XSS Defense Evaluation: A Comparative Analysis of CSP, Trusted Types, and DOMPurify Against Advanced Attack Vectors

Woojin Kim
Oscar Arbman
woojin@kth.se
oarbman@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Contents

1 Background

Cross-site scripting (XSS) remains one of the most prevalent web application vulnerabilities, allowing attackers to inject malicious scripts into websites that execute in users' browsers. Despite widespread awareness, XSS continues to pose a threat to modern web applications, as attack techniques have evolved beyond simple script injections. Today's attackers employ sophisticated methods, such as DOM-based XSS, which executes entirely on the client-side in JavaScript, and mutation-based XSS, which transforms innocent code after sanitization. Traditional defenses such as simple input validation often fail against these advanced threats.

In response to these challenges, a triad of modern defense technologies has emerged: Content Security Policy (CSP), which provides browser-enforced restrictions on resource loading; Trusted Types, which implement type-based protections against DOM manipulation attacks; and DOMPurify, an advanced HTML sanitization library designed to neutralize even sophisticated XSS vectors. While these technologies offer promising protection when properly implemented, their effectiveness against the full spectrum of modern attack techniques remains insufficiently documented and tested in real-world scenarios.

2 Goal

This project aims to systematically evaluate the effectiveness of CSP, Trusted Types, and DOMPurify against advanced XSS attack techniques by implementing a comprehensive test environment. We aim to map the security boundaries of these defenses by testing them against a catalog of attack vectors, including DOM clobbering, sanitization bypasses, CSP circumvention techniques, and Trusted Types manipulation.

Our test environment will simulate real-world conditions by implementing multiple frontend frameworks, various data entry points, and graduated security configurations. We'll document which attacks succeed and fail against different defensive setups, creating a clear visualization of protection effectiveness. Through this approach, we aim to provide a deeper understanding and offer practical guidance on effectively utilizing defenses such as CSP, Trusted Types, and DOMPurify for XSS defenses in modern web applications.

3 Description of Testing Environment Set-up

A link to the implementation and testing environment is given in section 5.2

3.1 Server Set-up

The server is located in the file *"server.js"*, which implements endpoints for testing XSS vulnerabilities and their mitigations. We used the following setup and configuration:

- **Express Framework**: Provides the web server foundation
- **Helmet**: Security middleware that sets various HTTP headers
- **Path**: Handles file path operations

- **Crypto**: Generates cryptographically secure random values for nonces
- **JSDOM**: Provides a DOM implementation for server-side JavaScript
- **DOMPurify**: HTML sanitization library requiring a DOM implementation

To initialize the server, we first create an *"express"* application and define *"port 3000"* for the server. Then, we initialize a virtual DOM environment using JSDOM and create a DOMPurify instance within this environment. We also implemented a *"Middleware"* function to process the request/response before reaching the final route handler. This includes *"express.json()"* to parse JSON requests, and *"express.static"* to use files from the public directory. Finally, we have different routes for testing. There is a vulnerable reflection endpoint *"(/reflect)"* that accepts user input via query string. This is deliberately made vulnerable to insert raw user inputs into the HTML response directly. We use this endpoint to demonstrate successful XSS attacks. On the other hand, we have a protected reflection endpoint *"(/safe-reflect)"* that is similar to the vulnerable endpoint mentioned before, but sanitizes input using DOMPurify. We use this to demonstrate how DOMPurify prevents XSS attacks.

For testing CSP against various attacks, we have the CSP test Endpoint *"(/csp-test)"*. CSP HTTP header defines strict security rules for the browser to follow when loading and executing content:

- OWASP Content Security Policy Cheat Sheet: Recommends using 'self', nonces, and advises on blocking plugins (object-src 'none'), clickjacking (frame-ancestors 'none'), and form/data exfiltration (form-action 'self')[5].
- MDN Web Docs - Content Security Policy: Recommends using report-uri/report-to for monitoring violations and details how to use nonces, restrict sources for scripts, styles, images, and other resources [4].
- Trusted Types : Guidance on using Trusted Types via CSP (require-trusted-types-for 'script' and trusted-types directives) [6].
- Strict CSP: Provides basic properties of strict policy to include, such as object-src 'none', script-src nonce, and script-src 'strict-dynamic' [1].

Some of the CSP Directives Detail:

- **default-src**: 'self' restricts resources to the same origin
- **script-src**: with nonce and strict-dynamic ensures only authorized scripts run
- **style-src**: with nonce controls stylesheet loading
- **object-src**: 'none' blocks plugin content
- **base-uri**: 'none' prevents base tag hijacking
- **require-trusted-types-for**: 'script' enforces Trusted Types API use
- **trusted-types**: defines allowed policies

```
const cspDirectives = [
  "default-src 'self'",
  `script-src 'nonce-${nonce}' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'`,
  `style-src 'nonce-${nonce}'`,
  "img-src 'self'",
  "font-src 'self'",
  "object-src 'none'",
  "base-uri 'none'",
  "connect-src 'self'",
  "frame-ancestors 'self'",
  "form-action 'self'",
  "manifest-src 'self'",
  "require-trusted-types-for 'script'",
  "trusted-types dompurify test-policy",
  "report-uri http://localhost:8888/collect-violations"
];
```

Figure 1. The CSP Header in file *"server.js"*.

In summary, we have implemented input sanitization (DOMPurify) to clean user input before it enters the application, remove known malicious patterns, and prevent both stored and reflected XSS. Moreover, Content Security Policy (CSP) controls what resources can load and execute, uses cryptographic nonces to validate legitimate scripts, and implements Trusted Types for DOM manipulation security [7].

3.2 CSP-bypass-test Implementation

The implementation is found in file *"csp-test.js"*, and the core architecture uses the Chromium browser tool from Playwright, which enables programmatic control of a Chromium browser instance to test web applications. We used *"Node.js"* built-in *"http"* module to create a violation report collection server.

For CSP violation reporting, we created a dedicated HTTP server on port *"8888"* specifically to receive and process CSP violation reports. The server listens for POST requests to the *"(/collect-violations)"* endpoint (specified in CSP Header-Uri), assembles the JSON request body, parses it, and stores it in the *"(violationReports)"* array.

For initial CSP inspection, we navigate to the test page at *"(http://localhost:3000/csp-test)"*. We created a CSP test page to ensure that a website's CSP is correctly configured and effectively blocks various security threats. We have implemented 10 different test categories, each targeting different aspects of CSP protection.

3.2.1 Basic Script injection tests. We used *"(page.evaluate())"* to run JavaScript directly within the browser context. For testing *"innerHTML"* script injection, we injected a "(< *script* >)" tag directly into the page's HTML using *"innerHTML"*. If the browser executes the injected script *"(console.log("CSP Bypass!"))"*, then the site is vulnerable to XSS. We also attempted to create a "(< *script* >)" element dynamically and inject it into the page. If the script executes, the site is vulnerable to XSS via dynamic script injection.

3.2.2 JSONP Bypass Testing. We test whether the CSP prevents loading scripts from unauthorized external domains. We first create a script element in the DOM. Then, the script's src points to a JSONP endpoint on *"evil-site.example"*, with a callback parameter set to *"alert("CSP Bypass!")"*. The script is added to the page and forces the browser to load and execute it. These attacks are dangerous as attackers could steal cookies/session data, redirect users to phishing pages, and perform other malicious actions.

3.2.3 Iframe Injection Testing. We test whether the CSP prevents injecting JavaScript via the *"srcdoc"* attribute of an iframe. We first create a malicious iframe, inject executable code using *"srcdoc"*, and then finally append it to the page.

3.3 Trusted types-bypass-test

Trusted Types is a browser security API designed to prevent DOM-based XSS by requiring that potentially dangerous DOM operations only accept special *"trusted"* objects rather than raw strings [2]. These trusted objects can only be created through explicitly defined *"policies"* which act as security gateways, sanitizing or validating input before creating these trusted objects. We used the same configuration and architecture created for *"CSP-bypass-test "*to further test for the Trusted Types.

3.3.1 Trusted Types innerHTML. We test whether Trusted Types properly blocks unsafe HTML injection via innerHTML. We create a container element, perform script injection, and finally append it to the page. To be more precise, we test whether *"(require-trusted-types-for 'script')"* directive properly protects against setting dangerous HTML content.

3.3.2 Script Source Manipulation. We test whether Trusted Types prevents setting arbitrary URLs as script sources. We first create a script tag, set a dangerous src using unsafe data URI *"(data:text/javascript,...)"*, and inject it into the page. We attempt to use a data URL as a script source, which would execute JavaScript directly.

3.3.3 Document.write Testing. We test whether *"document.write()"* is protected by Trusted Types. We use *"document.write()"* to inject script content.

3.3.4 DOM Clobbering Attack. We test whether Trusted Types can be bypassed via DOM clobbering attacks. We first create clobbering elements: create a new "(< *div* >)" element and set its HTML to contain a "(< *form* >)" element with *"id="trustedTypes""* and within it, a "(< *input* >)" element with *"name="createHTML""*. When this HTML is rendered, the browser creates two critical objects in the DOM: first, a form element accessible via *"window.trustedTypes"* (because of its ID) and second, an input element accessible via *"window.trustedTypes.createHTML"* (because of its name and parent relationship). This cleverly mimics the structure of the legitimate Trusted Types API, which normally has a *"window.trustedTypes"* object with a *"createHTML()"* method.

Now the attack tries to use the clobbered trustedTypes object as if it were the real Trusted Types API by creating a new div element and setting its innerHTML using what it thinks is the *"trustedTypes.createHTML()"* method. The real *"trustedTypes.createHTML()"* is a security function that sanitizes or rejects dangerous HTML. However, the clobbered version is an input form element, which behaves differently in this context.

3.3.5 Creating Disallowed Policy. We test whether the browser accurately restricts the creation of unauthorized security policies. First, we need to check if the browser supports the Trusted Types API. Next, we created a policy named *"evil-policy"*, which is not included in the site's CSP trusted-types directive. The evil policy is dangerous because its *"createHTML"* method simply returns the input string without any sanitization or validation.

3.3.6 DOM Manipulation. We perform a systematic test of multiple DOM manipulation methods to evaluate whether the Trusted Types security protection properly covers them:

- We first tested *"iframe.srcdoc"* attack. Trusted Types should prevent setting raw HTML strings to *"srcdoc"* without using a proper *"TrustedHTML"* object.
- For the second test, we tested *"Element.insertAdjacentHTML"* attack. This attack is similar to innerHTML.
- Finally, we tested *"Range.createContextualFragment"* attack, which creates a *"DocumentFragment"* from an HTML string using the *"Range API"*. The test first creates the fragment, then appends it to the document.

3.4 DOMPurify-bypass-test Implementation

3.4.1 What is DOMPurify? DOMPurify is a DOM-based sanitizer that protects against cross-site scripting (XSS) attacks in HTML, MathML, and SVG content. DOM stands for Document Object Model, and it represents an HTML document as a hierarchy of nodes. The entire page is a *"Document node"*, each HTML tag is an *"Element node"* such as *" "*, *"<a>"*, and *"<p>"*. All attributes of an element are *"Attribute nodes"*, text between tags is *"Text nodes"*, and comments are represented as *"Comment nodes"*. DOMPurify works by creating a DOM tree for the provided HTML. It then traverses every node in the DOM tree, removing all elements and attributes not on a whitelist, such as *"<script>"*, *"<iframe>"*, or event handlers like *"onerror"*. After DOMPurify has stripped the input HTML of what it identifies as potential cross-site scripting attacks, it serializes and returns the remaining input into hopefully safe HTML.

3.4.2 Implementation. The implementation is found in file *"DOMPurify-test.js"*, and like the CSP-bypass-test implementation, our DOMPurify tests implementation also use

"Playwright" and *"Node.js"*. The file defines a list of XSS payloads, and creates a Chromium instance with *"Playwright"*, and set arguments such as *"–disable-xss-auditor'"*, and *"–no-sandbox"*, to disable built-in, default protections for testing. Therefore, any similarities or differences between the reflected endpoint with no protection and the DOMPurify endpoint will be due to DOMPurify alone.

For both the reflected and DOMPurify endpoints, we run the array of payloads with XSS attacks. For each endpoint and XSS payload, we create a new Chromium instance, send the payload through a user input field, and then check whether any XSS was executed.

3.4.3 Basic XSS. For basic XSS attacks, we run payloads such as:

- <script>alert("XSS")</script>,
- <img src="x" onerror="alert('XSS')">,
- Hover me

For each XSS payload, we perform automated interactions such as hover and clicks, and then check if any *"window.alert"* has been executed. We also check for DOM attacks that explicitly call *"document.write"*.

3.4.4 Limitations with our implementation. Although we had prepared a broad range of payloads, meant to test how and if DOMPurify protects against, *"Basic XSS"*, *"CSP bypass attempts"*, *"DOM-based XSS"*, *"DOM Clobbering attacks"*, *"SVG-based attacks"*, *"Obfuscated attacks"*, *"Video source attacks"*, and *"Dangling markup injections"*. During the later stages of development, it became apparent that our testing environment had limitations due to only accurately testing for whether any *"window.alert"* has been executed or DOM attacks that explicitly call *"document.write"*. Effectively, meaning that our DOMPurify testing environment is only testing if DOMPurify removes alerts from the input, regardless of how complex the input is. Consequently, our DOMPurify testing environment implementation cannot provide accurate results regarding how DOMPurify protects against more complex payloads that don't contain basic XSS elements. We therefore decided to include an overview of the paper of Heiderich et al. [3], Heiderich is the creator of DOMPurify.

3.4.5 Overview of DOMPurify: Client-Side Protection Against XSS and Markup Injection. The goal of DOMPurify was to primarily offer client-sided security against all currently known XSS payloads, while being robust against DOM Clobbering [3]. To do this, DOMPurify *"comprises two main components: the DOM Clobbering Detection (DCD)"* and the *"HTML Sanitizer"* [3]. First, the HTML sanitizer is based on a whitelist of 206 different elements and 295 attributes, all of which are deemed safe by DOMPurify. According to the authors of [3], *"the majority of XSS filters actually behave in a too strict manner and remove too many benign elements"*; thus, DOMPurify's 206 elements and 295 attributes offer a reduction in false positive sanitization while still ensuring safety, by discarding any element or attribute not on the whitelist. Furthermore, the HTML sanitizer parses and processes attributes and elements *"in reverse order of appearance in the parent element or container"*, since this protects against some re-indexing and mutation-based XSS attacks.

Secondly, the DCD protects both the DOMPurify library itself and the host page's DOM before sanitizing the input. DOMPurify verifies that all its functions, such as *"removeChild"* or *"NodeIterator"*, are free from malicious interference, and *"if one DOM feature appears to have been tampered with, DOMPurify will abort immediately and return an empty string instead of potentially unsanitized content"* [3]. After this, while sanitizing the inert DOM that it builds from the untrusted markup, DOMPurify inspects every node, and *"for any element applied with an ID or NAME attribute, aiming at avoiding global and document clobbering. If an element in the untrusted HTML string may have clobbering effects on the surrounding document, then it will be removed"*. Eliminating any element whose id or name would overwrite existing window or document properties, while also ensuring that the DOMPurify library itself is secure, means that DOMPurify has strong protection against DOM clobbering attacks. We review the results of the paper in section 4.1.1.

3.5 How to run tests

You can run the tests by first starting the server and then running each test script. First, navigate to the parent folder using:

"cd xss-defense-research"

Then build the Docker image with:

"docker build -t xss-test-app ."

Next, run the Docker container using:

"docker run -p 3000:3000 xss-test-app"

The server should now be up and running, displaying the message "XSS test server running at http://localhost:3000". To run each test, go to the research folder with:

"cd xss-defense-research/research"

Then, run each test type the following two commands:

"node csp-test.js"
"node DOMPurify-test.js"

4 Results

The full test results for CSP and Trusted Types are displayed in the Appendix section 5.3. We did not include the test results of DOMPurify-bypass-test due to its limitations; for information about the limitations, see section 3.4.4.

4.1 DOMPurify

Because we aim to provide a deeper understanding and offer practical guidance on effective XSS defenses in modern

web applications, even though our DOMPurify testing environment is currently limited (see section 3.4.4), we instead reviewed (see section 3.4.5) the paper of Heiderich et al. [3], Heiderich is the primary creator of DOMPurify.

4.1.1 Results of DOMPurify: Client-Side Protection Against XSS and Markup Injection.

The authors of [3] evaluated DOMPurify against *"400 + attack vectors"* collected from the *"HTML5 Security Cheatsheet"*, the *"OWASP XSS Cheat Sheet"*, and from a *"large number of novel attack vectors by the security community"*. They designed DOMPurify to mitigate all of these attacks and concluded that *"in the current version of DOMPurify, there are neither undetected XSS vectors nor bypasses to DOMPurify"* [3].

For performance testing, the authors of [3] processed and sanitized 1413 real-world newsletters. They found that the *"average processing time of the 1413 emails is 54.7 ms" 80% of the emails are processed in ≤ 89 ms"*. They also observed that there were very few false positives, as *"more than 80% of the test set shows no visual differences (0%) between the original and the processed email"* and reported differences in the other samples were due to minor text shifts by a few pixels [3].

The paper [3] demonstrates that DOMPurify delivers robust client-side sanitization, protecting against a vast array of known XSS vectors with low processing overhead and false positives, making it a potentially vital component for an XSS-safe modern web application. Furthermore, the DOMPurify library is easy to deploy, as it requires just one function call for input: *"DOMPurify.sanitize(input)"* to clean potentially unsafe user input and prevent XSS attacks, making it a reliable and convenient choice for most web applications.

4.2 CSP and Trusted Types

The test results demonstrate a highly effective implementation of Content Security Policy (CSP) and Trusted Types to prevent Cross-Site Scripting (XSS) attacks. The configured security measures successfully blocked all attempted XSS injection vectors, with the CSP violation reporting mechanism properly capturing and documenting each security event.

1. Basic Script Injection for both CSP and Trusted Types tests were blocked with appropriate error messages about requiring TrustedHTML/TrustedScript
2. JSONP Bypass attempts were prevented, with the system correctly requiring TrustedScriptURL for script sources
3. Iframe Injection was blocked, requiring TrustedHTML
4. DOM Clobbering to override the TrustedTypes object was successfully prevented
5. Unauthorized Trusted Types policy named *"evil-policy"* was blocked, showing that the policy allowlist is properly enforced
6. Lesser-known DOM manipulation methods like insertAdjacentHTML and createContextualFragment were also protected

The test results indicate an exemplary implementation of modern XSS defenses. The combination of CSP and Trusted Types creates multiple layers of protection that successfully mitigate even sophisticated attack techniques. From our research we also found that:

- CSP alone remains vulnerable to DOM clobbering and HTML manipulation attacks
- Strong CSP with nonce-based script-src effectively blocks most traditional XSS
- CSP + Trusted Types together blocked all of test vectors
- Adding DOMPurify provided comprehensive protection against all tested attacks

This implementation represents a best-practice approach to preventing XSS vulnerabilities. It demonstrates that when properly configured, modern browser security features can effectively block even advanced web application attacks.

References

[1] Google Security Team. 2025. Strict CSP. https://csp.withgoogle.com/docs/strict-csp.html. Accessed May 2025.

[2] Google Web Fundamentals. 2025. Trusted Types. https://web.dev/trusted-types/. Accessed May 2025.

[3] Mario Heiderich, Christopher Späth, and Jörg Schwenk. 2017. DOMPurify: Client-Side Protection Against XSS and Markup Injection. 116–134. doi:10.1007/978-3-319-66399-9_7

[4] Mozilla Developer Network. 2025. Content Security Policy (CSP). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. Accessed May 2025.

[5] OWASP. 2025. Content Security Policy Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html. Accessed May 2025.

[6] Felix Ryan. 2025. Trusted Types Checker. https://portswigger.net/bappstore/1894edb751244e52856efa092d58979d. Accessed May 2025.

[7] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. 2016. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*. Vienna, Austria.

5 Appendix

5.1 Contributions

Oscar worked on developing DOMPurify tests, while Woojin worked on setting up the testing environment and CSP and Trusted Types tests. Both teammates have coordinated on the project and talked about and checked each other's code through internal meetings. **Note** that the GitHub log is not very representative of the workflow, since we changed the project repository, meaning the first commit is massive.

5.2 GitHub repository

GitHub repository

5.3 Sample CSP and Trusted Types Test Result

===== Running CSP Bypass Tests =====

1. Testing script injection in different contexts...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri 'none'; connect-src 'self'; frame-ancestors 'self'; form-action 'self'; manifest-src 'self'; require-trusted-types-for 'script'; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 6,
"column-number": 33,
"status-code": 200,
"script-sample": "Element innerHTML|<h1>Content Security Policy Test Page</h"

CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri 'none'; connect-src 'self'; frame-ancestors 'self'; form-action 'self'; manifest-src 'self'; require-trusted-types-for 'script'; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 15,
"column-number": 28,
"status-code": 200,
"script-sample": "HTMLScriptElement textContent|console.log(C̈SP Bypass!ÿ)"

Protected: innerHTML script blocked - Failed to set the 'innerHTML' property on 'Element': This document requires 'TrustedHTML' assignment.

Protected: script element blocked - Failed to set the 'textContent' property on 'Node': This document requires 'TrustedScript' assignment.

2. Testing JSONP/external script bypass...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri

’none’; connect-src ’self’; frame-ancestors ’self’; form-action ’self’; manifest-src ’self’; require-trusted-types-for ’script’; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 4,
"column-number": 20,
"status-code": 200,
"script-sample": "HTMLScriptElement src|https://evil-site.example/jsonp?callback"

Protected: JSONP injection blocked - Failed to set the ’src’ property on ’HTMLScriptElement’: This document requires ’TrustedScriptURL’ assignment.

3. Testing iframe injection...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src ’self’; script-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’ https://cdnjs.cloudflare.com/ajax/libs/ ’strict-dynamic’; style-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’; img-src ’self’; font-src ’self’; object-src ’none’; base-uri ’none’; connect-src ’self’; frame-ancestors ’self’; form-action ’self’; manifest-src ’self’; require-trusted-types-for ’script’; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 4,
"column-number": 23,
"status-code": 200,
"script-sample": "HTMLIFrameElement srcdoc|<script>parent.console.log(C̈SP Bypass!¨

Protected: iframe srcdoc injection blocked - Failed to set the ’srcdoc’ property on ’HTMLIFrameElement’: This document requires ’TrustedHTML’ assignment.

===== Testing Trusted Types Bypass Attempts =====

1. Testing Trusted Types with innerHTML script injection...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src ’self’; script-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’ https://cdnjs.cloudflare.com/ajax/libs/ ’strict-dynamic’; style-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’; img-src ’self’; font-src ’self’; object-src ’none’; base-uri ’none’; connect-src ’self’; frame-ancestors ’self’; form-action ’self’; manifest-src ’self’; require-trusted-types-for ’script’; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 4,
"column-number": 23,
"status-code": 200,
"script-sample": "Element innerHTML|<script>console.log(T̈rusted Types bypas"

Trusted Types properly blocked innerHTML script injection: Failed to set the ’innerHTML’ property on ’Element’: This

document requires 'TrustedHTML' assignment.

2. Testing Trusted Types with script.src manipulation...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri 'none'; connect-src 'self'; frame-ancestors 'self'; form-action 'self'; manifest-src 'self'; require-trusted-types-for 'script'; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 4,
"column-number": 20,
"status-code": 200,
"script-sample": "HTMLScriptElement src|data:text/javascript,console.log(T̈ruste"

Trusted Types properly blocked script.src manipulation: Failed to set the 'src' property on 'HTMLScriptElement': This document requires 'TrustedScriptURL' assignment....

3. Testing Trusted Types with document.write...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri 'none'; connect-src 'self'; frame-ancestors 'self'; form-action 'self'; manifest-src 'self'; require-trusted-types-for 'script'; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 3,
"column-number": 18,
"status-code": 200,
"script-sample": "Document write|<script>console.log(T̈rusted Types bypas"

Trusted Types properly blocked document.write: Failed to execute 'write' on 'Document': This document requires 'Trusted-HTML' assignment.

4. Testing Trusted Types with DOM clobbering attempt...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri

’none’; connect-src ’self’; frame-ancestors ’self’; form-action ’self’; manifest-src ’self’; require-trusted-types-for ’script’; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 5,
"column-number": 23,
"status-code": 200,
"script-sample": "Element innerHTML|<form id=ẗrustedTypes̈><input name=c̈re"

Trusted Types is not vulnerable to DOM clobbering: Failed to set the ’innerHTML’ property on ’Element’: This document requires ’TrustedHTML’ assignment.

5. Testing Trusted Types policy creation restrictions...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "trusted-types",
"effective-directive": "trusted-types",
"original-policy": "default-src ’self’; script-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’ https://cdnjs.cloudflare.com/ajax/libs/ ’strict-dynamic’; style-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’; img-src ’self’; font-src ’self’; object-src ’none’; base-uri ’none’; connect-src ’self’; frame-ancestors ’self’; form-action ’self’; manifest-src ’self’; require-trusted-types-for ’script’; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-policy",
"line-number": 4,
"column-number": 39,
"status-code": 200,
"script-sample": "evil-policy"

Properly blocked unauthorized policy creation: Failed to execute ’createPolicy’ on ’TrustedTypePolicyFactory’: Policy "evil-policy" disallowed.

6. Testing Trusted Types coverage of DOM manipulation methods...
CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src ’self’; script-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’ https://cdnjs.cloudflare.com/ajax/libs/ ’strict-dynamic’; style-src ’nonce-BXIuePMaOUhEShwIuuVV/A==’; img-src ’self’; font-src ’self’; object-src ’none’; base-uri ’none’; connect-src ’self’; frame-ancestors ’self’; form-action ’self’; manifest-src ’self’; require-trusted-types-for ’script’; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 6,
"column-number": 25,
"status-code": 200,
"script-sample": "HTMLIFrameElement srcdoc|<script>parent.console.log(T̈rusted Type"

CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",

"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri 'none'; connect-src 'self'; frame-ancestors 'self'; form-action 'self'; manifest-src 'self'; require-trusted-types-for 'script'; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 12,
"column-number": 15,
"status-code": 200,
"script-sample": "Element insertAdjacentHTML|<script>console.log(T̈rusted Types bypas"

CSP Violation Captured:
"csp-report":
"document-uri": "http://localhost:3000/csp-test",
"referrer": "",
"violated-directive": "require-trusted-types-for",
"effective-directive": "require-trusted-types-for",
"original-policy": "default-src 'self'; script-src 'nonce-BXIuePMaOUhEShwIuuVV/A==' https://cdnjs.cloudflare.com/ajax/libs/ 'strict-dynamic'; style-src 'nonce-BXIuePMaOUhEShwIuuVV/A=='; img-src 'self'; font-src 'self'; object-src 'none'; base-uri 'none'; connect-src 'self'; frame-ancestors 'self'; form-action 'self'; manifest-src 'self'; require-trusted-types-for 'script'; trusted-types dompurify test-policy; report-uri http://localhost:8888/collect-violations",
"disposition": "enforce",
"blocked-uri": "trusted-types-sink",
"line-number": 18,
"column-number": 34,
"status-code": 200,
"script-sample": "Range createContextualFragment|<script>console.log(T̈rusted Types bypas"

iframe.srcdoc properly protected by Trusted Types: Failed to set the 'srcdoc' property on 'HTMLIFrameElement': This document requires 'TrustedHTML' assignment.

Element.insertAdjacentHTML properly protected by Trusted Types: Failed to execute 'insertAdjacentHTML' on 'Element': This document requires 'TrustedHTML' assignment.

Range.createContextualFragment properly protected by Trusted Types: Failed to execute 'createContextualFragment' on 'Range': This document requires 'TrustedHTML' assignment.

===== CSP Violation Report Summary =====
Total violations detected: 12

Violation types:
- require-trusted-types-for: 11 violations
- trusted-types: 1 violations

===== SECURITY ASSESSMENT =====
Trusted Types are properly enforced - all attempts were blocked
Multiple CSP directives are active and enforcing restrictions
Test completed and servers shut down