inputs by sanitization against the overall number of inputs. Should we encounter an origin during usage of a data set that is not the initial one for which no config was generated, it is noted and skipped. We ran the entire experiment once for vanilla configs [2] generated from one snapshot, once for configs that used the `allow-any` with a threshold of 10 from one snapshot, and lastly once for vanilla configs generated from two snapshots. Additionally, should we find inputs which were altered or rejected during sanitization, we investigated the reasons.

4.2.1 Experiment Results

Next, we present and discuss the findings of our 10-day evaluation. Figure 4.2 shows the results in a graph. The x-axis marks the day of evaluation, that is, which dataset was used. For instance, day 2 uses the data set from the second day of the collection which is 14.02.2021. The y-axis shows the ratio of inputs that are correctly left unsanitized during enforcement of the configs by the `SanitizerLibrary`. The green points mark the values obtained with the vanilla configs, while the red ones display those obtained by using the `allow-any` directive with a threshold of 10. The blue points represent the values gathered while using the configs generated from the first two snapshots.

Using the data from the first day, that is, the data from which the configs were initially generated, we find a functionality preservation of 100%, meaning that no input was changed during enforcement in both cases. This in itself serves as an indicator that our configs are sound and match the inputs they were generated from. Considering the one-snapshot vanilla configs, we observe a decline in functionality preservation after the first day. In particular, the decline is almost monotonic, implying that configs must be re-generated in some intervals of time, though there are some days at which fewer inputs are altered than using the previous day's data, for instance from day 9 to 10. The reason for this is likely that the inputs were simply slightly more similar to the inputs from the initial day. Overall, though there is a decline, we can further see that the preserved functionality ratio never goes below 93% in the case of the vanilla configs, indicating that a great majority of functionality is preserved during this time span. For the `allow-any` configs, the preserved functionality is even higher, not going below 97% percent, which originates from the fact that the purpose of the `allow-any` flag is to sustain functionality in a better way. Considering the configs generated from two snapshots, we find that functionality is preserved perfectly on the first two days, which again shows that the generated configs are sound, since these two data sets are exactly the ones used for the generation in this case. After that, however, we also observe a decline in functionality-preservation similar to the other two config sets. Notably, the ratio of unchanged values goes below that of using the `allow-any` configs after day 3, implying that the functionality benefit from using `allow-any` may generally outperform that of using more than one snapshot in the generation. Further,

[2]The term *vanilla configs* refers to configs which do not make any use of the `allow-any` directive.

the ratio always stays above those of the single-snapshot vanilla configs since the two-snapshot configs contain a superset of allowlisted values of the one-snapshot vanilla configs. Interestingly, the benefit gained after day 3 between one-snapshot and two-snapshot vanilla configs is around 1% and thus rather low.

Overall, we find that enforcement of our generated configs still preserves a great majority of the functionality. It should, however, be noted that while a large ratio of functionality is indeed preserved, failure of a single script may cause noticeable dysfunction on a given site.

4.2.2 Reasons for Input Alteration

In this section, we explore the reasons as to which inputs were rejected or altered during the enforcement of the configs in our experiment. To this end, we performed a manual analysis using the evaluation results of the second day where the one-snapshot vanilla configs were used. We chose to only view this particular case due to time constraints, as also performing manual analysis on the results of other days would likely not give many new insights anyway. Additionally, we performed an analysis of the distribution of altered inputs over the problem types across the data of all 10 days.

Overall, in the evaluation using our second dataset, we found that 5642 out of 5842 inputs were left unchanged, leaving 200 altered inputs.

4.2.2.1 TrustedScript

We found 50 cases in which an input belonging to `TrustedScript` was altered. In particular, we found that the changed inputs would share their syntactical structure with previously observed inputs. However, these inputs were previously unique in their structure for their particular site/party combination and thus only a hash of them was persisted in their config, not accounting for any kind of dynamic variation in this input's structure since none was observed during collection.

One example of such an occurrence is a function name that is suffixed with a dynamic, possibly random id. Listing 14 shows such an example, found in the collected data.

```
// allowlisted input
fun_739534080154291152=function(res){cb(res)}
// new input
fun_685197305778528575=function(res){cb(res)}
```

Listing 14: Real-world JS example that is rejected during sanitization

4.2.2.2 TrustedScriptURL

Concerning `TrustedScriptURL`, a total of 69 values was found to be rejected during enforcement. During our analysis, we found that the rejection occurred due to new URLs values that did not match any of the entries allowlisted in the `prefixes` directive. In particular, we found two different cases. First, the unrecognized URL would have an entirely new host that was not observed before, implying that the external script was loaded from a dynamic resource that might change frequently or that another part of the party's code was triggered. The second case features an URL with a previously seen URL structure, but a different value in the path, thus pointing to another resource which was not previously allowlisted by the `prefixes` directive as it only allows previously seen paths.

```
// allowlisted input
"https://securepubads.g.doubleclick.net/gpt/pubads_impl_2021020901.js"
// new input from same host but different path
"https://securepubads.g.doubleclick.net/gpt/pubads_impl_2021021101.js?31060147"
// new input from different host
"https://wa.gtimg.com/web/default_ad.js?callback=crystal.getDefaultAd"
```

Listing 15: Real-world example of rejected URLs

Listing 15 shows an example of the described problem. Similar to the `TrustedScript` case, we also observe a different, possibly random id value as the difference between the two values that have the same structure.

4.2.2.3 TrustedHTML

We found the rejection instances in `TrustedHTML` often to be similar to those observed in `TrustedScript`. In particular, we observed JavaScript code samples within script tags and event handlers that would share their structure with previously allowlisted values, but those values were unique in the dataset and thus no regex was constructed, but rather only a hash value saved. Additionally, inputs were found that would match the general structure of a previously generated regex

from the `regexes` directive, but the value of some tokens would not match the previously determined length-range of allowed values. We also discovered some instances that featured a completely new value in a token where no different values were observed during generation, which caused that token's value to be hardcoded in the config.

```
// allowlisted value 1
window.registerInteractive && window.registerInteractive("100000007493277");
// new value
window.registerInteractive && window.registerInteractive("100000007342984");

// (simplified) allowlisted value 2
/ensemble\.setupPagelet\([a-zA-Z0-9\"_;-]{11,22}\)/
// (simplified) new value
ensemble.processChunk("user-chat")
```

Listing 16: Real-world example of removed scripting content

Listing 16 provides examples of the mentioned instances. In the first example, the problem is the string parameter of the function call that has changed, while the second example displays a new token value for a token previously observed to have a unique value. Note that the second example has been slightly shorted and simplified in contrast to the actual example in the real data for brevity and simplicity. Overall, 26 of the 200 rejected inputs from our experiment belonged to `TrustedHTML`.

4.2.2.4 Unseen Parties

In addition to inputs being rejected or altered due to their corresponding party's allowlist not recognizing them, we found the remaining 55 inputs to be rejected because the parties that invoked a sink with them had not previously been observed at all on the corresponding site. Hence, no entry for any of these parties existed in the corresponding site's config, and thus the input was by default marked as disallowed by the `SanitizerLibrary`.

4.2.3 Problem Type Distribution

We additionally re-performed the measurement of how many inputs were altered or rejected per problem type for all data sets collected over the 10 day span. Figure

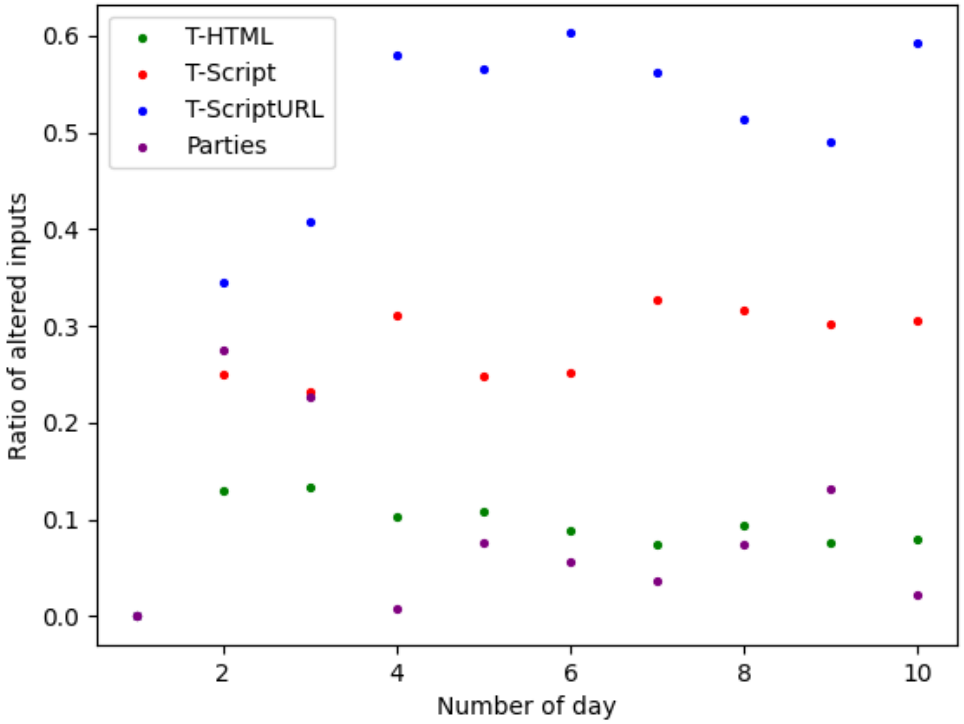

Figure 4.3: Ratio of altered inputs for each problem type

4.3 shows the results of this experiment. Since all inputs from the first data set remained unchanged, no interesting results occur from there. For the rest of the sets, we see that `TrustedScriptURL` provides the most rejected inputs, though there is some fluctuation in the exact ratio. This implies that the usage of varying hosts or varying paths on observed hosts to load script content from provides the most problems in preserving functionality as of the current usage of the `prefixes` directive. `TrustedScript` provides a much smaller, though less varying amount of the rejections, while `TrustedHTML` provides the smallest amount, which makes sense considering our finding from section 4.1.4 that only a very small amount of `TrustedHTML` inputs actually contain any JavaScript at all. Lastly, the ratio of altered inputs caused by new parties attempting to use sinks but not having any entry in the corresponding config fluctuates extremely, going from well over 20% on days 2 and 3 to below 10% on the days afterward. This shows that the choice of which third-parties access sinks on a given site is very non-deterministic in comparison to the parties displayed by a snapshot made at one particular time.

4.2.4 Number of breaking Origins and Parties

Over the course of the 10 day experiment, we found a total of 2684 inputs which were altered during sanitization using the one-snapshot vanilla configs generated from the dataset of day 1. We additionally analyzed which sites the inputs were found to be written on, and by which parties. We found 98 distinct site/party origin combinations. Further, we determined that the altered inputs came from 55 distinct sites, and were written by 76 distinct parties. These findings imply that the reasons for input alteration during sanitization are neither concentrated on one particular site, nor on one particular party, but rather across a bigger set of both. In particular, we observe that the number of distinct sites is almost half the number of distinct site origins found in the original data set at day 1, showing that the altered inputs are indeed spread over a larger portion of the sites. However, it would seem that there is more diversity across the parties that cause breakage than across the sites on which the altered inputs were observed.

4.2.5 Summary

In this section, we performed experiments to determine functionality preservation of enforcing configs over a time span of 10 days. We found that functionality is perfectly preserved against the original inputs, and while it does decrease over time, the decrease rate is quite low, staying well above 90% in the tested timespan. We further analyzed the reasons as to which inputs are rejected by the config enforcement over time, and established four categories: One for each `Trusted Type`, as well as one for new parties that were not observed in the initial data set. We observed that most of the rejections were caused by the usage of new hosts or new paths on existing hosts for fetching external script content, while the occurrence of new parties was found to be very inconsistent, implying a great amount of non-determinism involved in choosing which third-party scripts are to be included for a given site's run. Additionally, the altered inputs did not concentrate on a very small amount of sites and parties, but rather across a larger set.

4.3 Runtime Evaluation

In order to measure whether our infrastructure was also feasible in practice, we performed a runtime evaluation. This evaluation was only performed on the `SanitizerLibrary`, that is, the component actually enforcing configs in the browser. The rest of the components were not measured as they need only be run once, while the `SanitizerLibrary` may be called many times depending on how many sink accesses happen on the site. Further, the overhead of the `SanitizerLibrary` might potentially noticeably degrade the performance of the including site, leading to user dissatisfaction, while the data collection and config generation is done separately altogether. The configs used during the evaluation are once again obtained by repeating the data collection experiment as described in section 4.1, in particular, also for the top 100 sites of the Tranco list.

We then constructed another crawler module very similar in its functionality to the one described in section 3.3. In essence, the module performs two different kinds of measurements. First, we measure the overhead on the page load time that the setup as well as the enforcement of the configs exhibits on the given page. To this end, we load the config generated for the site we are visiting, then visit it once to fill the browser's cache and reduce non-determinism in the loading time of resources. Next, we visit the site an additional 10 times while once including a simple no-op default policy that only returns the given input in the site, measuring the load time in each visit and saving the average value. We perform the same step another 10 times, but while including the `SanitizerLibrary`, initializing it with the config generated for that site. Listing 17 provides a simplified version of the code for better understanding.

As a second measurement, we perform *benchmarking* to determine the execution time of each invocation of the `SanitizerLibrary`. Before the call and after the call the current absolute time is measured, and the difference of these values is then the runtime of the invocation. Listing 18 displays this in a simplified manner, again for clarity.

4.3.1 Loading Time Results

For the time required to load the pages, we determined an average value of 1895.7 milliseconds without using the `SanitizerLibrary`, and an average of 1966.6

```
    // visit site once to fill cache
        await that.page.goto(url);

        include_lib = false;
        let start, end, average;
        let sum = 0;
        // run 10 times without sanitizerlibrary
        for (let i = 0; i < 10; i++) {
                start = new Date();
                await that.page.goto(job.url, {
                    waitUntil: 'load',
                    timeout: args.timings.load,
                });
                end = new Date() - start;
                sum += end;
        average = (sum / 10).toFixed(3);
        // values are persisted
        sum = 0;
    include_lib = true;
        // run 10 times with sanitizerlibrary
        for (let i = 0; i < 10; i++) {
      start = new Date();
      await that.page.goto(job.url, {
        waitUntil: 'load',
        timeout: args.timings.load,
      });
      end = new Date() - start;
      sum += end;
        }
        average = (sum / 10).toFixed(3);
    // values are persisted
```

Listing 17: Simplified code for the load time overhead measurement

```
 // benchmarking using config
const policy = trustedTypes.createPolicy('default', {
  createScriptURL: (input, type) => {
    let start = performance.now();
    let res = sanitizerLibrary.sanitizeInput(input, type);
    let time = performance.now() - start;
    // time value is persisted
    return res;
  },
  createScript: (input, type) => {
    // equivalent to createScriptURL
  },
  createHTML: (input, type) => {
    // equivalent to createScriptURL
  }
});
```

Listing 18: Simplified code for the microbenchmarking measurement

milliseconds with including the library and its setup. Consequently, we observe an average overhead of *3.7%*, which is very similar to the overhead observed in the evaluation of `ScriptProtect` [33]. However, it may be noted that all dependencies of the `SanitizerLibrary` were fetched from external resources and the source code was not minimized, implying that the overhead could potentially be further minimized. Overall, the load time overhead exhibited by the `SanitizerLibrary` and its setup is thus rather low and in an acceptable range that should not cause any usability issues for the user.

4.3.2 Microbenchmarking Results

As we have estimated maximum values and average values for each origin, we consider them both in the following. Table 4.2 provides a few of the most interesting values found during our evaluation. In particular, we show a few of the highest values observed. As the table shows, the general average of maxima is about 20 milliseconds, but a few of the values are way higher. To determine why exactly such high benchmarks occur, we re-performed the experiment while logging any input and the origin on which it was observed that took the `SanitizerLibrary` longer than 60 milliseconds to sanitize. We found that the problematic values were all instances of `TrustedHTML` that are between about 50 and 2100 kilobytes in size. Since for each such `TrustedHTML` input, the `SanitizerLibrary` must parse it using the `DOMParser` and then iterate over all of its elements, it makes sense that such high values can potentially occur if a very large input is used to call a `TrustedHTML` sink. It may further be noted that `http://youtu.be` also resolves to `http://youtube.com`, which is the reason why these two show almost the same values.

Overall, we can see that the average values for each origin are rather low, as implicated by the overall average execution time being less than 2 milliseconds. Interestingly, the average observed from `google.com` is very high, but that is merely because only one sink access was observed on the site at all, as indicated by the identical maximum value for this origin. Thus, we can conclude that the `SanitizerLibrary` shows a low execution overhead in general, except for a few cases in which it is given huge `TrustedHTML` values to sanitize.

Origin	Max	Average
http://youtu.be	249.23	1.423
http://youtube.com	244.67	1.602
http://ebay.com	76.02	2.554
http://google.com	25.32	25.32
Total Averages	20.39	1.67

Table 4.2: An excerpt of the observed benchmarking values (in milliseconds)

4.4 Attack Surface Reduction

As described previously, the primary goal of using the `SanitizerLibrary` to enforce the generated configs is to reduce the attack surface exposed by vulnerable code. In particular, recall from section 3.1 that we assume *all* observed sink accesses to potentially be controllable by an attacker. Using the data set from day 1, we found that a total of 281 sink-accessing parties, out of which 194 were third-parties, that is, their origin did not match the origin of the site they were executing code in. We found that sites included an average of three third-parties, and a maximum of 15 distinct third-parties.

We were able to generate vanilla configs for *all* of these third-parties, meaning that we were indeed able to reduce the attack surface exposed by *all* third-parties. Thus, if a compromise occurs among one of these third-parties, an attacker can only inject code that is allowed by the corresponding entry in the site's config, which is restricted to those values similar to the benign values observed, assuming the vanilla configs are enforced. Considering the configs which we generated with `allow-any` threshold of 10, we found that 41 distinct parties were given config entries with the `allow-any` flag set. Hence, 14.6% of parties would be potentially exploitable using these configs, as a controllable sink access while enforcing these configs would not be restricted.

Chapter 5

Discussion

In the following, we present the limitations that our approach currently has, as well as presenting some issues that the current implementation of `Trusted Types` exhibits. Additionally, we discuss the trade-off assumptions between security and functionality made in the design of our approach as well as its implications.

5.1 Limitations

This section presents the limitations of our approach, in particular, potential bypasses for the enforced configs, as well as some minor problems of the data collection and the config generation.

5.1.1 JSONP endpoints

Normally, two different domains cannot share content directly due to the SOP. In particular, if a domain performs an HTTP request to an endpoint of another domain to fetch some data, it cannot read the response. To circumvent this issue, a mechanism called JSON with padding (JSONP) was invented. Such JSONP endpoints provide not only the data in their response, but rather JavaScript code that can be executed which contains the requested data. Since included scripts are always executed in the origin of the including site, the requested data can then be accessed. Listing 19 shows a small example of JSONP usage. The function that is

to be executed with the requested data is given as a query parameter, in this case called `cb` for `callback`.

```
<script>
function doSomethingWithData(data) {
  // perform some interesting computation using the data
}
</script>
<!--
returns data wrapped in function call, e.g.
 doSomethingWithData('{"user": "Daniel"}')
-->
<script src="https://foo.com/jsonp?cb=doSomethingWithData">
</script>
```

Listing 19: A minimal example of JSONP usage

Assuming such a JSONP endpoint does not properly allowlist the `callback` parameter, it is possible to call any *arbitrary* function using this parameter. In our current design that is a potential security flaw, since the `prefixes` directive always ignores the query parameters when allowlisting, and would thus allowlist the entire JSONP endpoint if one is observed on a site. Thus, if an attacker is e.g. able to manipulate a newly created script's `src` attribute, they are able to call *arbitrary* functions on the site. While they cannot simply call `eval` to execute any arbitrary code on the site, since any `eval` call's input is intercepted by the default policy, it is still possible to exploit this, e.g. to leak the current user's session cookie to the attacker (assuming it does not have the `httponly` flag set). Listing 20 provides an example of how such a manipulated `script` tag might look like.

```
<script
↪  src="//foo.com/jsonp?cb=fetch('//attack.com/leak?val='+document.cookie)//">
</script>
```

Listing 20: Example of exploiting JSONP to leak the user's session cookie

5.1.2 Open Redirects

A web site or application is vulnerable to *Open Redirects* if it redirects the user accessing the site to a user-controllable URL that is not sufficiently checked by the server. This may be given as a query parameter, for instance like in the URL `https://redirect.com?redirect=https://example.com`. Assuming such an

endpoint is used in an application to load a script, and the attacker can control the `src` attribute of a loaded script, the attack surface reduction provided by the `SanitizerLibrary` can again be circumvented. The redirection endpoint would be allowed by the `prefixes` directive, which ignores the query parameter and would thus allow *any* script to be loaded via this Open Redirect.

5.1.3 Attacker Model Assumptions

The attacker model that we assumed for this work, as described in section 3.1, is somewhat limited. In particular, we made the simplifying assumption that an attacker only controls *one* consecutive sink API call. This, however, allows an attacker who can control more than one consecutive HTML sink API call to circumvent the input's sanitization. For instance, if they can control arbitrarily many consecutive `document.write` calls, they can instead of adding a new malicious `script` tag in just one call, add the tag character-by-character. Hence, the `TrustedHTML` sanitizer never observes any kind of `script` tag that it would have to check against the given config, but the malicious script is still added to the HTML of the site and executed.

5.1.4 Data loss

As explained in section 3.3, we used a logging policy with a *report-only* CSP to collect data. Since the logging policy simply returns the given input, no CSP violations should occur unless the policy is not present. However, during data collection, we encountered a considerable number of such violations. We observe that a large number of these violations occur in subdomains of `safeframe.googlesyndication.com`, which are loaded as iframes on sites, as well as from `null` origins, which also only occur in frames, implying that our current hooking approach is not yet completely sufficient to inject the logging policy in all frames during data collection.

Hence, we lose a non-negligible amount of data during collection. Furthermore, the crawler module is never *logged into any site* that it visits, assuming that said site has login functionality. However, the site might perform different or additional sink API calls for a logged-in user, which again is data that our collection approach currently misses.

5.1.5 Inaccuracy of Party Detection

During data collection, the party that performed the call to the sink API is determined. Currently, this is done by iterating the stacktrace of the call upwards until a non-empty URL is found. This may lead to the calling party being reported as the JQuery library [9], which provides developers with APIs that allow for simpler interactions with the DOM. In particular, JQuery also provides sink APIs that internally call native JavaScript sinks, making JQuery a sink call redirector rather than a true third-party. The true caller party is thus the invoker of JQuery rather than JQuery itself, which is currently not considered during data collection.

5.1.6 Null origins

Any iframe observed during data collection that possesses an `about:` or `javascript:` URL as its `src` attribute provides an origin value of `null`. However, since configs are generated per *origin*, all sink API accesses that happen in any such iframe are grouped together into one single config, regardless of the site in which the frame was originally loaded. This makes the config enforced in these frames much more lenient than they need to be, potentially even exposing bypasses across all frames if e.g. one of these frames contains an *Open Redirect* as described in section 5.1.2.

5.2 Problems of Trusted Types

We would like to note that the current implementation of `Trusted Types` displays some odd behavior. For context, it is possible to specify the content of a dynamically created script via different properties, one of them being `scriptElement.innerHTML`. However, since this property is usually used on other `HTMLElements` to set HTML content rather than script text, setting it on a `scriptElement` will cause `createHTML` to be invoked instead of `createScript`, despite the input not being actual HTML markup. Even so, the browser invokes `createScript` once the script is actually appended to the DOM, meaning that this is not an obvious security flaw. The problem may come from the fact that `createHTML` is not supposed to work with pure JavaScript input. As that is usually just text without any HTML tags inside, nothing should be removed during the accidental `createHTML`, but it may still happen that the call fails at some point, e.g. due to the `DOMParser` not being

```
trustedTypes.createPolicy('default', {
    createScript: (s, type, sink) => {
      return s;
    },
    createHTML: (s, type, sink) => {
      return null;
    }
});
let script = document.createElement('script');
// example 1
// this invokes createHTML and fails
script.innerHTML = 'console.log(1)';
// this invokes createScript if script.innerHTML succeeded
document.body.appendChild(script);

// example 2
// this invokes createScript and succeeds
script.innerText = 'console.log(2)';
```

Listing 21: An example of the script.innerHTML oddity

able to parse the JavaScript input, which would cause the entire script execution to fail since the setting of the content via `innerHTML` would fail, causing some functionality disruption. Listing 21 gives an example of the described odd behavior.

```
let actual = trustedTypes.createPolicy('foo', {
  createHTML: (input, type, sink) => {
    return null;
  }
});

r = document.createElement("iframe");
r.src = 'about:blank';
document.head.appendChild(r);
win = r.contentWindow;
let attack = win.trustedTypes.createPolicy('foo', {
  createHTML: (input, type, sink) => {
    return input;
  }
});
// not executed
document.write(actual.createHTML("<script>alert(1)</scr" + "ipt>"));
// will be executed
document.write(attack.createHTML("<script>alert(1)</scr" + "ipt>"));
```

Listing 22: An example of a `Trusted Types` bypass using a new policy

Additionally, we were able to find a general bypass for the security provided by `Trusted Types`. If one is able to inject content into an `about:blank` iframe, one may register a new policy which simply returns its own input, and can then use this new policy to pseudo-sanitize their own sink inputs later, giving arbitrary access

to sinks again and circumventing the original `Trusted Types` policy. Listing 22 provides an example of the described bypass. However, it should be noted that the bypass is not necessarily a part of the target of `Trusted Types`, since it requires that an attacker can inject content into an iframe and to later be able to specify their own policy for sanitization, which necessitates a server-side code injection to some degree, an issue which is an explicit non-goal [19] of `Trusted Types`.

5.3 Security and Functionality Tradeoff

As we have previously described in section 1, the primary goal of our work is reducing the attack surface exposed by third-party code while still allowing all benign sink access unaltered. First, it should be noted that in our current simple implementation, no difference between first-party and third-party is made, meaning that the configs are generated and enforced also for any *first-party*. This is, however, if anything beneficial, since it also reduces the attack surface exposed by first-party code as a by-product.

We would also like to clarify why complete prevention of client-side XSS is not generally possible using our approach. The problem comes from the fact that we must often allow a *range* of values in order to preserve functionality. This means that any value in this range as well as any allowed constant is still available to the attacker, potentially causing a bypass. This becomes even worse when the `allow-any` directive is used, as then for all parties that have this flag set, an attacker is again able to inject arbitrary content and execute code. However, this only makes the code of these particular parties vulnerable, assuming they really suffer from an injection at all. Still, in such cases, we have only set the `allow-any` flag as it was necessary to account for the highly dynamic inputs observed, which is the trade-off between keeping functionality intact and potentially exposing a vulnerability. A similar problem can be seen for the usage of the `prefixes` directive: It currently causes the sanitizers to disregard any query parameters of given URLs. This can potentially cause bypasses, such as the previously discussed JSONP exploit, or the issue of `Open Redirects`. However, that is, too, a trade-off made to keep functionality intact, as query parameters of scripts may be dynamically generated in benign use cases and allowing only *previously seen* values would not properly account for future benign values, similar to how we have observed it to be the case for dynamic path values in section 4.2.2.2.

Chapter 6

Conclusion

In this work, we made use of the `Trusted Types` mechanism to enforce the principle of least privilege on first-party and third-party code to reduce the attack surface it exposes against client-side XSS attacks. This attack surface is exposed due to usage of attacker-controllable values in dangerous sink APIs that convert strings to code and execute said code. To this end, we developed an infrastructure that uses observed benign sink inputs to automatically generate configs which describe what sink inputs should be allowed in future sink accesses, and build a JavaScript library called `SanitizerLibrary` that enforces the configs on a given site.

Additionally, we used our infrastructure to collect real sink input data from the top 100 Tranco domains, and determined from this data that HTML sinks are most often only used to write HTML markup that does not actually contain any code. We further found that some sink accesses that are covered by `TrustedScript` are also only used to obtain data, in particular, to parse JSON from a string.

Using the collected data, we further generated configs and evaluated the functionality-preservation given by the configs over a span of ten days, as well as the runtime overhead exhibited on the site by the `SanitizerLibrary`, both again using the top 100 domains from the Tranco list. We found that functionality is preserved perfectly against the data from which the configs were generated, and that is also preserved well over the span of 10 days, never going below 94% of benign inputs being unaltered by sanitization. Even so, the functionality-preservation rate did in fact decrease over time, implying that configs must be re-generated in some regular time spans, and it is also not easily possible to predict how much the inputs that are indeed rejected or altered impact the functionality of the site. Considering

runtime, we found that the config enforcement exhibits a low overhead of less than 4% on the loading time of sites, while the invocations of the `SanitizerLibrary` were observed to take less than two milliseconds on average, though it can take a lot longer if the given inputs are huge samples of HTML markup. Additionally, the vanilla configs could be generated for *all* parties, meaning that during enforcement the attack surface exposed by all of these parties is significantly reduced.

Lastly, we discussed limitations of our approach as well as the tradeoff between security and functionality that we must make. As we aim to preserve functionality, we must allow a certain range of values, which may in some scenarios lead to exposure of a vulnerability. Even so, this is precisely the reason why our goal is not to prevent exploitation of client-side XSS in an application altogether, but to rather only reduce the exposed attack surface as much as possible.

Overall, we have shown that it is indeed feasible to use `Trusted Types` in an automated manner to reduce attack surface. While it may not necessarily be enough on its own to completely defeat client-side XSS in the wild, it is certainly another building block that could help to combat XSS even in the presence of usage of very dynamic third-party code.

6.1 Future Work

In the following, we give an outlook on possible future extensions of our work.

First, as we have discussed in section 5.1.4, there is a significant number of CSP violations that occur during the data collection process. It may be worthwhile to revisit the reasons for this occurrence and try to determine why they occur and how to fix them. Doing so would provide more input data that could be used to get a more accurate picture of the sink access behavior in the wild as well as allow generation of more accurate configs.

As we have further described in the results section 4.1, the data set we used in our evaluation only considered the top 100 origins from the Tranco list. For our small evaluation, this was sufficient, but it may be interesting to perform the data collection, config generation and evaluation with a bigger data set, e.g. top 1000 origins with some depth. This would not only allow to potentially find interesting edge cases that were missed due to the small-scale evaluation, but also determine

whether some of the findings from the small dataset can be generalized. For instance, we described in section 4.1.3 that we did not find any `data:` URLs when analyzing the `TrustedScriptURL` dataset, which is the reason as to which during config generation for `TrustedScriptURL` inputs, we simply saved as hash of the `data:` URL's content if one should be found. A larger-scale analysis may show that `data:` URLs are not only really used in the wild, but potentially that their content is also dynamically generated. This information could be used to rework the `dataHashes` directive to also account for such dynamic behavior to better preserve functionality.

Though we could verify that the enforcement of configs with the `SanitizerLibrary` would considerably reduce attack surface under our assumed attacker model, we were not able to achieve an actual concrete evaluation of the reduction with real world data. To this end, it may be considered in the future to collect a set of client-side XSS vulnerable sites as well as sample exploits for these flaws using taint tracking, and then testing whether enforcement of the configs does indeed prevent the exploitation of these flaws with the sample exploits.

Considering the evaluation of functionality it may be interesting to further consider generating configs from more than two data sets and testing how well they perform over time. Clearly, usage of such multi-snapshot configs would improve the functionality achieved to some extent, as it simply causes more values to be allowlisted. However, it may be worthwhile to concretely investigate by what ratio usage of configs from more than two data sets improves functionality preservation over time. Still, it is likely that with increasing size of the configs, the attack surface reduction becomes smaller. This could be further verified by performing a security evaluation like described in the previous passage. Furthermore, the entire functionality evaluation could be performed over a greater time span to observe how well the configs perform over a larger amount of time and how often they should be re-generated to preserve functionality decently.

As described in section 5.1.5, the way in which the calling party is currently determined during data collection is somewhat flawed in regards to usage of the JQuery [9] library. It might falsely determine said library as the sinks calling party, while the true caller made use of one of JQuery's APIs which only internally itself called a sink. Thus, during data collection when the calling party is determined, one should ignore any such JQuery library. To this end, one could use *retire.js* [15], which allows for detection of some JQuery versions within code. These parties

could then be put into the `ignoreList` directive of each config as well, which would cause the `SanitizerLibrary` to also ignore them during its own party detection.

Furthermore, we have seen in sections 5.1.1 and 5.1.2 that JSONP endpoints and Open Redirects can potentially cause bypasses to the reduced attack surface that our infrastructure provides. It may be worthwhile to further investigate which exact domains suffer from such vulnerabilities and to then modify the config generation for these domains such that the configs for these domains are stricter to prevent the flaws from being exploited.

Lastly, as we have shown in section 4.2.2.2, many of the inputs corresponding to `TrustedScriptURL` that were rejected during our functionality evaluation experiment were rejected due to having a different path than the allowlisted value, but having an identical structure. One could rework the generation of allowlists for the `prefixes` directive such that it not just simply allowlists observed path values, but instead attempts to construct regexes for all possible dynamic variations. However, if done so, one should consider that this also potentially opens up more attack surface as more scripting content can be included in case of an injection vulnerability.

Abbrevations

HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
XSS	Cross-Site Scripting
SOP	Same-Origin-Policy
DOM	Document Object Model
CSP	Content Security Policy
JSON	JavaScript Object Notation
JSONP	JSON with padding

List of Figures

List of Tables

Bibliography

[1] Homepage of the internet archive. https://archive.org/. Last visited on 06/03/2021.

[2] Python beautiful soup library for parsing html. https://www.crummy.com/software/BeautifulSoup/bs4/doc/. Last visited on 17/02/2021.

[3] Chrome devtools protocol. https://chromedevtools.github.io/devtools-protocol/. Last visited on 16/02/2021.

[4] Csp specification part explaining csp inheritance in srcdoc iframes. https://w3c.github.io/webappsec-csp/2/#which-policy-applies. Last visited on 16/02/2021.

[5] Mdn webdocs entry for data: Urls. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs. Last visited on 28/02/2021.

[6] client-side domparser utility for parsing html strings. https://developer.mozilla.org/en-US/docs/Web/API/DOMParser, . Last visited on 18/02/2021.

[7] Dompurify, a library for sanitizing html. https://github.com/cure53/DOMPurify, . Last visited on 18/02/2021.

[8] Javascript object notation. https://esprima.org/. Last visited on 17/02/2021.

[9] Homepage of the jquery library. https://jquery.com/. Last visited on 15/02/2021.

[10] Javascript object notation. https://www.json.org/json-en.html, . Last visited on 17/02/2021.

[11] Availability of the json parsing api in browsers. https://caniuse.com/?search=JSON, . Last visited on 25/02/2021.

[12] Github repository for trusted types. https://github.com/w3c/webappsec-trusted-types. Last visited on 28/02/2021.

[13] Source code of the pmforce infrastructure. https://github.com/mariussteffens/pmforce. Last visited on 28/02/2021.

[14] Github repository of the puppeteer framework. https://github.com/puppeteer/puppeteer. Last visited on 16/02/2021.

[15] Tool to detect vulnerable library versions used in code. https://retirejs.github.io/retire.js/. Last visited on 27/02/2021.

[16] Homepage of the secure web applications group at cispa. https://swag.cispa.saarland/. Last visited on 16/02/2021.

[17] Homepage of the tranco list. https://tranco-list.eu/. Last visited on 16/02/2021.

[18] Trusted types javascript: Url pre-navigation check. https://w3c.github.io/webappsec-trusted-types/dist/spec/#require-trusted-types-for-pre-navigation-check, . Last visited on 18/02/2021.

[19] Non-goals of trusted types. https://w3c.github.io/webappsec-trusted-types/dist/spec/#non-goals, . Last visited on 28/02/2021.

[20] Trusted types default policy specification. https://w3c.github.io/webappsec-trusted-types/dist/spec/#default-policy-hdr, . Last visited on 18/02/2021.

[21] Python library for parsing urls. https://docs.python.org/3/library/urllib.parse.html. Last visited on 17/02/2021.

[22] Csp specification. https://www.w3.org/TR/CSP/, 2018. Last visited on 15/02/2021.

[23] Ecmascript language specification. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/, 2020. Last visited on 15/02/2021.

[24] Overview of trusted types availability in browsers. `https://caniuse.com/trusted-types`, 2021. Last visited on 15/02/2021.

[25] Standard of the hypertext markup language. `https://html.spec.whatwg.org/`, 2021. Last visited on 15/02/2021.

[26] Trusted types editor's draft. `https://w3c.github.io/webappsec-trusted-types/dist/spec/`, 2021. Last visited on 15/02/2021.

[27] A. Barth and U.C. Berkeley. Standard describing http cookies and the set-cookie header. `https://tools.ietf.org/html/rfc6265`, 2011. Last visited on 15/02/2021.

[28] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. Jsand: complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[29] Aurore Fass, Michael Backes, and Ben Stock. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[30] Amit Klein. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles*, 4, 2005.

[31] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[32] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out domsday: Towards detecting and preventing dom cross-site scripting. In *2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[33] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. Scriptprotect: Mitigating unsafe third-party javascript practices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367523. doi: 10.1145/3321705.3329841. URL `https://doi.org/10.1145/3321705.3329841`.

[34] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[35] R. Fielding, Ed. and J. Reschke, Ed. Standard of the hypertext-transfer-protocol, describing semantics and content of http messages. `https://tools.ietf.org/html/rfc7231`, 2014. Last visited on 15/02/2021.

[36] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*, 2020.

[37] Marius Steffens and Ben Stock. Pmforce: Systematically analyzing postmessage handlers at scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[38] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. 2019.

[39] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against dom-based cross-site scripting. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014.

[40] Ben Stock, Stephan Pfistner, Bernd Kaiser, Sebastian Lekies, and Martin Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015.

[41] Ben Stock, Benjamin Livshits, and Benjamin Zorn. Kizzle: a signature compiler for detecting exploit kits. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016.

[42] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in) security. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017.

[43] Mike Ter Louw, Karthik Thotta Ganesh, and VN Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security Symposium*, 2010.

[44] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Webjail: least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[45] Michael Walfish et al. Treehouse: Javascript sandboxes to help web developers help themselves. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012.

[46] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.