Trust Me *If You Can* – How Usable Is Trusted Types In Practice?

Sebastian Roth, *TU Wien;* Lea Gröber, *CISPA Helmholtz Center for Information Security;* Philipp Baus, *Saarland University;* Katharina Krombholz and Ben Stock, *CISPA Helmholtz Center for Information Security*

https://www.usenix.org/conference/usenixsecurity24/presentation/roth

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Trust Me *If You Can*
How Usable Is Trusted Types In Practice?

Sebastian Roth
sebastian.roth@tuwien.ac.at
TU Wien

Lea Gröber
lea.groeber@cispa.de
CISPA Helmholtz Center for Information Security

Philipp Baus
s8phbaus@stud.uni-saarland.de
Saarland University

Katharina Krombholz
krombholz@cispa.de
CISPA Helmholtz Center for Information Security

Ben Stock
stock@cispa.de
CISPA Helmholtz Center for Information Security

Abstract

Many online services deal with sensitive information such as credit card data, making those applications a prime target for adversaries, e.g., through Cross-Site Scripting (XSS) attacks. Moreover, Web applications nowadays deploy their functionality via client-side code to lower the server's load, require fewer page reloads, and allow Web applications to work even if the connection is interrupted. Given this paradigm shift of increasing complexity on the browser side, client-side security issues such as client-side XSS are getting more prominent these days. A solution already deployed in server-side applications of major companies like Google is to use type-safe data, where potentially attacker-controlled string data can never be output with sanitization. The newly introduced *Trusted Types* API offers an analogous solution for client-side XSS. With Trusted Types, the browser enforces that no input can be passed to an execution sink without being sanitized first. Thus, a developer's only remaining task – in theory – is to create a proper sanitizer. This study aims to uncover roadblocks that occur during the deployment of the mechanism and strategies on how developers can circumvent those problems by conducting a semi-structured interview, including a coding task with 13 real-world Web developers. Our work also identifies key weaknesses in the design and documentation of Trusted Types, which we urge the standardization body to incorporate before the Trusted Types becomes a standard.

1 Introduction

The effect of Cross-Site Scripting (XSS) and its mitigation techniques have been studied extensively in recent years. For more than two decades XSS has been present in the OWASP Top 10 Web Application Security Risks [44], showing that XSS is here to stay. While the server-side part of the problem already has proper mitigation techniques, such as type safety [27] or automatic encoding [14], client-side XSS lacks proper solutions. However, prior work [26, 39, 41, 55, 56, 57, 59] has shown that client-side XSS vulnerabilities are on the rise.

A common mitigation strategy for XSS is the usage of a Content Security Policy (CSP). However, prior work has already shown that deployment of this lacks behind [9, 50, 66, 67] and developers even are unaware of the differences between client- and server-side XSS when attempting to grasp the threat models covered by CSP [51]. Nevertheless, to transfer the concept of type safety to the client side, Google proposed the Trusted Types API [33], a new addition to CSP. With this mechanism, it is not possible to invoke dangerous JavaScript APIs without passing the input through sanitization functions that the developers need to specify themselves. However, at the point where the World Wide Web Consortium (W3C) has discussed shipping Trusted Types as a First Public Working Draft (FPWD) [16], browser vendors raised concerns. Specifically, they noted that the complexity of self-creating sanitization functions for HTML, JavaScript, and script URLs, may be too difficult for the majority of developers. However, the claim of the browser vendors lacks a scientific foundation because the actual usability of Trusted Types deployment has not yet been evaluated. Still, in December 2023, Mozilla changed their opinion on Trusted Types such that it now will become a Technical Specification [4]. A timeline of the events around TT is depicted in Table 4.

To close the research gap regarding TT's deployment process and provide valuable input in the ongoing standardization process, the objective of our work is to get insights into each step of the deployment process of Trusted Types and investigate the developer's knowledge about the underlying issue, namely client-side XSS. Therefore, we designed a qualitative interview study in a "bottom-up" approach [24] that includes a coding task where the participants deploy Trusted Types for a small Web application. Throughout our study, we evaluate the participant's knowledge of the different dimensions of XSS and their knowledge of existing mitigation techniques. In addition, our iterative study process allowed us to get fine-grained insights into all deployment steps, allowing us to evaluate the entire deployment process of Trusted Types.

Our work first presents a comprehensive qualitative interview study methodology to evaluate the usability of Trusted

Types (Section 3). With 13 participants, we were able to uncover important roadblocks of Trusted Types deployment, such as misconceptions introduced by information sources or problems caused by same-origin iframes (Section 4). Based on our findings, we discuss and propose strategies on how to successfully deploy Trusted Types, while highlighting the pitfalls developers might face. Also, we present general improvement suggestions for the Trusted Types standard, like the inheritance of Trusted Types sanitizers for same-origin iframes (Section 5). With the improvement suggestions, we want to ease the deployment process of Trusted Types and influence the W3C's standardization process. Also, we hope that the W3C and research community sees this work as a role model to test working drafts with real-world Web developers before releasing them and fully implementing them in browsers.

Availability. For transparency and reproducibility, we provide a full replication package [11], including our survey and coding task (incl. custom third parties) Web apps.

2 Background & Related Work

This section describes details about the underlying threat model of Trusted Types, the security mechanism itself, as well as a brief overview of qualitative methodologies used in usable security research.

2.1 Cross-Site Scripting

To protect the content of a Web page, the Same Origin Policy (SOP) ensures that only documents with the same origin [3], i.e., same protocol, hostname, and port, can access each other. Notably, if attackers manage to execute JavaScript in the origin of a vulnerable Web site, this script has the same capabilities as a script that is served legitimately by the page itself. Thus, the attacker's script can steal sessions or login credentials, impersonate user actions, or change the page content to distribute fake news. Since its initial discovery back in 1999 [48] numerous publications and blog posts investigated the different types of Cross-Site Scripting, such as XSS through content sniffing [47], client-side (also called *DOM-based*) XSS [29], universal XSS [23], self XSS [52], scriptless XSS [20], mutation-based XSS [21], same-origin method execution [19], gadget-based XSS [40], or persistent client-side XSS [57]. In general, however, XSS can either be persistently stored or only reflected per request, while the vulnerable code is either located on the server or the client side. The plethora of different types of XSS, as well as its omnipresence in annually published OWASP Top 10 Web Application Security Risks [44], shows that XSS is a problem that is still not solved nowadays.

2.1.1 Server-Side Cross-Site Scripting

The first discovered type of XSS flaw was server-side XSS [48], i.e., the vulnerable code is located on the server side. In this case, parts of the request, such as the URL parameters (reflected), or stored user content, such as profile descriptions (persistent), are mixed up with the developer-specified markup in the response. This way, an attacker can inject malicious markup such as *script* tags that are then delivered via the server of the vulnerable application, which causes the script to run in the context of this Web application.

Notably, server-side XSS can be mitigated or even defended against by enforcing type safety on the backend such that a string (or XSS payload) can never end up in a response unfiltered [27]. Also, many popular frameworks, such as the Python Django Framework, automatically encode, and thus sanitize, every string that is programmatically passed to the template rendering engine [14]. Thus, a developer needs to explicitly use the `mark_safe` method to cause a server-side XSS vulnerability. Also, academic solutions such as Noxes [28] can be used to prevent XSS. Noxes uses a set of rules to detect XSS attempts. For example, it requires user interaction for all dynamically crafted links to enable the user to make either temporary or permanent security decisions.

2.1.2 Client-Side Cross-Site Scripting

With the advent of more complex client-side JavaScript code, XSS vulnerabilities can also occur in vulnerable client-side code. Also here, an injection can happen either in a reflected fashion (e.g., by using data from the URL via `location.href`) or payloads can originate from a persistent client-side storage such as cookies or the `localStorage` API. Either way, the vulnerability is that the attacker code from these locations at some point ends up in a JavaScript sink that can execute code such as `innerHTML` or `eval`.

For reflected client-side XSS, multiple works [39, 41, 59] used a modified browsing engine to taint dangerous flows from a URL into executing sinks, to show that client-side XSS is a prevalent problem even in the top most visited Web applications. Steffens et al. [57] also used taint-tracking to show the prevalence of persistent client-side XSS vulnerabilities in the top 5,000 Web sites. They showed that 21% of the sites that make use of data originating from client-side storages are vulnerable to persistent client-side XSS.

2.2 Content Security Policy

A Web site's operator can also deploy a proper Content Security Policy header to mitigate XSS [68]. CSP allows the developer to specify the sources from which scripts can be loaded through the `script-src` directive. This way, even in the presence of an injection flaw, the attacker is limited to relying on scripts that are allowed by the developer. By default, CSP also forbids the usage of inline scripts, event handlers,

and the dreaded `eval` functionality. However, setting up a restrictive and functional CSP has shown to be a hard task due to third-party behavior [58], or the unusability [51] of the mechanisms itself. Over the years, numerous papers have shown that the vast majority of all policies in the wild are trivially bypassable [9, 50, 66, 67]. And even those policies that are considered meaningful can be bypassed via JSONP [66], script-gadgets [40], or open redirects [49].

While in theory, a perfect CSP would mitigate the effect of server- and client-side XSS attacks alike, such a policy is hard — if not impossible — to craft for real-world Web applications. In particular, for client-side code, sites often rely on third parties for important functionality such as maps or ads. However, as shown by Steffens et al. [58], these third parties often require the usage of, e.g., `eval`. However, with CSP, developers lack fine-grained per-party control over such features: they can only decide to either allow all executions of `eval`, which makes the application prone to client-side XSS if the string-to-code conversion happens with attacker-controlled input or to deny all string-to-code conversions, which results in the loss of functionality. While academic approaches to overcome this issue exist [43], these are limited in both functionality and security and lack deployment in the wild.

2.3 Trusted Types

Although the problem of client-side XSS and its prevalence in Web applications is well understood, a proper defense or mitigation technique is still missing. For example, Google engineers enforce type safety on the backend such that dangerous markup can never end up in the body of a response without explicit sanitization [27]. To transfer this defense idea to the client side, Google suggested Trusted Types [33] to filter values before they flow into dangerous JS sinks such as `eval` or `innerHTML`. To deploy Trusted Types, a developer needs to set the CSP's `require-trusted-types-for` directive to `'script'`, enabling the enforcement by the browser. In addition, they can provide a list of allowed sanitizer policy names through the `trusted-types` directive to control the number of sanitizers. Thus, enabling Trusted Types via the CSP header only requires a fixed header to be deployed and does not require additional configuration of the header.

In addition, as shown in Listing 1, the boilerplate code for registering a sanitizer is simple. The challenging part is to implement the actual functional yet secure sanitizers (here `sanitize{HTML,JS,URL}`). Because we want our participants to focus on the implementation of the sanitizers, and because our prestudy (see Section 3.5) showed that deploying the header is just a copy & paste task, we set the fixed header value for the participants in our main study. Notably, Klein et al. [30] have already investigated the quality of deployed HTML sanitizers in the wild and classified 88 sanitizers used on 102 different domains as bypassable and thus insecure,

```
window.TT = trustedTypes.createPolicy('default', {
    createHTML: rawHTML => sanitizeHTML(rawHTML),
    createScript: rawJS => sanitizeJS(rawJS),
    createScriptURL: url => sanitizeURL(url)
});
```

Listing 1: Trusted Types API example.

emphasizing the importance of focussing on the sanitizers. We want to extend this by investigating why developers fail to deploy sanitizers for HTML, JS, and URLs in order to reveal possible deployment roadblocks for TT. The results of Klein et al. [30] and several vulnerabilities in the wild caused by manually crafted and broken sanitization have shown that developers should rather use established sanitizer libraries instead of self-crafted sanitizers. However, in many cases, custom sanitization is the only viable approach for deploying TT. Ready-made sanitizers like DOMPurify [13] or the upcoming HTML Sanitizer API [5] can only remove all code. This lack of customization paired with the behavior of third parties to inject content that contains (inline) JavaScript [58] requires the developer to write custom sanitizer functions as TT requires them to allow the specific code from their third parties but disallow malicious code. Each site has unique requirements caused by the first- and third-party code it runs, requiring a TT sanitizer to be highly custom per site.

When enabled, Trusted Types enforces that *all* dangerous client-side APIs that might lead to code execution can only be used with a Trusted Types Object. To create those objects, the developer can explicitly call the corresponding Trusted Types sanitizer function, e.g., `TT.createHTML(input)`. Alternatively, they can define the special *default* policy, whose respective sanitizers are implicitly invoked by the browser when unsanitized data is passed to a dangerous sink. In a 2021 document, Kotowicz [32] reported on Google's success in deploying Trusted Types on 130 of their services. Furthermore, the report mentions that Google faced no client-side XSS in their Trusted Types protected applications. The report also mentioned that new tools and better framework support can ease the deployment. In fact, Wang et al. [65] showed how Trusted Types can be incorporated into Web frameworks such as AngularJS. Hence, Trusted Types seem to have the potential to be the panacea that client-side XSS requires.

2.4 Qualitative Methods

Given that we aim to explore the origins of behaviors and misconceptions of a new mechanism, we chose the "bottom-up" approach of a qualitative study design [24].

In the area of IT security, the qualitative analysis of semi-structured interviews was already used to better understand problems, misconceptions, and mindsets of developers [37, 51], users [22, 25], or administrators [36] of IT systems. Also qualitative analysis of online discussions about code in

different languages [12] or the usage of cryptography libraries [45], or an analysis of Rust-related Stack Overflow questions [69] have shed light on those problems.

While the qualitative analysis of interviews or online resources is suitable for exploring a topic and investigating concepts and trains of thought of participants on topics that they are already familiar with, the investigation of previously unknown scenarios needs a more controllable study setting, namely lab studies. Here, the researcher can adjust the settings of the study to observe if certain changes have possible effects on the investigated scenario. In order to get qualitative data out of those lab tasks, the participants can be asked to think aloud such that the process sheds light on, for example, the motivations behind certain decisions. For example, Acar et al. [1] conducted an online coding task to access the usability and security of cryptographic libraries. While Krombholz et al. [35] performed a lab study and interviews to assess challenges in deploying TLS, Tiefenau et al. [61] conducted a lab study to evaluate the usability and security benefits of using a semi-automated certificate generation for TLS, and Mindermann et al. [42] conducted a lab study to evaluate the usability of Crypto APIs in Rust.

To the best of our knowledge, the only other work that specifically tackled the challenges of deploying a client-side Web security mechanism has been conducted by Roth et al. [51]. They used a lab study to investigate the deployment process of a widely used security mechanism, namely CSP. They conducted remote coding interviews with twelve developers to find strategies, roadblocks, motivations, and misconceptions regarding CSP deployment through a rather artificial setting with an application intentionally containing CSP-hindering features. Instead of artificially increasing the task's complexity, we instead rely on the usage of real-world third parties to maximize the ecological validity of our results. Also, in contrast to the fully standardized CSP, Trusted Types is still a Working Draft. Thus, a usability evaluation of this new mechanism has the potential to improve the mechanism before its widespread deployment would cause breaking changes.

3 Methodology

A GitHub discussion [16] of the World Wide Web Consortium (W3C) showed that some browser vendors oppose the standardization and deployment of Trusted Types because they claim it is too complicated for the long tail of sites. Therefore, Trusted Types, as a new mechanism, is currently only supported by Chromium-based browsers (from version 83, May 2020) [10]. This work aims to uncover problems with the current version of the mechanism and evaluate its ease of deployment. This can aid the W3C to improve the mechanism to enable developers to mitigate client-side XSS attacks.

Due to the youth of the mechanism and its limited support in modern browsers, the adoption rate of the mechanism is negligible. Therefore, the targets for our study are not Web developers with experience with the mechanism, as it is nearly non-existent, but rather the average Web developer who might potentially use Trusted Types in the future. Together with them, we want to get detailed insights into the deployment process of Trusted Types from a controlled coding task. In this way, we can uncover roadblocks and develop deployment strategies for the mechanism.

Thus, our work has the following research goals: **(1)** What roadblocks do developers face when implementing Trusted Types sanitizers and which strategies do they apply? (Section 4), and **(2)** How can the standard and overall deployment process of Trusted Types be improved and what are the biggest challenges? (Section 5)

3.1 Recruitment and Participants

To maximize the ecological validity of our results, the target population for our study is real-world Web developers who have experience with the development and deployment of Web sites. Given that we want to study the deployability of Trusted Types for all sorts of Web applications and given that Trusted Types is not widely used by Web sites anyways, we not only considered professional Web developers but also non-professional Web developers, for example, students, as long as they mention prior Web development experience during the screening survey. Since Web developers are a hard-to-recruit population [51], we relied on a combination of different recruitment strategies to optimize outreach. First, we posted a recruitment flyer on our institution's social media channels. The flyer contains details and timelines of the study procedure as well as information about us and the compensation for the study [53]. In addition to that, multiple researchers from our team presented talks about Web security at multiple industry-focused developer conferences. Those talks included a call for participants for this study so that we could have direct contact and immediately answer questions from the Web developers who showed interest. Also, we used contacts of researchers of our institution to industry and asked those people to forward our recruitment Web site to their internal team chat or in their Web development communities. We also posted our call for participation on the platforms `freelancer.com` and `upwork.com` to also get hands-on Web developers who are not permanently employed in a development team. In general, irrespective of the recruitment procedure, we always mentioned that in addition to the monetary compensation of 50€, the participants could learn something about client-side XSS vulnerabilities and how they can mitigate this issue for their Web applications.

3.2 Screening Survey

To ensure a homogeneous set of participants who are all security aware and have sufficient experience in Web development, we conducted a screening questionnaire before inviting

selected participants to the actual interview. This survey included 12 fields, where ten were actual questions about the demographics of the participant, plus a field for the contact email address as well as a feedback/issue field for the survey. We developed the corresponding survey app ourselves and self-hosted it under a subdomain of our institution to ensure full and exclusive control over the data entered into the survey.

The landing page of the survey informs the participants about the approximate time required for the survey (1-2 minutes) and the number of questions. Furthermore, they are informed that all data will be treated confidentially and that we will at no point disclose their identities or the sites they operate. The page also includes information on who we are and the terms and conditions of participating in the survey. In the survey itself, the participants are asked to (optionally) provide age, gender, current occupation, highest education, country of living, and the size of the company that they are working at. In addition to that, we also ask questions about their experience in deploying or maintaining Web security mechanisms, their IT security background, the Web presence of their company, and questions regarding the development team. With that information, we are then able to better assess the participant's eligibility for the study. Notably, all of the input fields, except for the contact email, are optional.

After the survey, we manually check the eligibility of the participants, and in case of a positive evaluation, invite them to the actual interview via email. To be eligible, participants must be a non-bot user, and their occupation should be related to Web development or one of the other answers (e.g., "Do you have any experience in deploying or maintaining Web security mechanisms? If yes, which?") should show that they have experience as a Web operator. The email consists of a short reminder regarding the schedule for the interview process, a link to the interview scheduling system of the Nextcloud instance hosted by our institution, and information about the interview setup. To give the participants a comfortable environment, we offer them the option to use any video conference software that they want, as long as it allows for screen sharing for the coding task. We also offer them different ways of setting up the Web application for the coding task. They can either use a docker image provided by us, can directly execute the Python Django code, or can remotely control the interviewer's machine (e.g., via Zoom or TeamViewer) to conduct the coding task.

3.3 Interview

In the first few minutes of our interview, the researcher gives the participant an introduction to the timeline of our interview session. The interview is partitioned into questions regarding the participant's working environment, some questions about Web security, the coding task, and a debriefing. The question set about the working environment is to check the security and Web development background of the participant but also to get the chat between researcher and participant rolling.

After the general questions, we bring the conversation to XSS to answer our question regarding the understanding of this vulnerability and the differences between client- and server-side XSS. Therefore, we explicitly asked our participants if they could explain to us how the vulnerability happens and about the differences between client- and server-side XSS. Here, we also try to find out if the participant knows client-side execution sinks for XSS attacks. Next, we chat about Web security mechanisms that come into the mind of the participants when they want to defend against XSS, which then transitions into questions regarding Trusted Types and its method of operation. The question block is topped off with questions about the impression of Trusted Types and if they would use it in one of their Web applications. Then, the coding task is performed, which we explain in detail in Section 3.4. After the coding task, a set of debriefing questions will follow. Based on the recent interactions with the deployment of Trusted Types the impression of the mechanism might have changed, which is why we ask about the feasibility of Trusted Types as a defense mechanism against XSS. The last question that we ask is about improvement suggestions for Trusted Types, to assess which is the biggest problem that currently hinders the successful deployment of Trusted Types. Finally, the participant needs to fill out a System Usability Scale (SUS) [7] to assess the usability of deploying Trusted Types. The SUS questionnaire contained ten questions on a five-point Likert scale.

3.4 Coding Task

In our coding task, we ask the participants to deploy Trusted Types for a small Web application, while the main work in the case of the deployment process is the implementation of the sanitizer functions. The Web application itself uses features, libraries, and frameworks that are frequently used by real-world Web applications to increase the ecological validity of the data we gathered throughout the coding task. We explain details of how we chose those in Section 3.4.1. Notably, throughout the coding interviews, the functionality of the Web application had to be changed to ease the initial deployment process such that we could get more fine-grained insights into the later deployment steps, but also to lower the level of frustration of the participants because they could get nowhere near to a proper solution. Due to our usage of real-world third parties for a better ecological validity of our results, we also had changes in the functionality of the application that were not driven by us but by third parties that changed the way their libraries operate during our study. We explain the exact changes of this iterative interview process in detail in Section 4.2.

Third-Party Site (eTLD+1)	Inclusions (# Sites)
google-analytics.com	2,626
googletagmanager.com	2,329
gstatic.com	2,311
doubleclick.net	2,162
google.com	2,132
facebook.net	1,632
youtube.com	1,571
googleadservices.com	1,359
googleapis.com	1,233
twitter.com	988

Table 1: Top 10 third parties by inclusions

Framework	Usage (# Sites)
jQuery	1,005
Bootstrap	358
Dojo Toolkit	79
Vue.js	75
Angular	68
Backbone.js	19
Ember.js	1

Table 2: Usage of JS frameworks

3.4.1 Web Application

The Web application's backend is written in Python 3.10 using the Django Framework. For Trusted Types, the actual backend language is less important because of the main work, i.e., the Trusted Types sanitizer being implemented in JavaScript. Also, as Trusted Types is enforced on the client side, client-side technologies used by Web sites are more important for our demo application than the server-side technologies.

To gather data about the usage statistics of third-party JavaScript we crawled the Tranco Top 5k sites [38, 62] with a Chromium browser instrumented by puppeteer [18]. Our crawl also collected the URLs present on the loaded page and visited those up to two levels of links from the start page, while crawling at most 500 URLs per site. During a page load, we waited a maximum of 20 seconds for the load event to be fired and then an additional 5 seconds before we continue. During this time, we capture all loaded script URLs.

In Table 1, we show the top 10 sites, by the number of including sites, that were loaded during our crawl. The most requested site by far was `google-analytics.com`, with 2,626 sites that included scripts from it. Due to this prevalence of the service in real-world Web applications, our demo application also uses Google Analytics.

Advertisements, specifically those from `doubleclick.net`, were also used frequently (on 2,162 sites). However, showing real advertisements to our participants would put costs without benefits on those who ordered the advertisement. These costs would not only be problematic from an ethical point of view but could also make us accountable for ad fraud. Therefore, instead of just ignoring the prevalence of advertisements in real-world applications, we decided to create a custom advertisement service for the Web application. Notably, a Trusted Types policy consists of three sanitizer functions. Thus, our coding task should require the participants to implement all three sanitizers. No other third-party service that we included used `eval` conversion (which requires the createScript sanitizer), although Steffens et al. [58] showed that in 58.5% of the sites, their third parties mandate the usage of unsafe-eval in the sites CSP. To not lose this part of the deployment, we designed an ad service that uses `eval` to ensure a realistic setting even without including actual real-world ads.

The biggest player in terms of social media integration was `facebook.net`. The majority of those page loads (742/1,632) were inclusions of the script that includes their widget. This is why we also included this feature in our application. Notably, 336 of the sites that used the Facebook integration also used the Twitter widget at the same time, which is why we also incorporated the Twitter widget in our Web application.

Another common third-party service used in Web sites is the integration of maps. Because the most widely used service here (Google Maps) needed a domain name to properly use all features – yet we only used a local setup for our study – we had to use a different service that supported our use case. Therefore, we used `openstreetmap.org` as maps integration. In addition to that, we also showed a news blog that used the third-party comment system `disqus.com`. As we discuss in Section 3.5, we removed these after the pre-study.

To determine the most frequently used client-side frameworks, we investigate both the included URLs as well as the script content to detect libraries. Table 2 shows that jQuery is by far the most used framework with over 1k sites that use it. Notably, the intersection of those sites with those from the second most used framework (Bootstrap) shows that 62.8% (225/358) of the sites that are using Bootstrap are also using jQuery. Thus, we used a combination of Bootstrap and jQuery to build our Web application.

3.5 Pre-Study

To ensure that our coding task and its setup do not contain errors and can be solved, such that no participant is frustrated after the interview, we conducted a pre-study involving two participants (P1 & P2). Both worked for an IT company, one as a developer and the other one in their security incident response team. Notably, both took Web-related university courses. By selecting participants who are presumably less trained in deploying features for real-world Web sites, we want to find a lower bound for the scope of our coding task.

Given that neither of the participants of the pre-study was close to sufficiently deploying Trusted Types for the Web application, we decided to remove some features from the Web site to ease the deployment. We based the decision of which of the third parties to drop on the results presented in Table 1. Thus, we removed the subpages that use *maps* integration and the `disqus.com`. We also added code snippets in the coding task to speed up the deployment. The pre-study showed that participants copy-paste the CSP header that enables Trusted Types and the skeleton for the sanitizer functions from online resources. As mentioned in Section 2.3 the header has a fixed value and does not require sophisticated configuration. Thus, by adding them to the source code files, we put the focus on the challenging part, the creation of the sanitizer functions, and at the same time we reduce the setup time for the coding task. The pre-study also showed that our interview guideline (see Appendix A) was sufficient to answer our research questions. From the transcribed interviews we were able to learn more about the developer's mental model of XSS, especially client-side XSS, and uncover roadblocks as well as strategies for Trusted Types deployment. In addition to that, the interviews also shed light on the participant's perceptions of Trusted Types and motivations and disincentives to deploy it.

3.6 Data Analysis

For the analysis of the collected data, we used the GDPR-compliant online transcription service `amberscript.com`. Here we used the *Human-made* mode which ensures that the transcriptions are created by professional transcribers and captioners. The first author who conducted the study discussed interesting concepts that arose during interviews or coding tasks with the others. Based on these, the first author built the codebook iteratively, with a technology-centric lens of analysis in an open coding approach [60]. To better analyze the results of the coding task and not miss important information, the recordings of the shared screen during the coding task were also considered during the analysis. We continued conducting interviews and analyzing them until no new concepts were added to the codebook from the latest two interviews. After reaching our criteria for saturation of our dataset, the main author handed over three interviews with the saturated version of the codebook (see Replication Package [11]) to two other authors to calculate the inter-coder reliability of our results. We selected three interviews that provide the broadest coverage of codes. The average inter-coder reliability Krippendorff α [34] was 0.987. We assume the near-perfect score is due to our codes capturing technical concepts, which does not leave much room for interpretation.

To better evaluate the mindset of our participants and to get a better understanding of the occurring roadblocks and the strategies that the developer chooses, our codes are segmented into high-level categories and more detailed low-level information. If, for example, a participant struggled with Trusted Types deployment due to a programmatically added sourceless iframe, we assigned the code "Roadblock: Same-Origin Iframe", where *Roadblock* is our high-level category and *Same-Origin Iframe* the low-level description. Following the methodology of previous research [45, 51, 54] to identify emerging concepts, patterns, and themes, we analyzed our data using thematic analysis [6]. We then used axial coding [60] to find relations between those concepts and patterns. For example, we analyzed the connection between roadblocks and the applied strategy to explore which factors in the deployment process can lead to the success or failure of the deployment of Trusted Types.

3.7 Ethical Considerations

To the best of our knowledge, we designed our interview, the coding task, and the data collection process always with the risks and benefits for the participant in mind. The participants had the freedom of choice to use their own machine for the coding task or to remote control the interviewer's machine. While using their own device is closer to the participant's normal and familiar coding behavior, the required installation of Python and Django might be perceived as invasive. However, they could have used the Dockerfiles to create an easy-to-remove docker image that runs our code.

In case the participants used a browser on their own device to access the Web application, the current IP address of the user is leaked to third parties as the app performs requests to Google Analytics, Twitter, and Facebook. We decided to still use those real-world third parties to maximize the ecological validity of our results.

Also, our participants, especially those from the pre-study, may have become exceedingly frustrated when they, after 90 minutes, were not even close to a proper solution. This was one of the main reasons why we cut down some of the features of our application so that we do not leave behind a long-lasting bad feeling of working with Trusted Types or deploying security features.

As noted before, we used Amberscript to transcribe the audio of our interview sessions. Notably, we uploaded the audio (not the video) track to only reveal the necessary minimum to the service. Also, Amberscript is fully compliant with GDPR and advertises their service by guaranteeing full confidentiality, transparency, and non-disclosure agreements of all the transcribers [8]. In addition to that, all participants gave their electronic consent in the initial survey and again verbal consent to our data collection and processing methods at the beginning of the interview. All data (on our side and on Amberscript) was processed and stored in compliance with the General Data Protection Regulation (GDPR), and the entire methodology of our study and data collection processes have been approved by our Ethical Review Board (ERB).

4 Results

In this section, we present the results of our analysis, starting with the demographics of our participants. We then elaborate on the participant's mindset regarding the difference between client- and server-side XSS, highlight the different roadblocks that we investigated for Trusted Types deployment, and shed light on different deployment strategies and their success.

Without the pre-study participants, the average length of an interview was 01h 23min 09s (Median: 01:24:20, Min: 01:02:25, Max: 01:44:50). The first non-pre-study interview was conducted in July 2022, the last one in April 2023.

4.1 Participant Demographics & Background

Our main study population consisted of 13 participants (12 male + one female). Their age ranged from 20 to 50 years. Although we posted our call for participation on the platforms freelancer.com and upwork.com, we have not received any offers from users on those platforms. Therefore, all participants found out about our study either based on talks at industry conferences, social media channel postings (5 participants), or direct contacts who forwarded the recruitment flyer to their web development teams (8 participants). Our survey asked the age in intervals of ten, e.g., 30-40, to not be too privacy-invasive. Appendix B shows an overview of the demographics of all interview participants.

In total, the survey received 50 submissions, of which 28 were eligible and invited to the study. 13 answered our invitation emails or the reminders we sent after one week and later participated in the interview.

Three participants are working in an IT security field, e.g., an Incident Response Team, while one of them also considered themselves as casual Web developer because they created small Web applications, e.g., for personal use. Eleven are working as Web developers, where one of them also mentioned IT security as one of his work areas. Also, one of the Web developers is at the same time working in DevOps, which was also mentioned by one other participant.

Regarding security education, seven participants mentioned that they took security-related courses during their studies. The other six participants had taken courses or certificates for IT security but reported to have self-taught knowledge about the topic via videos and blog posts. Thus, we reached our target to conduct this study with a homogeneous set of participants who are security aware and familiar with Web development. We also had two participants who admitted that they were not educated in the area of Web security but only read a few articles about XSS.

During the chat about Trusted Types, eight participants stated that they had never heard about the mechanism before, while the others did not explicitly mention knowing (or not knowing) Trusted Types before participating in our study. We also asked the participants about their knowledge of XSS mitigation techniques. Here, seven participants mentioned CSP, ten mentioned proper sanitization and/or filtering of user-controlled input, and one considered Web Application Firewalls (WAF) a mitigation for XSS. Notably, many of the participants mentioned multiple mitigation techniques here, so our participants proved to be knowledgeable regarding the mitigation of XSS attacks.

4.2 Strategies & Roadblocks for Trusted Types

Based on the approaches that we had observed throughout the coding tasks, we created a workflow diagram (shown in Figure 1) that we use in this section to shed light on the different roadblocks that we investigated in the different deployment stages. We also explain in detail the changes in the Web application and the reasons for those changes. In addition to that, we discuss the implications of the decisions and strategies on the success of the deployment process. We mitigate bias by carefully introducing participants to the task, starting by explicitly telling them to act as if they were doing this in their company or for their own projects and that they can and should use any resources and libraries (see Appendix A).

Stage 1: Enforcement and Initial Deployment

Participants P3 & P4 deployed CSP very late, after writing at least one sanitizer such that they were overwhelmed by the number of errors that were caused. Also, participants P5 & P6 did not manage to deploy a CSP. To get deeper insights into the deployment process, we tried to ease this part of the process. Thus, we added commented-out boilerplate code to set a CSP header that enabled Trusted Types for scripts and enforced a *default* policy. Also, the participants were confused about using named policies, because one of the information sources recommended their usage.

> *"Is this the default policy? Because they had something written here, you should not always take the default, but tinker with the specific one."*
> – Participant 12

However, this is impossible for third-party code that is unaware of Trusted Types deployment since that would have to explicitly invoke the named sanitizer (rather than implicitly invoke the default one). Therefore, we added boilerplate code for a *default* Trusted Types policy, which simply returns the input without modifications for all sanitizers after P5.

With the new changes, participant P7 managed to deploy the CSP that enforces the default policy with a sanitizer, which simply returns its input and therefore does not add security. However, the enforcement of this default policy resulted in errors caused by Twitter's usage of source-less same-origin iframes for their timeline widget. Those kinds of iframes inherit the CSP policy that mandates the usage of Trusted Types, but they do not inherit the sanitizer functions. Requiring the usage of those functions without them being present then results in multiple errors. Notably, Twitter's behavior

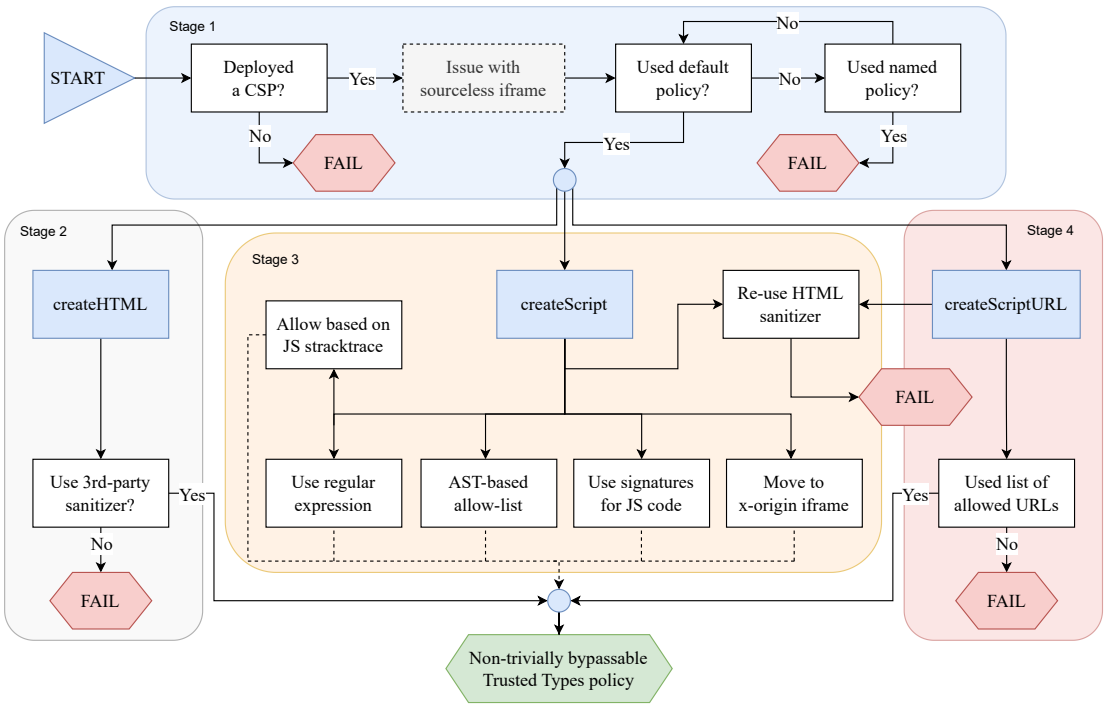

Figure 1: Diagram of the deployment workflow based on our participants.

regarding how the widget is built changed throughout the process of this study. To our question in the Twitter Community Hub [63], they acknowledged that this change is more of a usability/accessibility-driven change because the server-side rendering of the timeline is reducing the loading time of the widget rather than easing Trusted Types deployment.

Stage 2: HTML Sanitizer

All participants started with implementing the *createHTML* sanitizer. Participants P4, P7, P8, P11, and P12 used a third-party HTML sanitizer (DOMPurify [13]) to implement the *createHTML* function. Because none of the third parties that we used required the usage of adding JavaScript in any form into the DOM, this path led to a successful deployment of the *createHTML* sanitizer.

> *"There was also a library recommended here. If OWASP recommends DOMPurify, then I'll take it!"*
>
> – Participant 11

The other participants (P3, P5, P6, P9, P10) implemented the sanitizers themselves, which in all cases led to an easily bypassable filter. Most participants started by copy-pasting the example sanitizer from one of the information sources. This sanitizer, however, replaces all < with their HTML equivalent (`<`); thus, all inserted tags are destroyed, which causes Twitter and Facebook not to work anymore. Some of them (P6, P10) then changed this code to only remove `<script`. This is utterly insecure since an attacker can still inject an inline event handler (`<img src=foo onerror=alert(1)>`). Thus, in total, all custom sanitizers were trivially bypassable, so only seven out of 13 participants managed to implement a secure HTML sanitizer function for Trusted Types.

Stage 3: JS Sanitizer

Five of our participants (P4, P7 & P10-12) first reused their `createHTML` sanitizer to also sanitize JavaScript. Both the participant's custom HTML sanitizers as well as DOMPurify replace an opening HTML tag (<) with its encoded equivalent (`<`) or the empty string. The evaluated code from the custom advertisement widget contains the following snippet:

```
for (let i = 1; i <= 5; i++) { ... }
```

With the HTML sanitizer function, this code was therefore changed to, for example:

```
for (let i = 1; i &lt;= 5; i++) { ... }
```

Thus, the change causes a syntax error during the `eval` call. The participants that took this path needed quite a bit to notice this, as Chromium only pointed to the call of the `eval` for the syntax error. To find the root cause of the error, the developers were required to investigate how this functionality works in detail, determine the source of the evaluated source code, and compare it with the result of the sanitizer.

Those participants who did not re-use their `createHTML` and those who afterward changed their sanitizer due to the errors mentioned above mentioned diverse ways how to implement the `createScript` sanitizer. Participants 6, 8 & 13

mentioned that Regular Expressions could be used to differentiate between benign and attacker-controlled code. The possibility to base the `createScript` sanitizer on cryptographic signatures to only evaluate benign code was mentioned by participant 11. Another solution mentioned by two of our participants (P7 & P11), tried to circumvent the problem by moving the code that required the usage of eval into cross-origin iframes such that their execution context differs and Trusted Types are not inherited into the iframe. Participant 13 also mentioned the possibility of raising and catching a custom JavaScript Exception in order to use the stack trace of the JS execution to allow specific code snippets to run, while participant 10 thought about an allowlist-based on the Abstract Syntax Tree (AST) of the code snippet. We discuss the (in)feasibility of those sanitization approaches in Section 5.3.

Stage 4: URL Sanitizer

Also, here five participants re-used the HTML sanitizer, but because the URLs in our application did not contain any HTML characters, like $<$, they did not recognize that this might be an issue. The five participants that noticed or directly created a sanitizer themselves used an allowlist-based approach. Throughout that process, some participants noticed that the dynamically added URLs contained a changing string.

"It is a hash here. I could well imagine that it changes with every refresh or so."
– Participant 8

In those cases, the participants changed their implementation to an allowlist based on hostnames rather than URLs or used the `startsWith` function to check for the URL without considering the seemingly random string. As ignoring the URL path might lead to JSONP-based script executions [66] or open redirects [49] being possible, the `startsWith` or host-based allowlist is still trivially bypassable.

Notably, we also had three participants who gave up or ran into time issues during the implementation of the createScript sanitizer and did not implement the createScriptURL sanitizer.

Using AI for a solution:

During the introduction of the coding task, we mentioned that the participants could freely choose their information sources and that they could use anything during the coding task that they considered helpful. One participant used the AI-based GitHub Copilot to write the Trusted Types sanitizer functions.

"I could actually be quite bold here once and say: '// Generate Trusted Types using DOMPurify"'
– Participant 12

The generated code for the `createHTML` sanitizer was correct because it simply used DOMPurify (without actually importing it, though). However, the generated code for the other two sanitizers also used DOMPurify, which not only resulted in trivially bypassable sanitizers as the library is not designed to sanitize JavaScript code or URLs, but it also caused the ad feature of the application to not work anymore due to the reasons explained in Section 4.2. See Listing 2 for the entire output of the GitHub Copilot.

4.3 Client- vs. Server-side XSS

While eleven participants properly explained the concept of server-side XSS, only eight of those managed to describe client-side XSS and the difference between them. Notably, the participants who did not know the difference between the different XSS types failed to create a proper sanitizer as they, for example, tried to write a customized sanitization function.

"In the real world, if I wanted to protect against XSS attacks, I wouldn't just use JavaScript, I would use PHP, which has much more functions!"
– Participant 6

Those custom HTML sanitizers also ignored the fact that JS cannot only be executed by script tags but also via event handlers or URLs, as they only removed script tags but nothing else. Therefore, JavaScript URLs (URLs using the `javascript:` schema), or `onload`/`onerror` JavaScript handlers of, for example, HTML image tags, can be used to execute malicious code. Also, those participants spent quite some time creating those trivially bypassable filters and, therefore, did not manage to work on sanitizing JavaScript code or script URLs.

4.4 Perceptions on XSS and Trusted Types

From the seven participants that used third-party sanitization libraries to implement the `createHTML` sanitizer, four explained that they are doing this because self-created sanitization functions or custom security solutions, in general, are prone to be bypassable.

"Looks like a lot of manual work and gives the user many possibilities to do things wrong with this API"
– Participant 10

While this perception of sanitization seemingly pushed the participants into using proper sanitization libraries instead of creating a sanitizer themself, others might cause them to incautiously write dangerous code, as five participants mentioned that if you use common frameworks, XSS attacks cannot happen anymore or are at least mitigated by default.

"I don't know how all these frameworks, like React or something do that, that they don't have XSS."
– Participant 4

However, although React automatically escapes string variables in views if added to the DOM, there are still XSS vectors that are not secured per default. Despite the trivial exploitability if functions like `dangerouslySetInnerHTML` are present, it is also possible to execute JavaScript by setting `href` or `src` attributes of HTML tags to `javascript:` or `data:` URIs [46]. Even if frameworks would secure all first-party code,

the common practice in the Web to include third-party code, which might contain client-side XSS vulnerabilities, can still cause harm to the users of the Web application.

4.5 Usability Perspective

In total ten of our participants sent us the filled-out SUS survey. The highest SUS score was 80, the lowest 32.5, and the average was 58.75. The average score indicates that people rate Trusted Types as only marginally acceptable [2].

During the interviews, all participants mentioned that they see XSS as a big problem in modern Web applications. We also asked the participants before and after the coding task if they would use Trusted Types for one of their Web applications. Ten of the 13 participants rated TT more negatively after having completed the coding task. For five of those, the engineering effort of the deployment was the main concern. They especially mentioned that the deployment of Trusted Types into an existing application is hard due to existing code that needs to be restructured or third parties that need to be replaced. They claim that this would be considerably easier if security was considered from the planning phase of a project. Three participants explicitly mentioned this was a reason to only consider Trusted Types for private projects and that they would not use it in their company. Nearly all participants also explicitly mentioned the massive time and engineering effort in properly filtering or sanitizing JavaScript via the createScript sanitizer. This indicates that developers want to defend against XSS attacks; however, they think it requires time and engineering effort.

Key roadblocks for Trusted Types

Misconceptions about third-party sanitizers are causing disfunctional or bypassable sanitizers, as, e.g., HTML sanitizers are used for JS or URL sanitization.

As depicted in Stage 1, **Information Sources** provide misleading/superficial information, which is why, e.g., the choice between a **Named or Default policy** caused troubles during the deployment as only default policies work in real-world settings with third-parties.

Stage 2 of the deployment process reveals that **JavaScript sanitization** is hard up to impossible and requires heavy engineering effort.

5 Discussion

This section explains in detail some of the problems that our participants faced and discusses ways how those issues can be fixed or mitigated. Additionally, we discuss improvement suggestions that our participants mentioned after the coding task. Also, we discuss the limitations of this work at the end of this section.

5.1 Sourceless iframe Problem

CSP inheritance rules dictate that when a sourceless iframe (i.e., one without a `src` property) is encountered by the browser, the parent's CSP is enforced. This is necessary since, otherwise, an adversary could trivially bypass CSP's protection. In this scenario, assume an attacker capable of injecting markup into a page. They could then inject an iframe tag with a `srcdoc` attribute, which allows to specify the HTML (and thus JavaScript) rendered in the iframe. If now the parent's CSP would *not* be inherited, the attacker's code could run unimpaired in the origin of the parent's execution context.

For the first seven interviews, Twitter's timeline widget was loaded via a script that created a sourceless iframe, which then contains the requested timeline. This comes with a specific issue, which was identified and explicitly noted by one of the participants. Since the sourceless iframe is a different JavaScript context, none of the defined functions of the parent's document are available in the iframe. Since the actual sanitizers are implemented in JavaScript, this also means they are *not* present in the iframe. As a result, the iframe enforces the usage of a Trusted Types policy for which no default sanitizer exists, which, therefore, will result in errors and malfunction of the Twitter timeline widget.

One way for developers to fix this issue themself is to hook calls to all of the functions that can be used to append elements to the DOM (e.g., `insertBefore` or `appendChild`) and in case of an iframe being added directly inject the Trusted Types sanitizer into this frame (see Listing 3). We cannot hook the element creation of the iframe here as we can only script into iframes that are already added to the DOM. If now, however, code in these iframes again creates sourceless iframes, the problem will reoccur such that the JavaScript snippets that inject the sanitizers into added iframes need to be aware of their source code and inject themselves into the iframe such that all iframes are covered.

The above-mentioned solution can be solved through proper boilerplate code, yet developers should not have to deal with this issue in the first place. Instead, we propose that the standard be adjusted such that all registered sanitizers are passed on to sourceless iframes automatically. This way, the code in the iframe can make use of explicit sanitization through the named sanitizers, but more importantly, any code that is agnostic to Trusted Types still works as the default sanitizer is called implicitly. We note that if the iframe requires the usage of a *separate* sanitizer, it suffices for the main page's CSP to specify an allowed name for a sanitizer yet not register it. This way, the iframe's code can register and use the sanitizer with the previously unused name.

5.2 Standardized & Customizable Sanitization

Third-party sanitization libraries such as DOMPurify were used by seven of our participants. Notably, six of them then used this sanitizer for HTML content, also for JavaScript code and URLs, but three of them later noticed that this could not work. Thus, the initial misconception of DOMPurify as a general-purpose sanitizer was problematic. One participant also noted that leveraging a third-party library can be beneficial, yet requires this third-party sanitizer to be updated constantly in the face of potentially new variants of XSS.

"XSS attacks, they are always evolving. I'm not sure if Trusted Types can also keep up."
– Participant 6

Notably, the planned Web Sanitizer API [5] would fix those issues or at least shift the responsibility from the operators to the browser vendors' side. However, the Web Sanitizer API can only be used for `createHTML`, but not for `createScript`, as it can also only sanitize HTML content. Also, at the time of writing, the Web Sanitizer API is all-or-nothing – it can only remove all dangerous constructs but has no way to exempt certain scripts required by the page to function properly. While it might be feasible for the first party to update their code to not dynamically add script tags and hence be compliant with usage of the Web Sanitizer API, functionality of third parties depends on the coding practice of parties beyond the control of the site's developer.

5.3 Sanitizing JavaScript

As mentioned earlier, but also pointed out by our participants, the `createScript` sanitizer is the hardest part as it can neither be solved properly by an allowlist nor are there existing sanitization libraries or APIs for that task.

"createScript is another beast, because it checks the created JavaScript strings. Of course, it is difficult to ensure that this is valid at all, whether it is secure or not."
– Participant 8

Nevertheless, our participants mentioned or even implemented some ways how we might be able to sanitize or distinguish attacker code from benign code (see Section 4.2). During the implementation of the `createScript` sanitizer, participant 10 thought about an allowlist based on the Abstract Syntax Tree (AST). However, an attacker could then still create malicious JavaScript with the same AST [17] as the benign and allowed snippets and thus bypass the sanitizer.

Participant 11 mentioned that one way to allow specific scripts to be executed is signatures to ensure that the origin of that script is trusted. However, while this works for external resources under the control of the operator, it will only work for third-party scripts if the developer convinces this party to change their functionality to also sign the code that they deliver to the application.

Participants 6, 8 & 13 suggested Regular Expressions to only allow code with a certain pattern. Besides this only being maintainable at a small scale, it might be bypassable as attackers may be able to write their attacker code in the benign and allowed pattern.

"We can not insert a general-purpose sanitizer here now. Maybe Regex, but how efficient is that, then? And if-else stories about plain text comparisons of any script content is also kind of difficult."
– Participant 8

Notably, participant 13 mentioned the possibility to raise and catch exceptions in order to create an allowlist based on the caller of the function. This was also our sample solution (see Listing 4) for `createScript`. Here, we used the call stack to allow JavaScript execution from specific functions or parties via the stacktrace. While this might yield a functional solution, it shifts the trust to the third-party developer: if their JavaScript code has an exploitable XSS flaw, an adversary can simply exploit that to attack the including site.

Participants 7 & 11 mentioned that instead of somehow implementing the sanitizer, they would rather move the content that requires the usage of string-to-code conversion into an iframe that runs under a different origin. While this works for the third parties that we used, others will not work properly anymore as some advertisements require access to the Web page (including its DOM) [15]. Also, this solution is more of a compartmentalization than an actual sanitization, hence somewhat beside the point of Trusted Types.

5.4 Impact of Information Sources

The information sources at the time of writing focused on named policies rather than default policies. One of the information sources [31] suggested to "[u]se the default policy sparingly, and prefer refactoring the application to use regular policies instead". For four of our participants (P5, P5, P8 & P9), this suggestion caused problems as they first tried to use named policies and then later on noticed that they needed to implement a default policy anyway because they could not change the JS code provided by third parties.

"I found it difficult in the explanations how to handle own policies or how to set default policies."
– Participant 8

While the usage of named policies enables the developer to create sanitizer functions that are customized towards the data that flows into the respective sink, creating the policies and refactoring the code is work that should be done after having the default policy as a fallback, such that the developer is not overwhelmed by the number of errors. Thus, we argue that information sources should advertise the usage of a default policy as a first step and then the usage of named policies for certain parts of the application to harden the protection offered by Trusted Types.

"I could imagine, for example, would be to somehow outsource that into an iframe or something. So, to sandbox that, if I already have such a case. But that would be a completely different topic."
– Participant 11

Seven participants also complained about insufficient examples, especially for the usage of a default policy and the `createScript` and `createScriptURL` sanitizer functions. The examples on the information sources only contain explicit examples where strings are sanitized but only superficial examples of the sanitizers.

"In the examples here, there are just the strings that can be escaped and then appended, but I don't have any strings here to work with"
– Participant 3

The missing examples for the non-HTML sanitizer might be one of the reasons why seven participants initially used the examples for the `createHTML` sanitizer for all three sanitizers. Using the HTML sanitizer for JavaScript and URLs is not only insecure as it does not really sanitize anything for those strings but also leads to a loss of functionality depending on the third-party JavaScript code, as we have pointed out in Section 4.2. To ease the debugging process for code that is evaluated through string-to-code conversation, browser vendors might consider changing their error output for those functions to point to the evaluated string.

In general, our participants (P3, P5) mentioned that the information sources for Trusted Types only provide superficial information on the use case and the deployment process.

"Just with the half-hour now, I understood it much better than just in the article, and it makes complete sense to me now why I would use those."
– Participant 5

To improve the situation, we used our results to create an information source [11] with a roadmap for successful Trusted Types deployment with explanations and code examples for each of the individual deployment steps. We also plan to incorporate edge case examples, like the hook for dynamic iframe additions or an HTML parser that preserves specific event handlers.

5.5 Implications for the Trusted Types Design

Based on the roadblocks that we have seen and our participants' improvement suggestions and feedback, we elaborate on different implications for the design of Trusted Types in order to improve the mechanism.

One root cause of the roadblocks was that the information sources only provided superficial information about Trusted Types, such as only explaining the CSP header but not providing information about the sanitizers. But even in those cases where the information source tried to explain the sanitizers, participants noted insufficient or misleading examples. The focus on named policies, with the reason of a more explicit way of securing a Web application, misled participants to use only named policies, but not default ones, which resulted in the failure of third-party code. Thus, we recommended that Trusted Types need to have high-quality information sources with guidelines, examples, and detailed explanations.

Also, the rather technical problems of same-origin iframes or complicated workarounds to get more contextual information about the script execution can be at least eased by automating those workarounds. Given that the policies are registered using a centralized API call, it should be easy to automatically inherit all policies to same-origin iframes, such that those are not causing any issues anymore. In addition to that, the sanitizers could get an additional argument that gives contextual information, such as the call stack, that developers can then use to create an allowlist.

To further ease the creation of sanitizers, pre-configured sanitizing APIs, such as the planned Web Sanitizer API [5], should be provided. Notably, those APIs should provide the capability to allow specific inline scripts and/or events inside the injected HTML in order to enable the sanitization of third-party code. This could be done by either using code hashes or context information from the developer or, even better, the createHTML sanitizer could invoke the createScript sanitizer for inline script tags and inline events to not mix up the different steps, as suggested by participant 13.

Similar to CSP [51, 58], third-party scripts are also a big problem for the deployment of Trusted Types. It is hard to find a technical solution for this problem, which is why we argue that a political solution, e.g., by the W3C has to be elaborated to encourage third parties to be compliant with security mechanisms. One idea would be to have a warning icon similar to those for unencrypted connections if the Web site is not deploying properly configured security mechanisms. This way, companies would choose their partner services with care, and those that do not adhere to compliant coding styles will lose customers and, thus, revenue. Of course, this idea first should be tested and scientifically evaluated by us as a research community, e.g., by creating such a system in a study that follows a participatory design approach.

Key problems for each sanitizer

createHTML: As the example of DOMPurify and the Sanitizer API show, current libraries make it hard up to impossible to preserve JS code in HTML as they remove *all* existing JS code from the HTML.

createScript: There is (currently) no proper way of sanitizing JS code, as Stage 2 has shown that all sanitization ideas from the participants are bypassable. The only secure but hard-to-maintain solution would be using code hashes, which are hard to use here due to WebCrypto not being available in synchronous contexts.

createScriptURL: Randomness in URLs causes the use of startsWith (or RegEx), as shown by our participants. This, however, may cause bypasses through JSONP or Open Redirects.

5.6 Recommendations

This section provides a clear list of recommendations for all parties involved in the development or deployment of TT.

5.6.1 For Browser Vendors

(1) Ease the implementation of the HTML sanitizer, e.g., by introducing an allow list of not removed scripts or by calling the other sanitizers as soon as script tags or inline JS handlers are faced. **(2)** To get rid of the sourceless iframe problem, browsers could think about inheriting all registered TT sanitizers if a CSP containing the trusted-types keyword is inherited, which would also fix other security issues regarding TT and iframes [64].

5.6.2 For Developers

(1) Think about TT integration from the beginning of a project, as certain coding practices (e.g., sourceless iframes) make it hard to deploy TT. **(2)** Although named policies are more descriptive, developers should use default policies for existing projects such that they do not need to change the entire code base, but the sanitizer is automatically called.**(3)** As in the case of other CSP features [51], developers should, if possible, choose third parties with care as they might have behavior that makes it hard to deploy TT (e.g., injecting content with inline JS handlers).

5.6.3 For the Community

(1) As information sources and descriptions of sanitizers are causing misconceptions that lead to dysfunctional sanitizers, the security community needs to work on better materials to adhere to real-world scenarios. **(2)** The sanitization of JS code seems to be one of the biggest roadblocks for deploying TT. Here, we need to work on better tools and evaluate them to allow developers to easily distinguish between benign and malicious code.

5.7 Limitations

In addition to the expected limitations of self-reported data in interview studies (e.g., recall bias, social desirability bias), we acknowledge observer bias during the coding task. To mitigate this, we emphasized that we are not testing participants' knowledge or ability to deploy Trusted Types but are relying on their help to evaluate the header. Further, restricting the coding task to 90 minutes may limit how participants engage with Trusted Types. However, we argue that if a security mechanism needs massive engineering effort even for a small Web application, the deployment for complex Web applications in the wild will be challenging due to monetary and time constraints. Also, we acknowledge that supervised coding tasks in lab settings lead to lower ecological validity compared to field studies. In our analysis, the first author, who conducted the interviews, built the codebook iteratively after discussing interesting concepts from interviews with the other authors. Given that the first author might have missed concepts that arose during the interview that a second initial coder might have caught, the set of concepts described in this paper might not be exhaustive. However, with the final codebook, the second and third author codings were used to check for inter-coder reliability, and based on the final coding and codebook, all coders discussed possible themes such as deployment strategies and roadblocks. Last, we do not claim generalizability to the overall population of Web developers. We reached saturation with respect to high-level roadblocks participants encountered for the given Web application. However, we do not claim completeness with regard to low-level misconceptions that may be influenced by educational level or cultural background. Seven participants have an IT Security background, suggesting our sample is more educated than the general Web developer population. Thus, we position our results as a lower bound for problems that occur during the deployment of Trusted Types.

6 Conclusion

Our work presents the first qualitative study involving 13 real-world Web developers to evaluate the usability of Trusted Types deployment. With our qualitative interview study, and especially the coding task, we were able to uncover important roadblocks to Trusted Types deployment and the strategies that lead to successful sanitizers. Based on those, we elaborate on improvement suggestions for the mechanism so that we can ease the deployment of Trusted Types for developers. As the quality of the available information sources was one of the major problems that developers faced or mentioned during the study, we as a community need to work on better-quality information sources that need to contain examples and detailed explanations. In addition to that, technical issues like the inheritance of Trusted Types to same-origin iframes need to be fixed to ease the deployment of the mechanism.

Acknowledgments

We want to thank the reviewers for their feedback regarding our paper's presentation and all our participants for their time and insights that ultimately made this work possible.

References

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the usability of cryptographic apis. In *S&P*, 2017.

[2] A. Bangor, P. T. Kortum, and J. T. Miller. An empirical

evaluation of the system usability scale. *International Journal of Human–Computer Interaction*, 2008.

[3] A. Barth. RFC 6454: The Web Origin Concept. *Online at ietf.org*, 2011.

[4] F. Braun. Mozilla Standards Positions. Issue No. 20 (Trusted Types). *GitHub.com. Online at github.com*, 2023.

[5] F. Braun, M. Heiderich, and D. Vogelheim. HTML Sanitizer API. *W3C Draft. Online at wicg.github.io*, 2022.

[6] V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 2006.

[7] J. Brooke et al. SUS: A 'Quick and Dirty' Usability Scale. *Usability Evaluation in Industry*, 1996.

[8] A. G. B.V. Amberscript - Data Security & Privacy. *Online at amberscript.com*, 2022.

[9] S. Calzavara, A. Rabitti, and M. Bugliesi. Content Security Problems?: Evaluating the effectiveness of Content Security Policy in the Wild. In *CCS*, 2016.

[10] Can I use. Can I use trustedTypes API? Online at caniuse.com, 2022.

[11] CISPA - GIT. Full Replication Package. *Online at github.com*, 2023.

[12] R. Croft, Y. Xie, M. Zahedi, M. A. Babar, and C. Treude. An empirical study of developers' discussions about security challenges of different programming languages. *Empirical Software Engineering*, 2022.

[13] Cure53. DOMPurify. *Online at github.com*, 2022.

[14] Django Software Foundation. Security in Django. Online at djangoproject.com, 2017.

[15] X. Dong, M. Tran, Z. Liang, and X. Jiang. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In *ACSAC*, 2011.

[16] M. W. et al. CfC to publish as an FPWD. Issue No. 342. *GitHub.com. Online at github.com*, 2018.

[17] A. Fass, M. Backes, and B. Stock. Hidenoseek: Camouflaging malicious javascript in benign asts. In *CCS*, 2019.

[18] Google. Puppeteer. *Online at github.com*, 2022.

[19] B. Hayak. Same Origin Method Execution (SOME). *Online at benhayak.com*, 2015.

[20] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In *CCS*, 2012.

[21] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang. mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations. In *CCS*, 2013.

[22] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun. Home is safer than the cloud! privacy concerns for consumer cloud storage. In *SOUPS*, 2011.

[23] M. Jakobsson, Z. Ramzan, and S. Stamm. JavaScript Breaks Free. *Online at psu.edu*, 2007.

[24] R. B. Johnson and L. Christensen. *Educational research: Quantitative, qualitative, and mixed approaches*. Sage publications, 2019.

[25] R. Kang, L. Dabbish, N. Fruchter, and S. Kiesler. "my data just goes everywhere:" user mental models of the internet and implications for privacy and security. In *SOUPS*, 2015.

[26] Z. Kang, S. Li, and Y. Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *NDSS*, 2022.

[27] C. Kern. Preventing Security Bugs through Software Design. OWASP AppSec California, 2016.

[28] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *SAC*, 2006.

[29] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. *Online at webappsec.org*, 2005.

[30] D. Klein, T. Barber, S. Bensalim, B. Stock, and M. Johns. Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions. In *EuroS&P*, 2022.

[31] K. Kotowicz. Prevent DOM-based cross-site scripting vulnerabilities with Trusted Types. *Online at web.dev*, 2020.

[32] K. Kotowicz. Trusted Types - mid 2021 report. *Online at research.google*, 2021.

[33] K. Kotowicz and M. West. Trusted Types. *W3C Standard. Online at w3c.github.io*, 2021.

[34] K. Krippendorff. *Content analysis: An introduction to its methodology*. Sage, London, 2004.

[35] K. Krombholz, W. Mayer, M. Schmiedecker, and E. Weippl. "I Have No Idea What I'm Doing"-On the Usability of Deploying HTTPS. In *USENIX Security*, 2017.

[36] K. Krombholz, K. Busse, K. Pfeffer, M. Smith, and E. von Zezschwitz. "If HTTPS Were Secure, I Wouldn't Need 2FA"-End User and Administrator Mental Models of HTTPS. In *S&P*, 2019.

[37] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE*, 2006.

[38] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *NDSS*, 2019.

[39] S. Lekies, B. Stock, and M. Johns. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In *CCS*, 2013.

[40] S. Lekies, K. Kotowicz, S. Groß, E. A. Vela Nava, and M. Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *CCS*, 2017.

[41] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia. Riding out DOMsday: Toward Detecting and Preventing DOM Cross-Site Scripting. In *NDSS*, 2018.

[42] K. Mindermann, P. Keck, and S. Wagner. How usable are rust cryptography apis? In *QRS*, 2018.

[43] M. Musch, M. Steffens, S. Roth, B. Stock, and M. Johns. ScriptProtect: Mitigating unsafe third-party javascript practices. In *AsiaCCS*, 07 2019.

[44] OWASP. OWASP Top 10 Web Application Security Risks. Online at owasp.org, 2017.

[45] N. Patnaik, J. Hallett, and A. Rashid. Usability smells: An analysis of developers' struggle with crypto libraries. In *SOUPS*, 2019.

[46] R. Perris. Avoiding XSS in React is Still Hard. *Online at medium.com*, 2018.

[47] P. Ringnalda. Getting around IE's MIME type mangling. philringnalda.com, 2004.

[48] D. Ross. Happy 10th Birthday Cross-Site Scripting. *Online at microsoft.com*, 2009.

[49] S. Roth, M. Backes, and B. Stock. Assessing the Impact of Script Gadgets on CSP at scale. In *AsiaCCS*, 2020.

[50] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock. Complex Decurity Policy? A longitudinal analysis of deployed Content Security Policies. In *NDSS*, 2020.

[51] S. Roth, L. Gröber, M. Backes, K. Krombholz, and B. Stock. 12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP. In *CCS*, 2021.

[52] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS*, 2010.

[53] Sebastian Roth. Recruitment Flyer. *Online at twitter.com*, 2022.

[54] J. Smith, L. N. Q. Do, and E. Murphy-Hill. Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In *SOUPS*, 2020.

[55] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *NDSS*, 2013.

[56] M. Steffens and B. Stock. PMForce: Systematically Analyzing postMessage Handlers at Scale. In *CCS*, 2020.

[57] M. Steffens, C. Rossow, M. Johns, and B. Stock. Don't Trust The Locals: Investigating the Prevalence of Persistent Client-Side Cross-Site Scripting in the Wild. In *NDSS*, 2019.

[58] M. Steffens, M. Musch, M. Johns, and B. Stock. Who's Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI. In *NDSS*, 2021.

[59] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns. From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting. In *CCS*, 2015.

[60] A. Strauss and J. M. Corbin. *Grounded theory in practice*. Sage, London, 1997.

[61] C. Tiefenau, E. von Zezschwitz, M. Häring, K. Krombholz, and M. Smith. A usability evaluation of let's encrypt and certbot: usable security done right. In *CCS*, 2019.

[62] Tranco Website. Tranco List from 26 June 2022. *Online at tranco-list.eu*, 2022.

[63] Twitter Community. Reasons to move from same-origin iframes to third-party iframes? Online @ devcommunity.x.com, 2022.

[64] L. Veronese, B. Farinier, P. Bernardo, M. Tempesta, M. Squarcina, and M. Maffei. Webspec: Towards machine-checked analysis of browser security mechanisms. In *S&P*, 2023.

[65] P. Wang, B. A. Gudmundsson, and K. Kotowicz. Adopting Trusted Types in Production Web Frameworks to Prevent DOM-Based Cross-Site Scripting: A Case Study. In *EuroS&PW*, 2021.

[66] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP is dead, long live CSP! On the insecurity of whitelists and the future of content security policy. In *CCS*, 2016.

[67] M. Weissbacher, T. Lauinger, and W. Robertson. Why is CSP failing? Trends and challenges in CSP adoption. In *RAID*, 2014.

[68] M. West. CSP Level 3. *W3C Standard. Online at w3.org*, 2021.

[69] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song. Learning and programming challenges of rust: A mixed-methods study. In *ICSE*, 2022.

A Interview Guideline

Introduction:

Hi $Name,
I'm Sebastian, first of all thank you for participating in our Interview such that we can improve the security and usability of Trusted Types. Also, please note that we want to find general strategies and roadblocks of Trusted Types, so we are not evaluating the correctness of your answers we only want to improve the security mechanism. We first start with few questions regarding your working environment and some Web topics. Those will take around 15 minutes, and if you don't want to answer a question, feel free to say it, and we'll skip it. Afterwards, we will do the coding task, where you are asked to implement Trusted Types sanitizers for a small Web application. We can offer you the App as a Docker, you can run the code directly on your machine, or you can remote control one of our machines. Which would be the option you prefer? Any other questions before we start?

General Questions:

- In your company, what is the specific area that you cover with your work? If you are in a team, what is your specific task in this team?
- Are you considering yourself a Web developer? If yes, since: ...
- Do you have an IT-Security background? If yes, please specify: ...
- Was Web Security part of your education? If yes, briefly outline the basic content / topics covered?
- How familiar are you with XSS? How did you learn about it?

Web Security Questions:

- How would you explain Cross-Side Scripting?
  - How does the vulnerability happen?
  - Client- vs. Server-side XSS?
  - Client-side execution sinks?
- What kind of Web security mechanisms come into your mind when you want to defend against XSS?
- In preparation for this study, you were asked to inform yourself about Trusted Types. Did you already know about this mechanism?
  - How have you found/searched for information?
  - Which Information Sources did you find?
  - How would you rate the quality of those sources?
- After reading the information sources: How would you explain Trusted Types?
  - How is it enforced?
  - How does it defend against XSS?
- What is you impression of TT at this point? Any concerns?
- Would you use Trusted Types for one of your Web applications?

Coding Task / Knowledge Validation:

Your task is to create Trusted Types sanitizers that defend the given Web application against client-side XSS. The application itself is written in python using the Django Framework. However, you do not need to know any details about Django as you only need to write the sanitizer in JavaScript. During the coding task, it would be awesome if you could think aloud about your decisions such that we can learn from them. Also, it would be nice if you could share your screen such that we can see the roadblocks that you face during deployment. Notably, you can develop just like you wish, so feel free to use any online resources, textbooks, and so on. So let's have a look into the source code...
As said, we are using the Python Django Framework. In `app/templates/*` are the HTML files that are used as templates for rendering the content. In `app/views.py` you can find the logic / implementation behind every endpoint. In `app/static/*` all static files such as JavaScripts that are used by the application are located. In `app/static/js/trusted-types.js` we already have boilerplate code for the sanitizers. Explain the TT code and ask "While I'm preparing the application/the docker is building think about this code and if **you would say this code already prevents some XSS attacks?**. Before starting, let's quickly take a look at the Front end.

Debriefing:

- What is you impression of TT after working with it?
- Do you think TTs are a feasible defense against XSS?
- What would have to change for you to use the mechanism? / What would an ideal mechanism look like for you?

Ending: Send the SUS, and the voucher form.

B Participant Demographics

Part.	Age	Gen.	CC	Edu.	Occupation
P1 (*pre*)	20-30	M	DE	-	Student
P2 (*pre*)	20-30	M	DE	-	Student
P3	20-30	M	DE	Master	Security Operator
P4	20-30	M	DE	Bachelor	Research Assistent
P5	20-30	M	DE	Bachelor	Security Engineer
P6	20-30	M	UK	Master	Software Engineer
P7	40-50	M	US	Bachelor	Security Engineer
P8	20-30	M	DE	Bachelor	Research Assistent
P9	20-30	M	PK	Bachelor	Software Engineer
P10	20-30	M	DE	Bachelor	Master Student
P11	30-40	M	DE	Bachelor	Software Developer
P12	20-30	M	DE	Trainee	Software Developer
P13	20-30	M	DE	Bachelor	Freelance Developer
P14	20-30	M	DE	Master	Web-Developer
P15	20-30	F	DE	Master	Web-Developer

Table 3: Overview of the participants demographics.

C Additional Listings and Table

April 2010	Stamm et al. presented their work on a novel XSS mitigation, the Content Security Policy (CSP), at TheWebConf
November 2011	CSP becomes a First Public Working Draft by the WorldWideWeb Consortium (W3C).
since 2014	The research community extensively evaluated CSP and it became clear that for certain types of XSS, especially client-side XSS, it is not working.
15th Sep. 2017	Google engineers proposed Trusted Types as an addition to CSP to defend against client-side XSS.
19th May 2020	First full support for Trusted Types in the Chromium Engine.
06th Sep. 2021	Trusted Types support added to AngularJS by Wang et al. in the process of their IEEE EuroS&P Workshop paper.
17th Jun. 2021	Mozilla called a “Formal Objection” to TrustedTypes becoming a technical specification in the W3C.
06th Jul. 2021	Google published a Report on Google’s success in deploying Trusted Types on 130 of their services.
January 2022	Start with the design of the study followed by pre-tests and pre-studies for this work.
July 2022	The first interview with a non-pre-study participant has been conducted.
April 2023	The final interview has been conducted followed by a coding / evaluation process.
13th Dec. 2023	Mozilla change their standards position to *positive* such that Trusted Types can become standardized.
August 2024	This work is presented and published at the USENIX Security Symposium 2024.

Table 4: Timeline of Trusted Types (incl. this work)

```
// generate trusted types using dompurify
var trustedHTML = DOMPurify.sanitze(dirtyHTML,
↪ {RETURN_TRUSTED_TYPE: true});
var trustedScript = DOMPurify.sanitze(dirtyScript,
↪ {RETURN_TRUSTED_TYPE: true});
var trustedScriptURL = DOMPurify.sanitze(dirtyScriptURL,
↪ {RETURN_TRUSTED_TYPE: true});

// assign trusted types to DOM elements
document.getElementById('trustedHTML') .innerHTML =
↪ trustedHTML;
document.getElementById('trustedScript') .innerHTML =
↪ trustedScript;
document.getElementById('trustedScriptURL') .innerHTML =
↪ trustedScriptURL;

// assign trusted types to attributes
document.getElementById('trustedScriptURL') .src =
↪ trustedScriptURL;
document.getElementById('trustedScriptURL') .href =
↪ trustedScriptURL;
document.getElementById('trustedScriptURL') .onclick =
↪ trustedScriptURL;
```

Listing 2: GitHub Copilot output

```
let TTCode = `(${addTTFunc.toString()})();`;

let ogFunc = Node.prototype['appendChild'];
Node.prototype['appendChild'] = function () {
    let el = ogFunc.apply(this, arguments);
    if (el.tagName && el.tagName.toLowerCase() ===
    ↪ "iframe" && el.contentWindow) {
        let trusted = window.defaultPolicy
        ↪ .createScript(TTCode);
        el.contentWindow.eval(trusted);
    }
    return el;
};
```

Listing 3: Example hook for appendChild

```
let trustedCallers = [/* allowlist */]

let createScript = function (rawJS) {
    let trustedCaller =
    ↪ arguments.callee.caller.toString();
    if (trustedCallers.indexOf(trustedCaller) === -1)
        return null;

    return rawJS;
}
```

Listing 4: Sample solution for the createScript sanitizer