Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Trusted Types integration into open source frameworks and libraries

Masters Thesis

2021
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.

Bratislava, 2021
Emanuel Tesař, Bc.

Acknowledgments:

Abstrakt

Kľúčové slová: Trusted Types, Web APIs

Abstract

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 functions secure by default. Currently, they are supported in Chrome, Edge and Opera.

Integrating Trusted Types in web application requires code changes. The problem is when these changes need to be made in third party code which 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.

Keywords: Trusted Types, Web APIs

Contents

List of Figures

List of Tables

Concepts and definitions

1. **DOM source and sink** - In the context of XSS, a DOM source is the location from which untrusted data is taken by the application (which can be controlled by user input) and passed on to the sink (e.g. location, cookies). Sinks are the places where untrusted data coming from the sources is actually getting executed resulting in DOM XSS (e.g. eval, element.innerHTML) [1].

2. **Hot reload** - Hot reload just displays the code changes according to new code changes without restarting the app.

3. **DOM XSS** - DOM Based XSS is an XSS attack wherein the attack payload is executed as a result of modifying the DOM environment in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner [19].

4. **CSP** - 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 [6]. CSP provides a way for browsers to create safer APIs in a backwards compatible manner, since the API has to be opt in explicitely 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 standard browser behaviour applies.

Chapter 1

Introduction

1.1 Trusted Types

Trusted Types is a relatively modern web API designed by Google based on a long history of mitigating XSS [3].

It is a browser security feature that limits access to dangerous DOM APIs to protect against DOM XSS (3). Trusted Types provide type guarantees to all frontend code by enforcing security type checks directly in the web browser. They are delivered through a CSP header and have a report-only mode that does not change application behavior and an enforcement mode that may cause user-observable breakages [9].

When enforced, Trusted Types block dangerous injection sinks (1) from being called with values that have not passed through a Trusted Types policy [9].

There are many other resources which can be used to explore Trusted Types in depth [11].

1.1.1 Threat model

Trusted Types is a powerful API with a well scoped threat model. The main goals of the API are [13]:

- Reduce the risk of client side vulnerabilities caused by powerful and insecure by default DOM APIs
- Replace the insecure by default APIs with safer alternatives which are hard to misuse
- Encourage a design where the code affecting the application security is encapsulated in a small part of an application
- Reduces the security review surface for applications and libraries

The main idea behind Trusted Types is to replace the dangerous APIs with safer alternatives. It is very tempting to extend the threat model of Trusted types to cover other surfaces such as various server side attacks. However, this is a very complex area and it is part of the non goals of Trusted Types [14]:

- Prevent or mitigate server side generated markup attacks. To address these solutions, use the existing recommended solutions like templating systems or CSP script-src
- Control subresource loading. Trusted Types deal with code running in realm of the current document and does not guard subresources
- Guard cross origin JavaScript execution, for example loading new documents via data: URLs
- Protect against malicious developers of the web application

1.1.2 Content Security Policy (CSP)

Trusted Types are enabled through a CSP (4) using two directives:

- *require-trusted-types-for* - This directive instructs user agents to control the data passed to DOM XSS sink functions ([15]).

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

Listing 1.1: Syntax of require-trusted-types-for directive

- *trusted-types* - This directive instructs user agents to restrict the creation of Trusted Types policies ([16]). 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 1.2: Syntax of trusted-types directive

These directives together enable and configure Trusted Types behaviour for the particular web page. They allow the application authors to define rules guarding writing values to the DOM sinks and thus reducing the DOM XSS attack surface to small, isolated parts of the web application codebase, facilitating their monitoring and code review.

Apart from the standard *Content-Security-Policy* header there is also *Content-Security-Policy-Report-Only* which you can use to enable Trusted Types in report only mode. This way the application can gradually work on Trusted Types compliance without breaking the existing application.

Once the Trusted Types are enabled with the regular or the report only CSP, the browser changes the behaviour of insecure DOM API sinks and expects "trusted" values instead of regular strings. These trusted values are created via Trusted Types policies.

1.1.3 Trusted Types policies

The core part of Trusted Types API are policies which are factory functions for creating "trusted" values which can be assigned 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 creating a policy in it own module and exporting only a very specific functions which use this policy internally.

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

Listing 1.3: Creating Trusted Types policy

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

Listing 1.4: Create trusted value using a policy

The code listing 1.3 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 the sink value of string type as an argument and has to return a string value that is XSS free and thus can be trusted. In practice, this function may be implemented as identity if the payload is known to be safe. Another good example would be to sanitize the sink value inside this callback. In case the payload should not be used, you can return null or undefined which will trigger a Trusted Types violation. The callback function should never throw.

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

Listing 1.5: Using a policy to sanitize HTML values

1.1.4 Default policy

There is one special case for Trusted Types policies. Application may use create a policy called "default". This policy has special behaviour. When a string value is passed to a DOM sink when Trusted Types are enforced, the user agent will implicitely call pass the unsafe value through the default policy with the string payload, sink type and sink name respectively instead of triggering a Trusted Types violation immediately. This allows the application to recover from an unexpected sink usage, for example by sanitizing the unsafe value. If the default policy does not exist or returns null or undefined a CSP violation will be triggered [12].

This feature is intended to be used by applications with legacy code that uses injection sinks. The callback functions of the policy 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 defeats the whole purpose of Trusted Types API. Developers should be very cautious when using the default policy and preferably use it only for a transition period until the legacy code is refactored not to use the dangerous DOM sinks [12].

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

Listing 1.6: Creating a default policy [12]

1.1.5 Reviewability

Consider the problem of security reviews of web applications. There are many tools and methodologies which can help reason about the application security, but there is no automated way that can assert the application safety. This means that it is still necessary for security engeneers to manually review the implementation and look for potential vulnerabilities and analyze them. Depending on the implementation this may be easy or it can be difficult process.

When focusing on client side XSS safety, the engineer has assess whether application uses dangerous DOM sinks and whether there is way for attacker to misuse this.

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

Listing 1.7: Possibly dangerous function

Is the function in the listing above safe? There is not enough information to answer this. The safety of this function depends on how it is used. More generally, the safety of this function depends on it 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 1.8: Usage of the possibly dangerous function

Looking for the callers of the function is difficult because of the dynamic nature and 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 doesn't 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 is guarantees the security. When a trusted value reaches a sink, this guarantee still holds independently of how the value passed around in the future callers. Consider the third party call in the listing 1.8 and assume *data.html* contains a TrustedHTML value. If the third party call modified this value a violation would be thrown when passed to the DOM sink. It could also replace the trusted value with a different TrustedHTML value. However, the new value is again trusted, so this is not a security vulnerability, but an application feature.

This design reduces the code surface of application during security reviews and makes it easier to reason about the application security.

1.1.6 Browser support and polyfill

Trusted Types are currently supported in Chromium family of browsers [17]. Developers creating Trusted Types compliant application should always check if Trusted Types are available in the current execution context, which does not necessary need to be a browser. It is very common for nodeJS applications to pre render the client side code on the server to provide faster experience for the end users. This brings additional complexity and new attack vectors in forms of reflected XSS, various kinds of injection and more, which are out of the threat model under which Trusted Types operate.

The good thing about Trusted Types is that once the application is Trusted Types compliant then all of the application sinks are protected and thus the application has the same security properties in browsers where Trusted Types are supported and in the ones where they are not.

1.2 Web frameworks and libraries

Web frameworks and libraries is a software that is designed to support development of web applications and services related to web applications. Frameworks try to solve common problems faced in web development, such as building and reusing user interfaces, managing state, performing API requests and enforcing the best security practices.

1.2.1 Next.js

Next.js is one of the most popular frameworks for building web applications. The framework is build upon React library which is the most used library as of 2021 [18]. There are many reasons for why Next.js is among the most popular frameworks. Notably, it is opinionated about how the application backend should work. This design choice allows Next.js to decide the best possible way to build a particular web page. The important fact to take away is that Next.js is not only about the client side (the DOM), but it can also handle server side rendering, data processing, API requests which open up different means of attacks such as reflected XSS, SQL injections and more.

Next.js is a complex framework when looking at the source code. As of Oct 6, 2021 it consists of more then 200,000 lines of code. Looking at each file or have a complete understanding of the entire codebase is impossible for external contributor.

1.2.2 Create React App

Create React App is an officially supported way to create single-page React applications. It offers a modern build setup with no configuration. It is a CLI which creates and configures minimal React application. This is ideal for starting new projects as the configuration is abstracted away from the user.

The source code of this tool is not dependant on React directly. It only focuses on generating the project files from the template and then it installs React and other necessary dependencies.

Chapter 2

Trusted Types integration

Integrating Trusted Types to target applications and libraries is a complex task that requires good knowledge of the target inner workings. There is a small sample of already implemented integrations, which show that the necessary code changes are relatively small [10]. Each integration is different, but on a high level they follow the same steps:

1. Locate all of the DOM sinks (1)

2. Find the most suitable workaround for every sink found

3. Merging the integration and releasing a new version of the target

2.1 Locate all of the DOM sinks

Locating the sinks of the integration target is important for scoping the integration effort and the implementation afterwards. This task is complex since the problem of "locating all of the sinks in a codebase" generally cannot be proven. This is because JavaScript is a dynamic language and it is possible to access and set the sink value dynamically (using the property element access).

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

1. Static search through the codebase

2. Static code analyzers

3. Using the target in an real world application

2.1.1 Static search through the codebase

This method is the simplest one and arguably the least effective. Statically searching for sinks through codebase produces many false positives and it is also possible to miss some. The output of the static search is usually cluttered with less important violations in tests and built tools, which you need to manually exclude 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 target source code is not malicious and doesn't actively use code patterns which are trying to hide or cause an XSS vulnerability.

There is not an actively maintained static search tool, but it is relatively easy to build a script, which uses the existing tools such as *grep* with a list of already known sinks [24].

2.1.2 Static code analyzers

A better approach for locating the DOM sinks statically, is to search through the AST of the code. The quality of the output produced 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 [28] which uses the compiler to parse the code and create the AST to ultimately find the sinks in a more reliable manner.

These tools have an advantage that they can be used to maintain the Trusted Types compatibility even after the violations are fixed. For example, they can be run in CI pipeline or their output can be leveraged by linters and other language plugin tools and the errors can be shown to the developers directly in their IDE [29].

2.1.3 Using the target in a real world application

While static search through the codebase is fast and locates most of the sinks, it usually doesn't catch them all. The author of the integration should verify the integration on the applications using the target as a dependency. Also, having good test coverage in the application code greatly increases the chance that the integration is implemented correctly.

That said, finding a suitable application might 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 lot of real world applications using React, but most of them are already XSS free so you can only verify if the integration doesn't break these clients. What you want is to also test how the integration works when used with applications that uses the DOM sinks and now has to produce Trusted Types values. Another test vector is simulating an attacker by constructing malicious payloads and expect the site to break due to Trusted Types violations. These are done best by creating an application from scratch by the integration author.

2.2 Merging the integration and releasing a new version of the target

Merging the integration and releasing a new version of the target is undoubtedly an important part of Trusted Types integration. However, this can take long time for multiple reasons, which are described in separate sections:

1. Reasoning about the integration, section 2.2.1
2. Trusted Types compatibility in dependencies, section 2.2.2
3. Knowledge of the integration author, section 2.2.3

2.2.1 Reasoning about the integration

Proving that the integration is correct is often hard or impossible. However, proving implementation correctness in general is often impossible, so this is not that big of an issue. One can look at the integration empirically instead - if the integration is tested on multiple projects and they all seem to work it is safer to assume that the integration is correct. This is especially true when the integration is tested by large scale organizations on lot of their services.

That being said, Trusted Types integration is a breaking change and targets should make this change opt-in or release a new version with breaking changes [5]. If the integration is turned on by default when Trusted Types are available in the browser, the target is risking breaking existing applications [8].

Releasing an opt-in change might be problematic if the target is not configurable. This can result in the integration being put behind a feature flag which hurts the adoption [21].

2.2.2 Trusted Types compatibility in dependencies

Implementing the integration in large targets, which consists of many dependencies can bring a lot of overhead, because to be fully Trusted Types compliant one must ensure that all of the dependencies are Trusted Types complaint as well.

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

- Implement the integration for the dependency - This option has all the problems already mentioned and can possible lead to even more integrations.
- Find an alternative dependency - This option is often impossible as a viable alternative might not exist.
- Use Trusted Types default policy - This option is applicable only for applications, is more complex and the application is not fully Trusted Types compliant.

2.2.3 Knowledge of the integration author

It is likely that the integration author is either a security engineer familiar with Trusted Types or a part of the engineering team of the target familiar with the target code. This means that the author might not be fully familiar with both of the technologies, which results in both:

- Increased propability of a bug in implementation
- Harder and longer review, since the reviewers are likely in a similar position

2.3 Recommended setup for libraries and frameworks

The following section describes the recommended setup on how to prepare for the implementation of the integration based on our experience. We also assume, the target code is put under version control, such as *git*.

This section should give the reader a high level overview of how the integration setup is performed, not a definitive guide. We also use this section as a guideline when implementing the integrations. The general structure is very simple:

1. Clone the repository of the library locally
2. Create an application which uses your local version as a dependency

2.3.1 Clone the repository of the library locally

Having the repository cloned locally has an advantage that you can keep your integration up to date with the latest changes. It is also probably required to submit the integration once implemented.

2.3.2 Create an application which uses your local version as a dependency

This is the more interesting part, but it is also very project specific. For web development, the dependencies are usually managed with package managers such as *npm* or *yarn*. Both of these allow you to configure local dependencies. The only thing that the integration author needs to do is to be able to build the target as a dependency for the application.

Afterwards, one can implement the integration and verify assumptions on the application simultaneously, which can greatly speed up the integration. Also, after the integration is implemented, one can have a present the application code as an empiric proof of the integration correctness.

Chapter 3

Trusted Types integration into Next.js

Next.js (described in 1.2.1) was the initial choice for Trusted Types integration, because of a large impact this would have and also because the framework seemed interested and open to the integration of Trusted Types for a longer time [27].

The integration setup followed chapter 2. There were changes needed in both the application and Next.js code and we ended up with a working version of a blank application running in dev mode with Trusted Types aware Next.js as a dependency. However, we decided not to pursue this project no more and shifted focus to a different target.

3.1 Fixing violations reported by Tsec

Tsec found 8 violations [26] 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 allowed explicitely by using a policy. The implementation for this can be found on github [25].

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.

3.2 Fixing the dev mode of an example application

Fixing the violations found by Tsec was not enough and the application still wouldn't launch with Trusted Types enforced. The reason for this hasn't been clear, but it was probably a webpack plugin used in one of the Next.js dependencies. The violating code relied on eval, which threw an error when Trusted Types were enforced. The workaround for this was to use a default policy and allow all eval calls.

Apart from this, there was another violation that was triggered when the application wanted to hot reload (2). Tsec wasn't able to catch this, since the violation came from a JavaScript file where the AST information was limited.

Chapter 4

Trusted Types integration into Create React App

Create React App (CRA) (described in 1.2.2) was a second choice for integrating Trusted Types. After taking a look in the implementation of CRA we found that this tool doesn't depend on React directly, but only installs it after creating the project template. Our goal was to make sure the generated project by CRA is Trusted Types compatible.

4.1 Using Trusted Types compatible version of React

To accomplish this, we would need to change the implementation of CRA to install Trusted Types compatible version of React. Unfortunately, such version of React is implemented under a feature flag which needs to be turned on at build time. The published version of React contains the already built files with the feature flag turned off. This means that if you want to use the Trusted Types compatible version of React you have to clone the React repository from sources, turn on the feature flag and then build the framework. You can then use this version of React as a dependency in your project.

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

4.2 Using Trusted Types compliant version of Webpack

CRA internally uses webpack version 5 to provide fast development experience with convenient features, such as hot reload. For production it handles minification and other optimizations. Some of these features, especially hot reload breaks under Trusted Types in enforcement mode. Gladly, webpack can be configured to be Trusted Types compliant by a small configuration change [20].

The problem with CRA is that the webpack configuration is abstracted away from the user. This means that there is not a simple way how to change the configuration. One can find the webpack configuration in the downloaded dependencies and change it there directly, but this won't really work since those changes will be removed by the package manager when modifying the dependencies. In general, one should never modify the contents of dependencies and treat it only as an output of a package manager.

The optimal solution as of now is to use a package which allows you to spy on modules and change what they return. This pattern is often used when mocking or spying in unit tests. However, it is also a suitable workaround for this case [2]. CRA team is reluctant to change this, since the hidden configuration is a good default for the majority of users.

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

Chapter 5

Testing

Trusted Types are agnostic to the testing framework used and type of tests. It is very common for web application to have a combination of unit and end to end tests. The former are used to smaller parts of application and typically run in nodeJS environment. The latter usually test the whole application in browser environment and focus on the full application as a product.

5.1 Cypress

Cypress is one of the most popular frameworks for end to end testing of web applications [7]. It enjoys a rich ecosystem of plugins and supporting software. It runs the tests in real browsers and is fast and reliable due to its unique architecture.

Since the tests run in a browser developers can test the Trusted Types compliance of their application. Cypress loads the user application in an iframe, in which Trusted Types can be enforced without the need for Trusted Types compliance of Cypress itself. This means that Cypress commands are allowed to use dangerous sinks, because they run outside of the iframe. This is extremely important, because it makes all of the existing Cypress ecosystem work out of the box under Trusted Types enforcement.

5.1.1 Trusted Types plugin

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 have created a Cypress plugin which abstracts the low level details and provides a nicer API for the developers to use.

Removed CSP header

Cypress removes the CSP header sent by the application server in order to load the application in an iframe, which could 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 a *meta* tag. [4].

This is tedious for the developers, so we have 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

Testing the violations

While it is easy to test that the Trusted Types are supported (5.2) it is harder to test whether part of the code triggered Trusted Types violation. The reason is that the violation does not produce any user visible behaviour because the DOM is unchanged and uncaught exception is thrown and Cypress automatically fails the test. The developer needs to listen to *uncaught:exception* handler from Cypress and explicitely recognize the Trusted Types violation based on the error thrown and tell Cypress that this error is intended so it doesn't 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 have 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 violationTypes = ['TrustedHTML', 'TrustedScript', '
   TrustedScriptURL'] as const;
    const type = violationTypes.find((type) => err.message.includes(
   formatViolationMessage(type)));
    if (type) {
      trustedTypesViolations.push({
        message: formatViolationMessage(type),
        error: err,
      });
      // Return false to prevent the error from failing this test
      return false;
    }

    const policyCreationErrorMessage = formatViolationMessage('
   PolicyCreation');
    if (err.message.includes(policyCreationErrorMessage)) {
      trustedTypesViolations.push({
        message: policyCreationErrorMessage,
        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.enableCspThroughMetaTag();
  cy.catchTrustedTypesViolations();
  cy.visit(url);

  cy.contains('unsafe html').click();
  cy.contains('unsafe html').click();
  cy.contains('duplicate policy').click();
  cy.contains('unsafe script').click();

  cy.assertTrustedTypesViolations(['TrustedHTML', 'TrustedHTML', '
   PolicyCreation', 'TrustedScript']);
});
```

Listing 5.4: Example Trusted Types violation test

Releasing the plugin

The plugin is implemented in a standalone repository [22] and published as an npm package [23] for everyone to use. Together with the plugin API there is an example application and tests which showcase the plugin in action.

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

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

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

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

[9] 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.

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

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

[12] Krzysztof Kotowicz (Google LLC) and Mike West (Google LLC). Default policy. `https://w3c.github.io/webappsec-trusted-types/dist/spec/#default-policy-hdr`.

[13] Krzysztof Kotowicz (Google LLC) and Mike West (Google LLC). Trusted types goals. `https://w3c.github.io/webappsec-trusted-types/dist/spec/#goals`.

[14] Krzysztof Kotowicz (Google LLC) and Mike West (Google LLC). Trusted types goals. `https://w3c.github.io/webappsec-trusted-types/dist/spec/#non-goals`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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