femington

Int. J. Intell. Syst. Data Sci. (2026) 1(3):1–10
Article ID: IJISDS-2026-226

Integrating Trusted Types with Solid.js and Cypress: Enhancing Web Application Security

Santosh Kumar
santoshkuidev@gmail.com
Independent Researcher

Abstract

This paper studies the *Trusted Types*, which is an updated browser restriction to mitigate the *Cross-Site Scripting (XSS)* attacks for the Solid.js framework and the Cypress e2e testing environment. The research commences with the migration of the Solid.js build system from the Rollup bundler over to Vite, requiring minimal changes to the existing codebase while enhancing the efficiency of the development workflow. In order to achieve full Trusted Types compatibility, we relied on custom Vite and Solid.js versions. Also, we made necessary adjustments, such as renaming JavaScript source files from .js to .jsx and timplementingTrusted Types policies for generating third-party API content. Furthermore, the study describes a custom Cypress plugin used to validate the enforcement of Trusted Types in automated testing. A developer can easily configure Content Security Policy (CSP) headers and detect Trusted Types violations in real-time with the help of the plugin. It aids in making security testing more dependable. As shown in the experiments, the proposed integration strengthens application resilience against client-side injection attacks with the added benefit of efficient development and flexible testing. While usage of Trusted Types in the open source ecosystem is not widespread, it shows great potential for enhancing the security of modern web applications. The future work will extend Trusted Types support to large-scale real-world applications, improve compatibility with frameworks, and promote wide-scale adoption of secure-by-default web development practices.

Keywords

• Trusted Types • Cross-site scripting (XSS) • Solid.js • Cypress • Content Security Policy (CSP)

Concepts and definitions

This section introduces general terms used throughout the paper. Advanced readers may skip it and proceed to Chapter 1.

1. **DOM** – *Document Object Model*, representing the structure and content of a web page.
2. **DOM source and sink** – In XSS, a *source* provides untrusted data (e.g., *location*, *cookies*), while a *sink* executes it (e.g., *eval*, *Element.innerHTML*), potentially causing DOM XSS [1].
3. **SPA** – *Single-Page Application*, dynamically updates content without reloading pages.
4. **TypeScript** – A typed superset of JavaScript designed for large-scale applications.
5. **Hot reload** – Automatically restarts an application after code changes; *HMR* updates modules without full reload.
6. **npm** – Package manager for JavaScript and Node.js.

1. Introduction

This chapter introduces cross-site scripting (XSS), with emphasis on DOM-based XSS and Trusted Types as a mitigation.

XSS occurs when untrusted input is executed as code due to missing sanitization. It is commonly classified as:

- **Stored** – Malicious scripts persist on the server.
- **Reflected** – Scripts are reflected to the browser via requests [2].
- **DOM-based** – Executed in the browser via DOM manipulation (1).

With single-page applications (3), dynamic DOM updates increased XSS risks. Many DOM APIs act as *sinks* (2) that execute input [3].

Listing 1: Common DOM XSS sinks [1] [3]

```
document.write()
innerHTML
insertAdjacentHTML
DOMParser.parseFromString
eval()
```

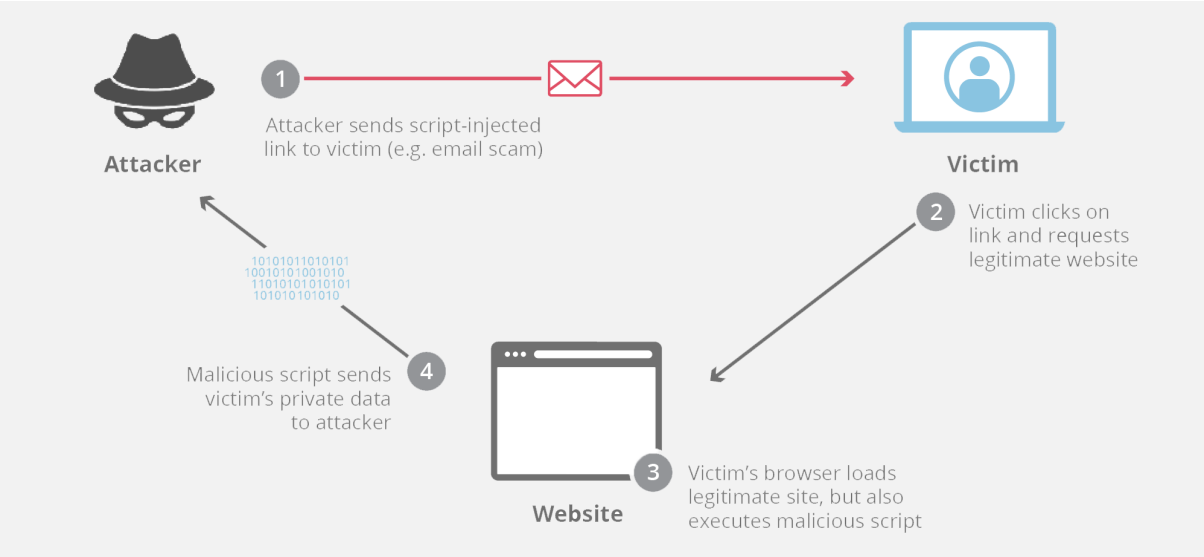

Figure 1: Common XSS vulnerability flow

A typical example is unsafe URL interpolation:

Listing 2: Basic XSS via URL

```
<div id="content"></div>
<script>
const c = decodeURIComponent(location.search.substr(1))
document.getElementById('content').innerHTML = c
</script>
```

XSS impact ranges from minor issues to account compromise and data leakage. Existing defenses (static/dynamic) are limited by scalability and JavaScript complexity [3, 4].

Trusted Types is a browser API designed to mitigate DOM-based XSS [5]. It enforces type checks on unsafe DOM APIs via CSP, offering report-only and enforcement modes. In enforcement mode, only values created through policies can reach sinks, preventing DOM XSS [6].

Its goals include reducing injection risks, replacing unsafe APIs, centralizing security logic, and simplifying reviews [7]. However, it does not address server-side XSS, subresource control, cross-origin execution, or malicious developers [8].

Content Security Policy (CSP) mitigates XSS via HTTP headers [9]. Trusted Types are enabled through the *require-trusted-types-for* and *trusted-types* CSP directives:

Listing 3: require-trusted-types-for syntax

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

Listing 4: trusted-types syntax

```
Content-Security-Policy: trusted-types <policyName>;
```

These restrict access to DOM sinks, reducing the attack surface. Report-only mode enables gradual adoption [3].

Trusted Types policies generate safe values and must be strictly controlled:

Listing 5: Policy creation

```
const p = trustedTypes.createPolicy('p', {
  createHTML: v => v
});
```

Listing 6: Using a policy

```
const safe = p.createHTML("<b>ok</b>");
```

Callbacks may sanitize or validate input; unsafe values return *null/undefined*, triggering violations.

Listing 7: Sanitization

```
trustedTypes.createPolicy('s', {
  createHTML: v => DOMPurify.sanitize(v)
});
```

Policies must be secure or limited to trusted inputs [10].

A special *default* policy processes all untrusted strings automatically. Its callback receives payload, type, and sink, enabling fallback sanitization. Missing or rejecting values triggers violations. It is mainly for legacy code and should be used cautiously and temporarily [11].

Listing 8: Default policy

```
trustedTypes.createPolicy('default', {
  createScriptURL: (v, t, s) => v
});
```

Security reviews traditionally require tracing unsafe sink usage:

Listing 9: Function writing HTML to the DOM

```
function setHtml(e, h) {
  e.innerHTML = h;
}
```

Listing 10: Usage of the DOM-writing function

```
function process(d) {
  someThirdPartyCall(d);
  setHtml(document.body, d.html);
}
```

Due to JavaScript's dynamic nature, this is difficult [5]. Trusted Types shift focus to policy creation, as trusted values remain safe regardless of data flow.

Although Trusted Types significantly reduce XSS [3, 6], adoption is limited by dependency constraints, creating a "chicken-and-egg" problem. This work explores practical integrations to encourage adoption and reduce DOM-based XSS.

2. Trusted Types integration process

This chapter outlines a general methodology for integrating Trusted Types based on practical experience. Although integrations may seem complex, prior work shows that required changes are often small and do not demand deep knowledge of project internals [3]. Despite differences across projects, integrations typically follow three steps:

1. Locate DOM sinks
2. Resolve each sink
3. Implement and release the integration

Locating sinks is challenging due to JavaScript's dynamic nature. However, three approaches can help:

- **Static search** – Simple and fast but produces false positives/negatives. Tools like *grep* with known sink patterns are commonly used [12].
- **Code analyzers** – Use AST analysis for better accuracy. Tools such as Tsec leverage TypeScript for improved results and can be integrated into CI pipelines [13].
- **Runtime analysis** – Validates integrations in real applications, helping detect missed sinks and edge cases. Report-only mode is recommended for gradual adoption [6].

Once sinks are identified, violations can be resolved in three ways:

1. **Use safer alternatives** – Replace unsafe APIs (e.g., *innerHTML*) with safer ones like *textContent* where possible.
2. **Wrap values in policies** – Convert trusted values into Trusted Types immediately after sanitization to preserve trust guarantees [14].

3. **Ensure immutability** – Prevent modification or stringification of trusted values, as this breaks guarantees and causes runtime errors.

After resolving sinks, integration must be implemented and released. Proving correctness is generally infeasible; instead, correctness is established empirically through testing [5]. Since Trusted Types enforcement can introduce breaking changes, integrations should be opt-in or released as major updates.

Dependency compatibility is a major challenge. All third-party code must be compliant, which may require integrating dependencies directly, replacing them (often impractical), or using a default policy (less secure). Integration complexity varies widely. Some require minimal effort, while others (e.g., Angular) are more involved [3]. Even contributors unfamiliar with project internals can implement integrations, though increased complexity may affect adoption.

3. Integrations into preprocessors

This chapter describes integrations categorized as preprocessors, focusing on a JSX Babel plugin for Solid.js and a Vite integration for development support. These enable Trusted Types compliant development in Solid.js applications.

Web applications commonly rely on preprocessors such as TypeScript, Babel, and bundlers. These tools transform code (e.g., transpilation, bundling, hot reloading) and improve developer experience, but may produce code that is not Trusted Types compliant. Ensuring compliance requires either safe transformations or dedicated integrations.

Some preprocessors preserve semantics and are inherently compatible, including TypeScript (4), Babel transpilation, and minification. However, others introduce non-compliant patterns, requiring integration.

Babel is widely used for JSX transformation, converting JSX syntax into JavaScript [15]. In Solid.js, JSX is compiled into native HTML templates rather than virtual DOM calls.

Listing 11: Solid.js component using JSX

```
export function Counter() {
  const [count, setCount] = createSignal(0);
  return <button onClick={() => setCount(count() + 1)}>
    {count()}
  </button>;
}
```

Listing 12: Transformed output

```
const _tmpl$ = template(`<button></button>`, 2);
const _el$ = _tmpl$.cloneNode(true);
insert(_el$, count);
```

The transformation uses a *template* function, which internally relies on *innerHTML*, causing Trusted Types violations. This can be resolved by wrapping static template content in a Trusted Types policy, which is safe because templates are derived from static code.

Listing 13: JSX transformation behavior

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

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

JSX preprocessing ensures only static values are used in templates, while dynamic values are applied safely at runtime.

Bundlers handle tasks such as bundling, minification, and development features. While many are compliant, development features (e.g., overlays, hot reloading) often introduce violations through DOM injection. Webpack supports Trusted Types natively [16], but newer tools like Vite require additional integration.

Vite uses native ES modules and fast HMR. Following the integration process (2), three main violation sources were identified: style injection via *innerHTML*, error overlay rendering, and dynamic module evaluation. Each was resolved by wrapping the relevant output in a Trusted Types policy.

In future, Trusted Types may support safe construction from literals without policies [14].

These integrations demonstrate that preprocessing tools can be adapted with minimal changes to ensure Trusted Types compliance while preserving developer experience.

4. Integrations into web frameworks

This chapter presents Trusted Types integrations in web frameworks. We describe a partial integration for Next.js, enabling Trusted Types in React via Create React App (CRA), and validate Solid.js integrations on a real-world project.

Web frameworks simplify application development by handling UI, building, and bundling. Modern applications typically combine multiple frameworks and libraries, making Trusted Types integration dependent on the entire ecosystem.

Next.js, built on React, supports client-side, server-side, and static rendering, introducing additional attack vectors. Our initial integration effort demonstrated that Trusted Types support could be achieved with minimal changes using a single encapsulated policy. This proof of concept worked in development mode, but was not pursued further due to parallel community efforts. Static analysis using Tsec revealed several relevant violations, some resolved via typing and others requiring dedicated policies. Additional runtime issues (e.g., *eval* in Webpack plugins and hot reloading) required a default policy workaround.

For React, Create React App doesn't enable Trusted Types out of the box. It offers a zero-configuration setup. React includes support behind a feature flag, requiring custom builds, which complicates maintenance. Webpack 5, used by CRA, supports Trusted Types with configuration [16], but CRA hides this configuration.

A workaround is to override Webpack settings programmatically:

Listing 14: Enable Trusted Types in CRA Webpack

```
config.output.trustedTypes = { policyName: 'webpack-policy' };
```

Solid.js, a reactive UI framework, was integrated and tested on a real-world project. The setup required custom Vite and Solid.js versions, Trusted Types policies, and end-to-end tests. Rollup was

replaced with Vite to validate integrations, and JSX files were adapted accordingly [17]. Trusted Types enforcement was enabled via CSP:

Listing 15: CSP configuration for Trusted Types

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

A custom policy was required for a third-party API assigning HTML to *innerHTML*, highlighting a potential XSS risk mitigated by Trusted Types [6]. End-to-end tests using Cypress verified correct behavior, demonstrating that Trusted Types can be adopted across frameworks with manageable effort, though challenges remain in dependencies, tooling, and configuration [3].

5. Integrations into testing frameworks

We present a Cypress plugin to simplify testing of Trusted Types in web applications. It is implemented from scratch, tested extensively, and used to validate the Solid.js real-world application.

Trusted Types are independent of testing frameworks. Unit tests (Node.js) lack DOM and Trusted Types support, while end-to-end tests run in browsers and can validate real behavior.

Cypress is a popular end-to-end testing framework [18]. It works with Trusted Types out of the box, but testing violations requires additional handling. Our plugin abstracts these details and provides a simpler API.

Cypress removes CSP headers when loading apps in an iframe. To preserve CSP, we intercept responses and inject the policy via a meta tag [19]:

Listing 16: Enabling CSP via a Meta Tag

```
Cypress.Commands.add(
  'enableCspThroughMetaTag',
  (options) => {
    cy.intercept(
      options?.urlPattern ?? '**/*',
      (req) => {
        req.reply((res) => {
          const csp =
            res.headers['content-security-policy'];

          if (!csp || typeof res.body !== 'string') {
            return;
          }

          res.body = res.body.replace(
            /<head>([\s\S]*)<\/head>/,
            `<head>
               <meta
                 http-equiv="Content-Security-Policy"
                 content="${csp}">
             $1</head>`
          );
        });
    }
```

```
    );
  }
);
```

Testing Trusted Types violations is non-trivial since they do not affect the DOM but throw exceptions. Cypress fails tests on such errors, so we capture and classify them:

Listing 17: Catch Trusted Types violations

```
Cypress.Commands.add('catchTrustedTypesViolations', () => {
  cy.on('uncaught:exception', (err) => {
    const type = violationTypes.find(t => err.message.includes(t));
    if (type) {
      trustedTypesViolations.push({ type, message: err.message });
      return false;
    }
  });
});
```

Example test:

Listing 18: Violation test

```
it('assertTrustedTypesViolations', () => {
  cy.contains('unsafe html').click();
  cy.assertTrustedTypesViolations([{ type: 'TrustedHTML' }]);
});
```

The plugin is published as an npm package [20], providing reusable utilities and example tests. This approach confirms that Trusted Types enforcement can be validated reliably in automated end-to-end pipelines [12].

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 [3].

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 [20].

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. 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 [6].

References

[1] PortSwigger. DOM-based XSS. https://portswigger.net/web-security/cross-site-scripting/dom-based.

[2] Gustavo E. Rodríguez, Jefferson G. Torres, Priscila Flores, and Diego E. Benavides. Cross-site scripting (XSS) attacks and mitigation: A survey. *Computer Networks*, 166:106960, 2020. doi: 10.1016/j.comnet.2019.106960.

[3] Pei Wang, Bjarki Ágúst Guðmundsson, and Krzysztof Kotowicz. Adopting trusted types in production web frameworks to prevent DOM-based cross-site scripting: A case study, 2021. https://storage.googleapis.com/pub-tools-public-publication-data/pdf/9c56e856f0ea76454f01cabec9959f7c5b31b285.pdf.

[4] Abdelhakim Hannousse, Salima Yahiouche, and Mohamed Cherif Nait-Hamoud. Twenty-two years since revealing cross-site scripting attacks: A systematic mapping and a comprehensive survey. *Computer Science Review*, 52:100634, 2024. doi: 10.1016/j.cosrev.2024.100634.

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

[6] Sunnyeo Park, Jihwan Kim, Seongho Keum, Hyunjoon Lee, and Sooel Son. TrustyMon: Practical detection of DOM-based cross-site scripting attacks using trusted types. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIA CCS '25)*, 2025. doi: 10.1145/3708821.3733889.

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

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

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

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

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

[12] William Melicher, Christine Fung, Lujo Bauer, and Limin Jia. Towards a lightweight, hybrid approach for detecting DOM XSS vulnerabilities with machine learning. In *Proceedings of the Web Conference 2021*, pages 2741–2753, 2021. doi: 10.1145/3442381.3449902.

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

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

[15] Meta Platforms, Inc. JSX specification. https://facebook.github.io/jsx/.

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

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

[18] Shubham Bose and Anirban Roy. Cypress: Redefining end-to-end testing for the web. *IEEE Software*, 38(4):58–64, 2021. doi: 10.1109/MS.2021.3060290.

[19] 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.

[20] Emanuel Tesar. cypress-trusted-types, . https://github.com/Siegrift/cypress-trusted-types.