Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Trusted Types integration into open source frameworks and libraries

Masters Thesis

2022
Emanuel Tesař, Bc.

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Trusted Types integration into open source frameworks and libraries

Masters Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	FMFI.KAI - Department of Applied Informatics
Supervisor:	RNDr. Peter Borovanský, PhD.
Consultant:	Krzysztof Kotowicz

Bratislava, 2022
Emanuel Tesař, Bc.

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Emanuel Tesař
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Trusted Types integration into open source frameworks and libraries

Annotation: Trusted Types is a modern Web API which aims to reduce DOM XSS attack surface in web applications.
They give you the tools to write and maintain applications free of DOM XSS vulnerabilities by making
the dangerous web API secure by default. Currently, they are supported in Chrome, Edge and Opera.

Integrating Trusted Types in web applications and libraries requires code changes. The major problem
is when these changes need to be made in third party code which you don't have access to and you
can't easily modify. Trusted Types support in open source projects is gradually improving and our
plan is to analyze these integrations and implement one or more of the challenging ones.

Aim: Main goals of the thesis are the following:
- Review of Trusted Types integrations on various open source projects
- Design and verification of a Trusted Types library integrated into one or more projects
- Open sourcing the integration changes, ideally merging directly into the project sources
- Illustration of the newly created integrations on a real world project

Keywords: Trusted Types, Web APIs

Supervisor:	RNDr. Peter Borovanský, PhD.
Consultant:	Krzysztof Kotowicz
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	06.09.2021

Approved: 13.10.2021 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

Acknowledgments: I want to thank everyone who motivated me to finish my studies and for the support when I was struggling. Thank you K. Kotowicz and P. Borovanský for all the help and friendliness. I thank "MatFyz" because it is just the perfect school for me. Lastly, deepest thanks to my family, and girlfriend Jitka for everything they have done for me. This thesis is devoted to the memory of my Mom, who would be very happy.

Abstrakt

Trusted Types je moderná webová knižnica, ktorá má za cieľ zredukovať riziko DOM XSS zraniteľností vo webových aplikáciách. Integrácia knižnice Trusted Types do webových aplikácií a knižníc vyžaduje zmeny v kóde. Obrovský problém nastáva, keď tieto zmeny musia byť implementované v kóde tretích strán, ku ktorému nemá autor prístup.

Trusted Types podpora vo voľne dostupných knižniciach postupne stúpa a naším cieľom je analyzovať niektoré tieto integrácie a implementovať nové. Najprv v práci popisujeme ako proces integrácie funguje vo všeobecnosti. Potom implementujeme viaceré integrácie rôznej obtiažnosti do knižníc rôznych veľkostí. Naše najväčšie výsledky sú podpora knižnice Trusted Types do knižnice Solid.js a doplnkový nástroj do knižnice Cypress. Ďalej ukazujeme, že pomocou našich integrácií sa dajú vyvíjať, testovať a nasadzovať aplikácie, ktoré využívajú knižnicu Trusted Types. Tieto výsledky demonštrujeme na skutočnej aplikácii menšieho rozsahu.

Kľúčové slová: Trusted Types, Web APIs

Abstract

Trusted Types is a modern Web API that aims to reduce DOM XSS attack surface in web applications. Integrating Trusted Types in web applications and libraries requires code changes. The major problem is when these changes need to be made in third-party code which the developer does not have access to.

Trusted Types support in open source projects is gradually improving and we plan to analyze some of these integrations and implement new ones. We first describe how the integration process works in general. We then implement multiple integrations into different libraries of various complexities. Our biggest achievements are Trusted Types support for Solid.js framework and Cypress testing plugin. We show that it is possible to develop, test, and release applications with Trusted Types enforcement enabled using the integrations we implemented. We demonstrate this on a smaller-sized real-world application.

Keywords: Trusted Types, Web APIs

Contents

Concepts and definitions

These are the common terms used in this paper that are too broad to be explained in specific chapters but should be understood by the reader before reading the paper. Advanced readers may skip this section and start with the first chapter (1).

1. **DOM** – DOM stands for *Document Object Model* and it is the data representation of the objects that comprise the structure and content of a web page.

2. **DOM source and sink** – In the context of XSS, a DOM source is the location from which untrusted data is taken by the application and passed on to the sink, for example, *location* or *cookies*. DOM sinks are the places where untrusted data coming from the sources is getting executed resulting in DOM XSS, for example, *eval* or *Element.innerHTML* [1]. We might also refer to a sink as an *injection sink* because the untrusted value is injected into the sink by the attacker.

3. **SPA** – SPA stands for *Single-Page Applications* and it is a website that interacts with the user by dynamically rewriting the contents of the site instead of loading new pages from the server.

4. **TypeScript** – TypeScript is a strict syntactical superset of JavaScript and adds optional static typing to the language. It is designed for the development of large applications and transpiles to JavaScript.

5. **Hot reload** – The concept of a *hot reload* is that the running application running in development mode is automatically restarted after code changes are made. There is also a related concept called *Hot Module Replacement (HMR)*. The former restarts the whole app, the latter only patches the running application with code changed and preserves the application state.

6. **npm** – *npm* is a package manager for the JavaScript language and JavaScript runtime environment Node.js.

Chapter 1

Introduction

In this chapter we give a brief overview of cross-site scripting (XSS), which is one of the most common web application vulnerabilities. We mainly focus on DOM-based XSS for which Trusted Types are the most effective. We then explain the design of Trusted Types, how it helps to detect, and mitigate these vulnerabilities. Lastly, we describe the motivation and background for our work.

1.1 Cross-site scripting

Cross-site scripting (XSS) is one of the most prevalent vulnerabilities on the web. It is an attack of web applications taking untrusted user input and interpreting it as code without sanitization or escaping. There are multiple categories of XSS. Most experts distinguish at least between non-persistent *(reflected)* and persistent *(stored)* XSS. There is also a third category, DOM-based XSS, which will be explained in more depth as this is a variant that Trusted Types target.

- Stored – A malicious injected script is permanently saved in a server database. The client browser will then ask the server for the requested page and the response from the server will contain the malicious script.
- Reflected – Typically delivered via email or a neutral website. It occurs when a malicious script is reflected off of a web application to the victim's browser [11].
- DOM-based – The vulnerability appears in the DOM (1) by executing a malicious code. In reflected and stored XSS attacks one can see the vulnerability payload in the server response. However, in a DOM-based XSS, the attack payload is executed as a result of modifying the DOM environment in the victim's browser so that the client-side code runs in an unexpected manner.

Cross-site scripting vulnerability became more widespread with the boom of single-page applications (3), where most of the behavior is achieved by modifying the DOM

using JavaScript. There are many functions, element attributes, and properties in the DOM API which interpret the arguments as an executable code. We call these DOM sinks (2). These sinks make it easy for developers to accidentally introduce this vulnerability [29].

```
document.write()
element.innerHTML
element.insertAdjacentHTML
DomParser.parseFromString
frame.srcdoc
eval()
script.src
Worker()
```

Listing 1.1: Common DOM XSS sinks [30] [29]

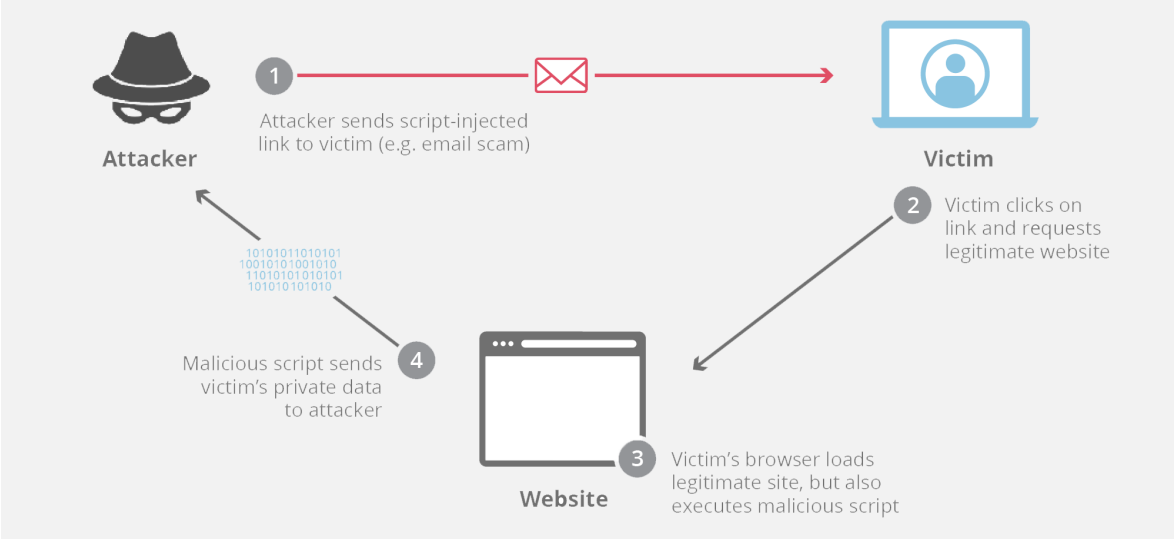

Figure 1.1: Common XSS vulnerability flow [6]

One of the most basic examples of XSS is the interpolation of URL parameters in the DOM. The attacker can prepare a malicious URL which they then send to a victim. The victim executes the payload just by navigating to the site sent by the attacker.

```
<!--
Assume this page is on https://example.com.
It can be misused by the following attack payload:
https://example.com?<img%20src=x%20onerror="alert(1)"></img>
-->
<!DOCTYPE html>
<html lang="en">
  <body>
    <div id="content"></div>
    <script type="text/javascript">
      const content = decodeURIComponent(location.search.substr(1))
      document.getElementById('content').innerHTML = 'URL content: ' +
    content
    </script>
  </body>
</html>
```

Listing 1.2: Basic example of XSS via unsafe URL parameter interpolation

The consequences of XSS vary a lot. Their severity can range from benign annoyances to unrecoverable damages such as full account compromise, and disclosure of users' cookies, storage, secrets, or session. Other attacks may use XSS to change the application content and present fraudulent information to the user.

There are many attempts to reduce the risk of DOM XSS either by dynamic or static checkers. The former usually suffers from performance and scalability issues when applied on large codebases while the latter provides suboptimal results due to JavaScript's dynamic nature [29] [27].

1.2 Trusted Types

Trusted Types is a relatively modern web API designed by Google based on a long history of mitigating XSS [3]. They are currently supported in Chrome, Edge, and Opera.

It is a browser security feature that limits access to dangerous DOM APIs to protect against DOM XSS. Trusted Types provide type guarantees to all frontend code by enforcing security type checks at potentially user-controlled and malicious values. They are delivered through a CSP header and have a report-only mode that does not change the application behavior and an enforcement mode that may cause user observable breakages [15].

When enforced, Trusted Types block dangerous injection sinks from being called with values that have not passed through a Trusted Types policy [15]. If an untrusted value is passed to sink a Trusted Types violation is raised and the DOM is unaffected.

In practice, this means that a potential DOM XSS has been prevented. There are many other resources that can be used to explore Trusted Types in detail [17].

1.2.1 Threat model

Trusted Types is a powerful API. Its main goals are to [20]:

- reduce the risk of client-side vulnerabilities caused by injection sinks.
- replace the insecure by default APIs with safer alternatives which are harder to misuse.
- encourage a design where the code affecting the application security is encapsulated in a small parts of an application.
- reduce the security review surface for applications and libraries.

The main idea behind Trusted Types is to make the dangerous DOM APIs secure by default and encourage application authors to use safer APIs. This model is very effective for preventing DOM-based XSS. However, there are still areas where Trusted Types are not effective enough and other security measures are needed. Some of the non-goals of Trusted Types are [21]:

- preventing or mitigating server side generated markup attacks – Defending against XSS on both client and server can be really complex, especially for applications where parts of the code can run on both client and server, for example, as in Next.js (4.1). To address these attacks, use the existing recommended solutions like templating systems or CSP *script-src* directive.
- controling subresource loading – Trusted Types deal with code running in realm of the current document and do not guard subresources.
- guarding cross-origin JavaScript execution, for example, loading new documents via *data:* URLs – Trusted Types do not guard cross-origin executions at all
- protecting against malicious developers of the web application – It is implicitly assumed that untrusted developer can cause more severe damages. Attempting to guard against malicious developer would lead to more complex and impractical design.

1.2.2 Content Security Policy

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including XSS and data injection attacks [8]. CSP provides a way for browsers to create safer APIs in a backward compatible manner since the API has to be opted in explicitly by the application server by sending the CSP response header or using the CSP inside the HTML meta tag in the response body. If the browser does not support the CSP directive, the directive is ignored and the standard browser behavior applies.

Trusted Types are enabled through a CSP using two different directives:

- *require-trusted-types-for* – This directive instructs user agents to control the data passed to DOM XSS sink functions ([22]).

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

Listing 1.3: Syntax of *require-trusted-types-for directive*

- *trusted-types* – This directive instructs user agents to restrict the creation of Trusted Types policies ([23]). Syntax:

```
Content-Security-Policy: trusted-types;
Content-Security-Policy: trusted-types 'none';
Content-Security-Policy: trusted-types <policyName>;
Content-Security-Policy: trusted-types <policyName> <policyName>
    'allow-duplicates';
```

Listing 1.4: Syntax of *trusted-types directive*

These directives together enable and configure Trusted Types behavior for the particular web application. They allow the application authors to define rules guarding write access to the DOM sinks and thus reducing the DOM XSS attack surface to small parts of the web application. These smaller parts can be modularized where they can be more easily monitored, reviewed, and maintained.

Apart from the standard *Content-Security-Policy* header, there is also *Content-Security-Policy-Report-Only* header which can be used to enable Trusted Types in a report-only mode. In this mode, Trusted Types violations are only interpreted as warnings. This way the application can gradually work on Trusted Types compliance without breaking the existing users. It is also recommended to use the report-only mode in production for some time to make sure the integration is working as expected [29].

Once Trusted Types are enabled, the browser changes the behavior of insecure DOM API sinks and expects "trusted" values instead of regular strings. These trusted values are created via Trusted Types policies.

1.2.3 Trusted Types policies

The core part of Trusted Types API are policies which are factory functions for creating "trusted" values that can be safely passed to DOM sinks when Trusted Types are enabled. The policies are created by the application and access to them should be restricted. In javascript, this can be easily achieved by encapsulating a policy in its own module and exporting only very specific functions which use this policy internally. The values created from the policies are unforgeable and immutable, meaning there is no way for an attacker to pass a value to dangerous DOM APIs.

```
const allowAll = (value) => value
const createHTMLCallback = allowAll;
const createScriptCallback = allowAll;
const createScriptURLCallback = allowAll;
const myPolicy = window.trustedTypes.createPolicy('my-policy', {
  createHTML: createHTMLCallback,
  createScript: createScriptCallback,
  createScriptURL: createScriptURLCallback,
});
```

Listing 1.5: Creating a Trusted Types policy

```
const trustedHtml = myPolicy.createHTML("<span>safe html</span>");
```

Listing 1.6: Create trusted value using a policy

The code listing 1.5 creates a Trusted Types policy using the callback functions. This callback function is called when the policy is used to create a trusted value. It receives a sink value of a string type as an argument and returns a string value that should be XSS free and thus can be trusted. In practice, this function may be implemented as an *identity function* if the payload is known to be trusted. Another example of policy callback implementation would be to sanitize a sink value or perform certain whitelist logic. In case the payload should not be used or can not be sanitized, the function should return *null* or *undefined* which will trigger a Trusted Types violation.

```
const myPolicy = window.trustedTypes.createPolicy('sanitize-html', {
  createHTML: (untrustedValue) => DOMPurify.sanitize(untrustedValue),
});
```

Listing 1.7: Using a policy to sanitize HTML values

It is important to ensure that policies are either secure for all possible inputs, or limit the access to insecure policies, such that they are only called with inputs that are not attacker controlled [18].

1.2.4 Default policy

There is one special case for Trusted Types policies. Applications may create a policy called "default". This policy has special behavior. When a string value is passed to a DOM sink when Trusted Types are enabled, the user agent will implicitly flow the untrusted string value through the default policy. The callback function of the default policy receives three arguments instead of one – the string payload, sink type, and a sink name respectively. This allows the application to recover from an unexpected sink usage, for example, by sanitizing the untrusted value. If the default policy does not exist, it returns *null* or *undefined* a CSP violation will be triggered [19]. Applications can use this policy to enable enforcement mode even though the application is not fully Trusted Types complaint.

This feature is intended to be used by applications with legacy or third-party code that uses injection sinks. The policy callback functions should be defined with very strict rules to prevent bypassing security restrictions enforced by Trusted Types API. For example, having an "accept all" default policy allows malicious attacker payloads to reach the DOM sinks. Developers should be very cautious when using the default policy and preferably use it only for a transitional period until the offending code is refactored not to use the dangerous DOM sinks [19].

```
trustedTypes.createPolicy('default', {
  createScriptURL: (value, type, sink) => {
    return value +
      '?default-policy-used&type=' +
      encodeURIComponent(type) +
      '&sink=' +
      encodeURIComponent(sink);
  }
});
```

Listing 1.8: Creating a default policy [19]

1.2.5 Reviewability

Consider the process of security reviews for web applications, specifically reasoning about DOM-based XSS. There are many tools and methodologies which can help reason about the application security, but there is no automated way that can assert the safety of a web application. This means that it is still necessary for security engineers to manually review the implementation and look for potential vulnerabilities and analyze them.

When focusing on client-side XSS, the engineer has to determine whether an application uses dangerous DOM sinks, either implicitly or explicitly, and whether there is a way for an attacker to misuse them.

```
function setHtml(element, html) {
  element.innerHTML = html;
}
```

Listing 1.9: Possibly dangerous function

Is the function in the listing above safe? There is not enough information to answer this. The safety of the function depends on the context of how it is used. More generally, the safety of a function depends on both its direct and indirect callers, both present and future ones [3].

```
  function processHtml(html) {
    setHtml(document.body, html);
  }

  function processUserData(data) {
    log('Processing user data');
    // Is this safe? Can this call change "data.html"?
    someThirdPartyCall(data);
    processHtml(data.html);
  }
```

Listing 1.10: Usage of the possibly dangerous function

Looking for the callers of the function can be difficult because of the dynamic nature of JavaScript. Also, JavaScript is a mutable language which makes it harder to reason about function calls, especially the third-party ones.

When the application enforces Trusted Types, the engineer does not have to care about the dangerous functions, and its callers. The focus of the security review shifts to reasoning about the creation of trusted values, and policies. Once a trusted value

is created, it is immutable and the policy which created it provides the security guarantees. When a trusted value reaches a sink, this guarantee still holds independently of how the value flew the callers. Consider the third-party call in the listing 1.10 and assume *data.html* contains a TrustedHTML value. If a third-party call modifies this value a violation is thrown when it is passed to a DOM sink.

1.2.6 Browser support and polyfill

Trusted Types are currently supported in the Chromium family of browsers [24]. Developers creating Trusted Types compliant applications should always check if Trusted Types API is available in the current execution context, which does not necessarily need to be a browser. It is very common for Node.js applications to pre-render the client-side code on the server to provide faster experience for the end users. This brings additional complexity, new attack vectors in the forms of reflected XSS, various kinds of injection and more, which are out of Trusted Types scope.

An additional benefit of Trusted Types compliant applications is that all sinks are protected. The application can use a polyfill for browsers that do not support Trusted Types, maintaining the same security properties as in the browsers where they are supported [14].

There are multiple variants of the polyfill:

- Full – Creates the API of Trusted Types, parses the HTML meta tag from the web page and enables the enforcement in the DOM.
- API only – Creates the API of Trusted Types so the application can use policies and create trusted values. However, no enforcement rules are applied.
- Tiny – Polyfills only the most important part of the API surface with a single line of code for a minimal bundle size.

1.3 Motivation and background

Trusted Types provide the opportunity to significantly reduce the risk of a client-side XSS and has been proven in various projects of various scale [29] [16]. However, projects need to refactor parts of their code, which can be difficult, especially in the open-source ecosystem where the software is composed of multiple dependencies which can not easily be modified. This presents a large barrier in Trusted Types adoption [29] with a "chicken and egg" problem. There is not enough pressure for framework authors to migrate to Trusted Types because there is not enough usage. However, there is not enough usage because a lot of applications are prevented to migrate because their dependencies do not work with Trusted Types.

Security engineers and software developers have often different priorities, the former focusing on security, the latter on the application features. Trusted Types bridge this gap by providing secure by default software. The ideal scenario would be if the open-source libraries used Trusted Types transparently to the application authors.

In this paper, we try to analyze some of the most popular open-source frameworks and libraries to show that there are workarounds for projects to use Trusted Types without too many difficulties. We want to implement some integrations and an example application to encourage application and framework authors to adopt Trusted Types to eradicate DOM-based XSS.

Chapter 2

Trusted Types integration process

In this chapter, we describe how the integration process works in general. We have implemented multiple integrations, some of which are described in this paper. We try to provide an abstract view of the common properties of integrations based on our experience. This methodology is especially important for new integration authors as it provides guidelines on how to create a new integration.

Integrating Trusted Types to applications and libraries might seem like a complex task that requires good knowledge of the project internals. There is a small sample of already implemented integrations, which shows that the necessary code changes are relatively small [16]. Based on our experience, the integration does not require the expertise of the project's inner workings either. This experience is also supported by other integration authors [29]. Each integration is different, but on a high level they follow the same principles:

1. Locate DOM sinks
2. Find a workaround for every sink found
3. Implement and release the integration

2.1 Locating DOM sinks

Locating the sinks in a framework is important for scoping the complexity and the implementation afterward. This is a hard task since there is no bulletproof way of locating all the sinks in a codebase, especially due to JavaScript being a dynamic language.

Fortunately, there are a few methods and tools, which can help to catch most of the sinks:

1. Static search
2. Code analyzers
3. Runtime analysis

2.1.1 Static search

This method is one of the simplest ones. Statically searching for sinks in a codebase produces many false positives and false negatives. The former are produced for read-only sink usage, the latter when the project contains sinks that are missed by the static search. This usually happens when the project uses dynamic property access on an HTML document or element. The output of the static search is usually cluttered with violations in tests and build tools, which need to be manually excluded from the search.

Nevertheless, this method is easy to reason about, fast to iterate and the final search results can provide a good estimate of the integration scope. This works well in practice since there is an assumption that the project developers are not malicious and do not use code patterns that are trying to hide or cause an XSS vulnerability.

There is not an actively maintained static search tool for finding DOM sinks. However, it is relatively easy to build a script, which uses the existing tools such as *grep* with a list of already known sinks [41].

2.1.2 Code analyzers

A special subgroup of static search tools is code analyzers. Code analyzers use the AST of the project source code to find the DOM sinks. The quality of the output produced proportionally depends on the quality of the AST information. When the codebase is using TypeScript, the AST information is generally richer compared to codebases using JavaScript. One can then build and use tools like Tsec [49] which uses the compiler to parse the source code and create the AST to ultimately find the sinks more reliably and produces fewer false positives.

These tools have the advantage that they can be used to maintain Trusted Types compatibility even after the violations are fixed. For example, they can be run in a CI pipeline or their output can be leveraged by linters and other language plugin tools and the errors can be shown to the developers early on and directly in their IDE [50].

2.1.3 Runtime analysis

While static search through the codebase is fast and locates most of the sinks, it usually does not find them all. The code analyzers suffer from the same problem. If the project is a library or a framework intended to be used by other applications, the integration should be thoroughly tested. The integration author should verify the integration implementation using a real-world application. This also helps to ensure backward compatibility with older versions.

That said, finding a suitable application to test the Trusted Types integration can be challenging and it may not be an ideal way to test all edge cases. For example, when implementing an integration to React, one can find a lot of real-world applications using React, but most of them are already XSS free so they can be used only to verify if the integration does not break these clients. For this reason, it is also desirable to create an application from scratch to focus more on the integration edge cases. Another benefit to creating a custom application is that many violations are found more easily through runtime analysis. We recommend doing this for all integrations.

The recommendation for application authors is to use the report-only mode of Trusted Types. This enables gradual migration and allows them to work on application features in parallel.

2.2 Finding a workaround for a sink

In this section, we assume that all the sinks discussed produce Trusted Types violations. Note, that a sink can produce a Trusted Types violation for a value that is provably secure. For example, this happens when a constant string is assigned to an *innerHTML* property of a DOM element. Generally, there are three ways how to resolve a Trusted Types violation produced by a sink:

1. Find a safer alternative
2. Wrap the value in a Trusted Types policy
3. Ensure user-supplied sink value is immutable

2.2.1 Finding a safer alternative

For some sinks, there exists a safer alternative API that can be used. For example, for non-script elements, one can use *Element.textContent* instead of *element.innerHTML*. Developers use *innerHTML* mostly because of legacy reasons since *textContent* is only available in IE from version 9 [47]. However, the support for this version has ended in January 2016 and the support for the latest IE 11 will end in June 2022 in favor of

Microsoft Edge. Due to these reasons, applications are encouraged to avoid this trick and drop the support for IE or at least IE 9.

Another use case *innerHTML* property is to assign a constant string representing the HTML markup. This is done for convenience to create a small chunk of HTML that is displayed on a page. The alternative is to create DOM nodes dynamically and only append the generated elements to the target element as a child.

Unfortunately, these workarounds can be applied only when there is a safer API alternative, which is often not the case and this solution is not applicable.

2.2.2 Wrapping the value in a policy

When there is no way to avoid using the sink and the value is trusted, developers can use a Trusted Types policy to promote a trusted value to a Trusted Types instance. One should apply a policy as soon as the value can be trusted. This usually means the place where the value is created. For example, the author should promote a value to Trusted Type immediately after sanitization or escaping. This way the unforgeability of Trusted Types can preserve the "trust" independently of how the value flows through code until it reaches the sink.

In the future, Trusted Types API might provide support for constructing trusted values from string constants via template literals without using a policy, but this is not yet supported [31].

After the value is converted to a Trusted Types instance, the application needs to make sure the value is not altered before it reaches the sink and no string function calls are attempted (2.2.3).

2.2.3 Ensuring the sink value is immutable

As we mentioned, Trusted Types instances are immutable and unforgeable values. Previously, the DOM sinks accepted string values so the application might have used string operations on the value, such as calling a common string function. Since the new values do not have string methods defined, any call attempt results in an error. Another problem is stringification which converts a Trusted Types instance to a regular string, dropping the trust guarantees provided by Trusted Types, which then throws an error when the value is passed to the sink.

The integration author needs to refactor the code to prevent modifying the trusted values to preserve the trust guarantees secured by Trusted Types.

2.3 Implementing and releasing the integration

Implementing the integration and releasing a new version of the application or library is undoubtedly an important part of the Trusted Types integration process. However, this can take a long time for multiple reasons. Some of them are described in separate sections below:

1. Reasoning about the integration
2. Trusted Types compatibility in dependencies
3. Integration complexity

2.3.1 Reasoning about the integration

Proving that the integration is correct means to determine if all sinks have been located and properly addressed. This is generally infeasible or impossible to prove. Projects usually look at patches empirically. If the integration looks correct and is properly tested then it is safe to assume that the integration is indeed correct. This is especially true when the integration is tested by large-scale organizations by a lot of services.

That being said, Trusted Types enforcement is a breaking change for projects. Libraries that can conditionally create Trusted Types should make this change opt-in. If the integration is turned on by default when Trusted Types are available in the browser, the library is risking breaking the existing applications since they might expect the values to be string instances [12]. When an opt-in configuration is not possible, the library should release the integration as a breaking change with a new major version. The alternative is to put Trusted Types integration behind a feature flag. The downside is that this can hurt the adoption [35].

2.3.2 Compatibility of dependencies

Implementing the integration in large-scale projects, which consist of many dependencies and vendors can bring a lot of overhead because to be fully Trusted Types compliant developers must ensure that all of the third-party code is Trusted Types complaint as well.

When there is non-compliant dependency, one has multiple options:

- Implement the integration for the dependency – This option leads to more integrations being needed. Also, these dependencies might be reluctant to adopt Trusted Types and accept the integration.

- Find an alternative dependency – This option is often impossible because a viable alternative might not exist. Also, replacing a dependency with an alternative increases the integration effort, and projects might not want to migrate to an alternative dependency only because of Trusted Types compliance.

- Use Trusted Types default policy – This option should only be used by applications and implementing the default policy can be difficult. Moreover, the default policy brings a cost, the application is not fully Trusted Types compliant, and using the default policy can reduce application performance.

2.3.3 Integration complexity

The difficulty of Trusted Types integrations can vary. Some integrations may require minimal effort or no code changes at all. Some integrations can be more complex, for example, the Angular integration [29].

Especially in the open-source ecosystem, the author of the integration might be an external contributor who has limited knowledge about the project. Our experience suggests, that developers can implement integrations despite not being familiar with the project internals. That being said, the project might be reluctant to merge the integration due to the introduced complexity.

Chapter 3

Integrations into preprocessors

In this chapter, we describe a large group of integrations, formally defined as preprocessors. The rest of the chapter explains our integrations. The first one is a JSX Babel plugin for the Solid.js framework. The next one is a Vite integration to support the development mode of applications. These are the two core integrations that provide the support for the development of Trusted Types compliant applications in the Solid.js framework.

It is very common for web applications to depend on multiple preprocessing tools which can perform many different things. For example, one can consider the TypeScript compiler a code preprocessor that compiles TypeScript code to JavaScript. One of the most notorious code preprocessors is Babel, which takes a modern JavaScript code and transpiles it into an older versions that are then compatible with a wide range of browsers and their versions. However, the biggest group of code preprocessors are bundlers which simplify the developer experience by providing advanced development features such as hot reloading, debugging and source map support.

These tools are essential for web application developers and are usually configured declaratively and abstracted away from the application authors. Unfortunately, these transformations might produce code that is not Trusted Types compliant. To achieve Trusted Types compliance in frameworks and applications, one must ensure that the resulting code from preprocessing tools is Trusted Types compliant or implement the Trusted Types integration themselves.

There are a lot of preprocessors that do not affect the Trusted Type compliance of a code. It is hard to formally define this group, but these are usually the code transformations that do not change the semantics of the code. For example:

- TypeScript – TypeScript (4) compiler transforms the TypeScript sources to equivalent code written in JavaScript by mostly removing the type annotations. Such transformation yields semantically equivalent code.
- Transformation to older JavaScript standards – One of the functionalities of Babel

is to transpile code written in modern JavaScript language to older standards compatible with a wide range of browsers and browser versions. Many of these transformations are solved with polyfills or simple AST replacements but the code remains semantically the same.

- Minification – The purpose of the code minification is to reduce the overall bundle size that needs to be sent by the server to the clients' browsers. In practice, this includes removing dead code, changing the names of the functions, shortening expressions and statements, however, the minified code is functionally equivalent to the non-minified one.

There are many Trusted Types compliant preprocessors that work with Trusted Types out of the box. Unfortunately, there are also tools and preprocessors which produce code that is not Trusted Types compliant. For these tools, a Trusted Types integration is needed.

3.1 Babel

Babel is an extremely large collection of tools and plugins for preprocessing source code. For this paper, the most interesting feature of Babel is JSX transformation. JSX is an XML-like syntax extension to ECMAScript without any defined semantics. It is intended to be used by preprocessors to transform the JSX markup into standard JavaScript. These preprocessors provide the semantics the JSX on its own does not have.

The JSX was originally designed to simplify the syntax for React applications by allowing mixing HTML, JavaScript, and CSS all in the same file. However, the specification [25] is generic and can be used for multiple use cases and not strictly UI related.

Solid.js JSX preprocessor

One of the JSX use cases is the JSX preprocessing inside the Solid.js framework (4.3). This framework on the surface looks like React as it is component-oriented, uses the JSX, and supports native APIs which resemble the React hooks API. The important fact is that the translated JSX is very different under the hood. React translates the JSX to React.createElement calls but Solid.js translates the JSX markup to native HTML templates. The framework differences are not that important for this paper. However, both of them use Babel to transform the easier to more convenient JSX syntax into JavaScript compatible code. See the transformation in action in the code listings below (generated using Solid.js playground [32]).

```
import { createSignal, splitProps } from "solid-js";

export function Counter(props) {
  const [local] = splitProps(props, ["ref"]);
  const [count, setCount] = createSignal(0);
  const increment = () => setCount(count() + 1);

  return (
    <button ref={local.ref} type="button" onClick={increment}>
      {count()}
    </button>
  );
}
```

Listing 3.1: Example of a component in Solid.js using JSX

```
import { template, delegateEvents, insert } from 'solid-js/web';
import { splitProps, createSignal } from 'solid-js';

const _tmpl$ = template(`<button type="button"></button>`, 2);
function Counter(props) {
  const [local] = splitProps(props, ["ref"]);
  const [count, setCount] = createSignal(0);

  const increment = () => setCount(count() + 1);

  return (() => {
    const _el$ = _tmpl$.cloneNode(true);

    _el$.$$click = increment;
    const _ref$ = local.ref;
    typeof _ref$ === "function" ? _ref$(_el$) : local.ref = _el$;

    insert(_el$, count);

    return _el$;
  })();
}

delegateEvents(["click"]);

export { Counter };
```

Listing 3.2: Example of a Solid.js component after Babel transformation

The transformed Solid.js code uses a template function (imported from *solid-js/web*). This function internally uses the *HTMLTemplateElement.innerHTML* property when

creating the template. This triggers a Trusted Types violation when the code is executed in the browser. The solution is to create a Trusted Types policy wrapping the content assigned to the *innerHTML* property. This is secure because the JSX transformation only creates the templates from statically known HTML. All dynamic and interpolated markup is rendered using JavaScript without using any dangerous sinks.

```
// Example input 1
let dynamic = "<img src=x onerror='alert(1)'>"
return (
  <iframe srcdoc={dynamic}></iframe>
);

// Example output 1
const _tmpl$ = template(`<iframe srcdoc="<img src=x onerror='alert(1)
    '>"></iframe>`, 3);

// Example input 2
let dynamic = "<img src=x onerror='alert(1)'>"
dynamic += '<br />'
return (
  <iframe srcdoc={dynamic}></iframe>
);

// Example output 2
const _tmpl$ = template(`<iframe></iframe>`, 2);

// Example input 3
<form action="/action_page.php">
  <input type="text" id="fname" value={name()} />
  <input type="text" id="lname" value={lname()} />
  <input type="submit" value="Submit" />
</form>

// Example output 3
const _tmpl$ = template(`<form action="/action_page.php"><input type="
    text" id="fname"><input type="text" id="lname"><input type="submit
    " value="Submit"></form>`, 5);
```

Listing 3.3: Examples of JSX transformations

The JSX preprocessing is able to ensure only statically known JSX values are passed to the *template* function. These statically known values are coming from the source code written by the developer, so this code is implicitly trusted. Any dynamic properties and attributes are set when the component renders and is not part of the generated template calls.

3.2 Bundlers

Bundlers deal with many different tasks such as code minification and obfuscation, loading different file types, bundling the source files into big chunks, hot reloading in development and more.

Many of the bundler responsibilities are Trusted Types compliant out of the box. However, many of the development only features break when Trusted Types are enforced. Usually, bundlers include a development server that injects some markup into DOM to provide a better developer experience for the application authors. Common examples include hot reloading changed parts of the application or displaying error details in an overlay widget.

Fortunately, Trusted Types are already supported in Webpack (starting from version 5), which is the most used bundler at the time of writing. However, there are some new promising ones that provide better performance or use ES modules which leads to simpler architecture and a more performant design.

Vite

Vite is one of the modern bundlers which serves the source files via native ES modules which have many built-in features and extremely fast hot module replacement (HMR) for development. It also provides a simple way to bundle production code using another common bundler called Rollup.

Vite is a preferred bundler for creating applications using the Solid.js framework. To support the development of these applications, we needed to create a Trusted Types integration for Vite.

We followed the integration process from chapter (2) and have identified three places that cause Trusted Types violations. These can be categorized into:

- Creation of style elements – Internally, Vite used *innerHTML* of an HTML style element to apply styles to the DOM. These two occurrences can be replaced with a safe *textContent* property.

```
if (!style) {
  style = document.createElement('style')
  style.setAttribute('type', 'text/css')
  style.innerHTML = content
  document.head.appendChild(style)
} else {
  style.innerHTML = content
}
```

Listing 3.4: Creation of style elements using *innerHTML* in Vite [52]

- Showing error overlay – Vite supports HMR during development and part of this feature is to display an error overlay when there is an error triggered by the development server. This usually happens when there is a syntax error in the code. The overlay widget uses the *innerHTML* property to inject the overlay HTML markup on top of the client's web page. In this case, the solution was to introduce a Trusted Types policy to wrap the overlay code inside the Trusted Types policy. An alternative solution would be to create the overlay markup dynamically using *document.createElement*.

  In the future, Trusted Types API might provide support for constructing trusted values from string constants via template literals without using a policy, but this is not yet supported [31].

```
export class ErrorOverlay extends HTMLElement {
  root: ShadowRoot

  constructor(err: ErrorPayload['err']) {
    super()
    this.root = this.attachShadow({ mode: 'open' })
    this.root.innerHTML = template
    (further lines omitted for brevity...)
```

Listing 3.5: Creation of error overlay using *innerHTML* property [51]

```
export class ErrorOverlay extends HTMLElement {
  root: ShadowRoot

  constructor(err: ErrorPayload['err']) {
    super()

    let policy
    if (window.trustedTypes) {
      policy = window.trustedTypes.createPolicy('vite-overlay', {
        createHTML: (s) => s
      })
    }

    this.root = this.attachShadow({ mode: 'open' })
    this.root.innerHTML = policy
      ? (policy.createHTML(template) as any)
      : template
    (further lines omitted for brevity...)
```

Listing 3.6: Creation of error overlay using Trusted Types policy [44]

Chapter 4

Integrations into web frameworks

In this chapter, we describe a partial Trusted Types support for Next.js. Then we show how to use Trusted Types with React. React already supports Trusted Types behind a feature flag and we only show how to enable this feature flag for applications created via *Create React App*. Lastly, we build upon the Solid.js integrations from the previous chapter and verify the integrations in an open-source real-world application. This shows both, that our integrations are correct and also how to migrate an application to Trusted Types.

Web frameworks and libraries is a software that is designed to support the development of web applications and services. They try to solve common problems faced in web development, such as building user interfaces, testing, building, and bundling the application. The typical web application consists of numerous libraries and software frameworks that together create the resulting web application.

4.1 Next.js integration

Next.js is one of the most popular frameworks for building web applications. The framework is built upon React which is the most used UI framework as of 2021 [26]. Parts of a Next.js application can be rendered statically, server-side, or fully client-side on a page-by-page basis. Next.js is thus not only about the client-side code, but it can also handle server-side logic which opens up different means of attacks such as reflected XSS, SQL injections and more.

Next.js was our initial choice for Trusted Types integration because of the large impact this integration would have. The framework itself seemed interested in the integration of Trusted Types for a longer time [48].

We started working on the integration and created a basic Next.js application for testing. We used a local version of Next.js as a dependency for our application. The example application was not working when Trusted Types were enforced and we needed

to make some changes in the Next.js. We ended up with a working version of the integration which supported the application in development mode with Trusted Types under enforcement mode. We managed to accomplish this with one simple Trusted Types policy in a short timeframe.

```
let policy;

const whitelistAll = (str) => str;

// The policy getter is a private part of the module
// and cannot be used directly.
const getOrCreatePolicy = () => {
  if (policy) return policy;

  policy = window.trustedTypes?.createPolicy('next', {
    createHTML: whitelistAll('createHTML'),
    createScript: whitelistAll('createScript'),
    createScriptURL: whitelistAll('createScriptURL'),
  });
  return policy;
};

export const __unsafeAllowHtml = (html) => getOrCreatePolicy()?.
   createHTML(html) ?? html;

export const __unsafeAllowScriptUrl = (scriptUrl) => getOrCreatePolicy
   ()?.createScriptURL(scriptUrl) ?? scriptUrl;

export const __unsafeAllowScript = (script) => getOrCreatePolicy()?.
   createScript(script) ?? script;
```

Listing 4.1: Example of Next.js Trusted Types API

The fixes needed were small and the Trusted Types API specific code was encapsulated in a single small module. That said, the implementation was only a proof of concept. However, we found out that there are other engineers working on this integration and we decided not pursue this project further. Instead we shifted our focus to a different projects and integrations.

4.1.1 Tsec violations

Tsec found 8 violations [46] inside Next.js sources. Out of these 7 were indeed Trusted Types violations that needed to be fixed. Some of these could be fixed simply on a type system level since they expected a value from the user. The others needed to be

explicitly allowed through a policy. The implementation for this proof of concept can be found on GitHub [45].

Since our utmost goal was to find the sinks and create a prototype for the integration we wrapped all of these values in Trusted Types objects. This was to be revisited in the future.

4.1.2 Development mode violations

Fixing the violations found by Tsec was not enough and the application still would not work under Trusted Types enforcement. This was caused by a Webpack plugin used in one of the Next.js dependencies which used eval internally. The workaround for this was to use a default policy and allow all eval calls. The proper solution would be to fix the eval issue in the Webpack plugin via a policy.

Apart from this, another violation that was triggered was caused by the application hot reloading during development. Tsec did not catch this problem since the violation came from a JavaScript file where the AST information was limited.

4.2 Create React App integration

Create React App (CRA) is a CLI and an officially recommended way to create single-page React applications. It offers an easy React application setup with no configuration. The source code of this tool does not depend on React directly. It is only used to generate the project files based on a hard-coded template and then it installs the latest version of React and other necessary dependencies.

CRA was a second project we wanted to integrate Trusted Types to. Our goal was to make sure the generated application is Trusted Types compatible.

Using Trusted Types compatible version of React

To accomplish this, we would need to change the implementation of CRA to install the Trusted Types compatible version of React. Unfortunately, such version of React is implemented only under a feature flag that needs to be turned on at build time. The published version of React has a feature flag turned off. This means that to use the Trusted Types compatible version of React, the developers need to edit the React source code and turn on the feature flag. Then build the framework themselves. This custom-built version can then be used as a dependency in the project generated by CRA.

Implementing this is non-trivial since it requires knowledge about React. More importantly, this is harder to maintain for the application authors since they need to keep up with the new releases of React manually.

Using Trusted Types compliant version of Webpack

CRA internally uses Webpack 5 to provide convenient development features, transform and bundle production applications. Some of these features cause Trusted Types violations. Gladly, Webpack can be configured to be Trusted Types compliant by a small configuration change [33].

The problem with CRA is that the Webpack configuration is hidden from the user. Unfortunately, there is no way how to override this configuration manually. The workaround is to spy on imported JavaScript modules and change their content. This pattern is often used when mocking or spying in unit tests. However, it is also a suitable solution for this case [2].

Another issue we faced was caused by *webpack-dev-server* which is only used in the development and provides features like hot module replacement or showing error overlays. We found out that the error overlay widget is created via *innerHTML* which causes a Trusted Types violation and prevents the error from being shown. The solution is to create a policy for the overlay widget and use a custom version of this dependency. Since Trusted Types are supported in Webpack we directly opened a GitHub issue [42] and proposed our solution [43].

```
// File 'scripts/start.js'
const rewire = require('rewire')
const defaults = rewire('react-scripts/scripts/start.js')
const webpackConfig = require('react-scripts/config/webpack.config')

// In order to override the Webpack configuration without ejecting the
    create-react-app
defaults.__set__('configFactory', (webpackEnv) => {
  let config = webpackConfig(webpackEnv)

  // Customize the Webpack configuration here, for reference I have
   updated Webpack externals field
  config.output.trustedTypes = {
    policyName: 'webpack-policy',
  }

  return config
})
```

Listing 4.2: Script to start React application with Trusted Types enabled in Webpack

The application is then started simply by running this script using the Node.js environment.

```
node ./scripts/start
```

Listing 4.3: Starting the CRA application

Another option is to use the *eject* command from CRA. This is a one-way operation and it unwraps all of the hidden configurations and creates these files in the project. Developers can then edit the exposed Webpack configuration which is now part of the project. This is not recommended and should be used only when other attempts fail. The generated Webpack configuration is pretty complex and updating it in the future might not be trivial.

4.3 Solid.js integration

Solid.js is a simple, modern, and reactive framework for building user interfaces. Framework syntax is largely inspired by React, but the internals are different. JSX internally makes use of HTML template elements. Both frameworks are component-oriented. The main difference is that React uses a concept called *virtual DOM* in which the UI representation is kept in memory and synced with the "real" DOM. Solid.js does not use virtual DOM and performs all UI updates directly.

We have already described the Solid.js JSX transformation in the preprocessor chapter (3.1) where we implemented a Trusted Types integration. We wanted to test the changes on a real-world project. Fortunately, there is a fully fledged full stack application which we used [5]. We made a few additional changes, specifically:

1. Used custom version of Vite and Solid.js
2. Added Trusted Types policies
3. Implemented e2e tests

Finally, we documented all the necessary changes needed to build the project and try the application with all the integrated projects [38].

4.3.1 Using custom dependencies

The real-world project uses the Rollup bundler, but we decided to replace Rollup with Vite because Vite is the preferred bundler for developing Solid.js applications and we also wanted to test that our integration works properly.

The only changes we needed to implement were to use our custom version of Vite and rename the file extension for all source files from *.js* to *.jsx* for Vite to correctly preprocess the JSX. We also used the custom version of Solid.js to generate Trusted

Types compliant code after JSX transformation. See the necessary code changes in [40].

4.3.2 Adding Trusted Types policies

After all of the dependencies were ready, we needed to enable the Trusted Types enforcement. We used a HTML meta tag and enabled policies needed. Because hot reloading in Vite reloads the full module, we had to use *'allow-duplicates'* to allow recreating the policy when it's module is reloaded. Having this is not needed for production.

```
<meta
  http-equiv="Content-Security-Policy"
  content="require-trusted-types-for 'script'; trusted-types solid-dom
   -expressions trusted-article vite-overlay 'allow-duplicates';"
/>
```

Listing 4.4: Creation of style elements using *innerHTML* in Vite [52]

We also needed to create a Trusted Types policy for the application itself. One of the source files uses a third-party API to load a HTML content and assign that into *innerHTML* property. This could easily lead to an XSS if the API was malicious or got hacked. This means that Trusted Types helped find and prevent a possible attack vector. Since the application used this only for demonstration purposes we decided to allow this pattern via a policy.

See the necessary changes in [34].

4.3.3 Implementing end to end tests

After making sure the application works as intended both in development and production, we decided to create e2e tests to verify this. We chose Cypress as the testing framework and created our testing plugin which is described in its chapter (5.1). See the implementation of these tests in [39].

Chapter 5

Integrations into testing frameworks

In this chapter, we describe a Cypress plugin that we created to simplify the testing of Trusted Types in web applications. This plugin is created from scratch, verified with many tests, and also used to test the real-world application written in Solid.js from the previous chapter.

Trusted Types are agnostic to the testing framework used and the type of tests. It is very common for a web application to have a combination of unit and end-to-end tests. The former are used to test smaller parts of applications and typically run in the Node.js environment, where neither DOM nor Trusted Types are available. The latter usually test the whole application in a browser environment and test the application features from the user perspective.

5.1 Cypress Trusted Types plugin

Cypress is one of the most popular frameworks for end-to-end testing of web applications [10]. It enjoys a rich ecosystem of plugins and supporting software. It runs the tests in real browsers. Developers can test Trusted Types compliant applications in Cypress out of the box because most of the Cypress commands only query the DOM which does not produce Trusted Types violations.

Even though Cypress supports Trusted Types out of the box, there are a few nuances the developer has to overcome to be able to test the application and Trusted Types violations. For this reason, we created a Cypress plugin that abstracts the low-level details and provides a nicer API for the developers to use.

5.1.1 Applying the CSP header

Cypress removes the CSP header sent by the application server to load the application in an *iframe*, which could otherwise be prevented by a *scriptSrc* CSP directive. There is a workaround for now, which is to intercept the request of the initial HTML application

payload and copy the CSP policy from the response headers to the response body inside an HTML meta tag. [4].

This is tedious for the developers, so we created a custom command in our plugin which does exactly this.

```
// NOTE: Based on https://glebbahmutov.com/blog/testing-csp-almost/
Cypress.Commands.add('enableCspThroughMetaTag', (options) => {
  const { urlPattern } = options ?? {};

  // Intercept all requests by default
  cy.intercept(urlPattern ?? '**/*', (req) => {
    return req.reply((res) => {
      const csp = res.headers['content-security-policy'];
      if (!csp || typeof res.body !== 'string') return;

      res.body = res.body
        .replace(
          new RegExp('<head>([\\s\\S]*)</head>'),
          new RegExp(`<head><meta http-equiv="Content-Security-Policy"
    content="${csp}">$1</head>`).toString()
        )
        // The following are needed because the regex replacement
   above inserts some characters
        .replace('/<head>', '<head>')
        .replace('<\\/head>/', '</head>');
    });
  }).as('enableCspThroughMetaTag');
});
```

Listing 5.1: Intercept requests and enable CSP header inside via meta tag

5.1.2 Testing the violations

While it is easy to test whether Trusted Types are supported (5.2) it is harder to test whether some part of the code triggered Trusted Types violation. The reason is that the violation does not produce any user-visible behavior because the DOM is unchanged and an uncaught exception is thrown and Cypress automatically fails the test. The developer needs to listen to *uncaught:exception* handler from Cypress and explicitly recognize a Trusted Types violation based on the error thrown and tell Cypress that this error is intended so it does not fail the test.

```
it('supports TT', () => {
  expect(window.trustedTypes).not.to.be.undefined;
});
```

Listing 5.2: Test Trusted Types support

To simplify this low-level handling for the developers we created a custom command which recognizes Trusted Types violations and remembers them for the lifetime of the current test so they can be asserted later on.

```
Cypress.Commands.add('catchTrustedTypesViolations', () => {
  if (catchTrustedTypesViolationsEnabled) return;
  catchTrustedTypesViolationsEnabled = true;
  cy.clearTrustedTypesViolations();

  // https://docs.cypress.io/api/events/catalog-of-events#To-catch-a-
   single-uncaught-exception
  cy.on('uncaught:exception', (err) => {
    const type = violationTypes.find((type) => err.message.includes(
   quote(type)));
    if (type) {
      trustedTypesViolations.push({
        type,
        message: extractViolationMessage(err),
        error: err,
      });
      // Return false to prevent the error from failing this test
      return false;
    }
  });
});
```

Listing 5.3: Custom command to catch Trusted Types violations

The final part of the plugin is the API to assert the caught violations. There are multiple commands which assert that certain types of violations happened. An example of a full Cypress test which asserts the violation is listed in (5.4).

```
it('assertTrustedTypesViolations', () => {
  cy.contains('unsafe html').click();
  cy.contains('unsafe html').click();
  cy.contains('duplicate policy').click();
  cy.contains('unsafe script').click();

  cy.assertTrustedTypesViolations([
    {
      type: 'TrustedHTML',
      message:
        "Failed to set the 'srcdoc' property on 'HTMLIFrameElement':
   This document requires 'TrustedHTML' assignment.",
    },
    {}, // No assertion is made for this violation
    {
      type: 'TrustedTypePolicyFactory',
      message: `Failed to execute 'createPolicy' on '
   TrustedTypePolicyFactory': Policy with name "my-policy" already
   exists.`,
    },
    { type: 'TrustedScript' },
  ]);
});
```

Listing 5.4: Example Trusted Types violation test

5.1.3 Releasing the plugin

The plugin is implemented in a standalone repository [36] and published as an npm (6) package [37] for anyone to use. Together with the plugin API there is an example application and tests which showcase the plugin in action. The plugin is to be added to the cypress community list of plugins for increased visibility.

Chapter 6

Conclusion

We showed the support of Trusted Types in various open source technologies and discussed their integrations. We support the claims from the empirical research for web frameworks [29].

We discussed Trusted Types integrations for various libraries, frameworks, and supporting software. We see a lot of opportunity for further research, integrations, and tools to be created to make Trusted Types usage easier and more widespread.

We implemented a Trusted Types integration into Solid.js together with an example application written in this framework. We showed how integration can cascade as we needed to implement minor changes in multiple packages. The final implementation was fairly small and not that difficult. We also implemented Cypress end-to-end tests for the application showing that testing is not a problem with Trusted Types either with a testing plugin we created.

As a next step, we would like to merge our Trusted Types Solid.js integration into the framework itself as it currently lives only on our forked repositories. It would be nice to see the integration working on multiple real-life applications which run also in production. We would like to see more web platform primitives be created to make Trusted Types migration simpler, for example, via HTMLSanitizer [13]. We would like to contribute with more open-source integrations, tooling, and use our knowledge to help other integration authors. All of our work is open-sourced and available for anyone to see. All repositories we used during the research and implementation are used as submodules in the main repository.

Unfortunately, we do not see strong demand in the open-source community for Trusted Types compliant applications and libraries as of now. We hope this will gradually improve. We hope that our work will encourage other people to create more integrations and that together we will make the web a safe place.

Bibliography

[1] Anirudh Anand. Xss dom source and sink definition, February 2019. https://blog.0daylabs.com/2019/02/24/learning-DomXSS-with-DomGoat/#source.

[2] ashvin777. Modify webpack configuration in cra (issue comment). https://github.com/facebook/create-react-app/issues/10307#issuecomment-898889701.

[3] Multiple authors. Trusted types design history, September 2019. https://github.com/w3c/webappsec-trusted-types/wiki/design-history/8a57f5cb1a7773cfa512943e9b7560813d61a83f#why-client-side-trusted-types.

[4] Gleb Bahmutov. Figure out how to load site with content-security-policy without stripping header (github issue). https://github.com/cypress-io/cypress/issues/1030#.

[5] Ryan Carniato. Solid real world project. https://github.com/solidjs/solid-realworld.

[6] Cloudflare. What is cross-site scripting. https://www.cloudflare.com/learning/security/threats/cross-site-scripting/.

[7] cure53. Dompurify 2.0.0 release. https://github.com/cure53/DOMPurify/releases/tag/2.0.0.

[8] MDN Web Docs. Content security policy (csp). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

[9] Microsoft docs. Internet explorer 11 desktop application ending support for certain operating systems. https://docs.microsoft.com/en-us/lifecycle/announcements/internet-explorer-11-end-of-support.

[10] Sacha Greif and team. State of js 2021 - testing. https://2021.stateofjs.com/en-US/libraries/testing.

[11] Imperva. Reflected cross site scripting (xss) attacks. https://www.imperva.com/learn/application-security/reflected-xss-attacks/#:~:text=Reflected%20XSS%20attacks%2C%20also%20known,enables%20execution%20of%20malicious%20scripts.

[12] Uzlopak (DOMPurify issue #361). Trusted types breaking applications using dompurify in chrome 77. https://github.com/cure53/DOMPurify/issues/361.

[13] Krzysztof Kotowicz. Trusted types - mid 2021 report. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/2cbfffc0943dabf34c499f786080ffa2cda9cb4c.pdf.

[14] Krzysztof Kotowicz. Trusted types increase security also in browsers without tt support. https://github.com/w3c/webappsec-trusted-types/wiki/FAQ#do-trusted-types-address-dom-xss-only-in-browsers-supporting-trusted-types.

[15] Krzysztof Kotowicz. Trusted types background, December 2021. https://github.com/w3c/webappsec-trusted-types/wiki/Effects-of-deploying-Trusted-Types-on-browser-extension-developers/c110df6329c46363c65e3c3de4ef24c5fbcecee9#background.

[16] Krzysztof Kotowicz. Trusted types integrations, November 2021. https://github.com/w3c/webappsec-trusted-types/wiki/Integrations/3e5ff5b7613f18c15a1895129fec063d5491ff83.

[17] Krzysztof Kotowicz. Trusted types resources, July 2021. https://github.com/w3c/webappsec-trusted-types/wiki/Resources/22dcd3a4cc55fae2e1706b47f9d26feb2aacaefb.

[18] Krzysztof Kotowicz and Mike West. Best practices for policy design. https://w3c.github.io/webappsec-trusted-types/dist/spec/#best-practices-for-policy-design.

[19] Krzysztof Kotowicz and Mike West. Default policy. https://w3c.github.io/webappsec-trusted-types/dist/spec/#default-policy-hdr.

[20] Krzysztof Kotowicz and Mike West. Trusted types goals. https://w3c.github.io/webappsec-trusted-types/dist/spec/#goals.

[21] Krzysztof Kotowicz and Mike West. Trusted types goals. https://w3c.github.io/webappsec-trusted-types/dist/spec/#non-goals.

[22] MDN. Csp: require-trusted-types-for. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-trusted-types-for.

[23] MDN. Csp: trusted-types. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/trusted-types.

[24] MDN. Trusted types compatibility. https://developer.mozilla.org/en-US/docs/Web/API/Trusted_Types_API#browser_compatibility.

[25] Inc. Meta Platforms. Jsx specification. https://facebook.github.io/jsx/.

[26] Stack overflow insights. Web frameworks statistics for 2021, 2021. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-web-frameworks.

[27] OWASP. Cross site scripting prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[28] OWASP. Dom-based xss. https://owasp.org/www-community/attacks/DOM_Based_XSS.

[29] Krzysztof Kotowicz Pei Wang, Bjarki Ágúst Guðmundsson. Adopting trusted types in production web frameworks to prevent dom-based cross-site scripting: A case study. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/9c56e856f0ea76454f01cabec9959f7c5b31b285.pdf.

[30] Portswigger. Dom-based xss. https://portswigger.net/web-security/cross-site-scripting/dom-based.

[31] shhnjk. Create [trustedtypes].fromliteral method (github issue). https://github.com/w3c/webappsec-trusted-types/issues/347.

[32] Solid.js team. Solid.js playground. https://playground.solidjs.com/.

[33] Webpack team. Webpack trusted types documentation. https://webpack.js.org/configuration/output/#outputtrustedtypes.

[34] Emanuel Tesar. Add trusted types policies for solid real worl project. https://github.com/Siegrift/solid-realworld/commit/4e0ace099712e4dd107d4cf1c817eb6686cd8a1c.

[35] Emanuel Tesar. Add trusted types to react on client-side pr. https://github.com/facebook/react/pull/16157.

[36] Emanuel Tesar. cypress-trusted-types (github). https://github.com/Siegrift/cypress-trusted-types.

[37] Emanuel Tesar. cypress-trusted-types (npm). https://www.npmjs.com/package/@siegrift/cypress-trusted-types.

[38] Emanuel Tesar. Document trusted types integration for solid real worl project. https://github.com/Siegrift/solid-realworld/commit/e72662cd5b834b749e25cb08cab346c2f1efa926.

[39] Emanuel Tesar. Implement e2e tests for trusted types integration for solid real worl project. https://github.com/Siegrift/solid-realworld/commit/3de79726ed0799c3b1e908fcb9d263cff4b301be.

[40] Emanuel Tesar. Replace rollup with vite in solid real worl project. https://github.com/Siegrift/solid-realworld/commit/c087818effaf0de76bd7546800f790057de8a52d.

[41] Emanuel Tesar. Static xss code scanner. https://github.com/Siegrift/xss-sink-finder/tree/5c98c3e80e16c15bda2ccc0609288cc536601a83.

[42] Emanuel Tesar. Support trusted types in webpack dev server (github issue). https://github.com/webpack/webpack-dev-server/issues/4400.

[43] Emanuel Tesar. Support trusted types in webpack dev server (github pr). https://github.com/webpack/webpack-dev-server/pull/4404.

[44] Emanuel Tesar. Vite usage of trusted types policy to create error overlay. https://github.com/Siegrift/vite/blob/8134f9244a5916a090a55e5c5c690061c0c36633/packages/vite/src/client/overlay.ts#L125.

[45] Emanuel Tesar. Fix tsec violations in next.js, October 2021. https://github.com/Siegrift/next.js/commit/a45df82d5e86f7cbaa9f0e327f1b4ea0e956fbdf.

[46] Emanuel Tesar. Tsec output for next.js, October 2021. https://github.com/Siegrift/next.js/blob/a45df82d5e86f7cbaa9f0e327f1b4ea0e956fbdf/tsec-packages-next-output.

[47] MDN textContent docs. Node.textcontent mdn browser compatibility. https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent#browser_compatibility.

[48] TyMick. Add trusted types policy (next.js pr), 2020. https://github.com/vercel/next.js/pull/13509.

[49] Google (unofficial product). Tsec. https://github.com/google/tsec/commit/a57933da74af5aef5fd2342f207b03c4fe305003.

[50] Google (unofficial product). Tsec language service plugin documentation. https://github.com/google/tsec/tree/a57933da74af5aef5fd2342f207b03c4fe305003#language-service-plugin.

[51] Vite. Vite usage of innerhtml to create error overlay. https://github.com/Siegrift/vite/blob/788d2ec1dc9853be4ecdef67d1a458a2b46ec9bf/packages/vite/src/client/overlay.ts#L124.

[52] Vite. Vite usage of innerhtml to create style elements. https://github.com/Siegrift/vite/blob/788d2ec1dc9853be4ecdef67d1a458a2b46ec9bf/packages/vite/src/client/client.ts#L259.