Comenius university in Bratislava
Faculkty of mathemmatics, physics and computer science

Trusted Types and bundlers

(School project for IT security)

2020
Emanuel Tesař

Abstract

Keywords:

Contents

Chapter 1

Cross site scripting

Cross site scripting (abbr. XSS) is one of the most prevalent security vulnerabilities and the most common one when talking about web applications. When we take into the account only the last year (year 2020), XSS has been ranked on 7th place in the OWASP top 10 vulnerabilities ranking. [3]. When, we look at the bounty programs only, and the money rewarded for each security vulnerability, we can see that XSS is the winner. Total bounty just for XSS was more than $4,2$ million USD [9].

Citing very well written introduction from [19].

> XSS attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script (for example encoded in URL), to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it [19].
>
> An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page [19].
>
> *(Be sure to read the cited article for more information and links [19]. The above paragraphs were also largely copied from that very well written article.)*

1.1 Types of XSS

There are many types of XSS, although there is no single finite list of XSS types to rule them all [20]. Most experts distinguish at least between non-persistent *(reflected)* and persistent *(stored)*. There is also a third category *DOM based XSS* which will be explained in more depth as this is the threat model under which Trusted Types operate.

- Stored - The injected script is permanently saved in server database. The client *(user's browser)* will then ask the server for the requested page and the response from the server will contain the malicious script.

- Reflected - Typically delivered via email or a neutral web site. occur when a malicious script is reflected off of a web application to the victim's browser [7].

- DOM based - The vulnerability appears in the DOM - by executing some malicious code. In reflective and stored Cross-site scripting attacks you can see the vulnerability payload in the response page but in DOM based cross-site scripting, the HTML source code and response of the attack will be exactly the same. You can only observe the change at runtime.

1.2 Modern classification of XSS

The classification in the *Section 1.1 - Types of XSS* was created many years back and a lot has since changed. The web got more secure and modern frameworks try to enforce best security practices for developers using them. However, web has since evolved rapidly *and still evolves* while still mostly preserving the backward compatibility with original JS spec - meaning you could still browse the web page created 20 years ago with subtle differences *(Compare that with running Android app created for version 4.1 Jelly Bean created in mid 2012 on Android 11 released 2020 - good luck with that)*.

The previous classification is not ideal, because the categories overlap. Citing from [13]:

> You can have both Stored and Reflected DOM Based XSS. You can also have Stored and Reflected Non-DOM Based XSS too, but that's confusing, so to help clarify things, starting about mid 2012, the research community proposed and started using two new terms to help organize the types of XSS that can occur:

Instead what they propose is just two categories (again, fully citing [13]):

- Server XSS - Server XSS occurs when untrusted user supplied data is included in an HTTP response generated by the server. The source of this data could be from the request, or from a stored location. As such, you can have both Reflected Server XSS and Stored Server XSS. In this case, the entire vulnerability is in server-side code, and the browser is simply rendering the response and executing any valid script embedded in it.

- Client XSS - Client XSS occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to introduce valid JavaScript into the DOM. This source of this data could be from the DOM, or it could have been sent by the server (via an AJAX call, or a page load). The ultimate source of the data could have been from a request, or from a stored location on the client or the server. As such, you can have both Reflected Client XSS and Stored Client XSS.

Just for completeness, DOM based XSS is a subset of client XSS. The source of the data is client side only. And again, study the full article *(together with further references)* for more information [13].

1.3 Common examples of XSS

The basic example of XSS *and probably the easiest to understand* is to take a page which interpolates data from an URL. This is a common practice - you browse a site, find something of value and you want to share it with your friend. Nothing easier, you just copy the URL and they see the same content *or at least similar* to you. This basic example is common for nearly all shopping sites, tourism agencies, accommodation services etc...

This type of attack can be easily demonstrated with the following example page:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>URL XSS</title>
  </head>
  <body>
    <p>Try searching something:
      <i> (for example "<img src=x onerror=alert(1) />")</i>
    </p>
    <input id="query" type="text" />
```

```
    <button id="submit">Search</button>

    <p>You have searched for</p>
    <p id="attack-target"></p>
  </body>
  <script>
    window.addEventListener('DOMContentLoaded', () => {
      const urlParams = new URLSearchParams(location.search)
      document.getElementById('attack-target').innerHTML =
          urlParams.get('query')
    });

    document
      .getElementById('submit')
      .addEventListener('click', () => {
        const query = document.getElementById('query').value
        location.replace(
          `${location.pathname}?query=${encodeURIComponent(query)}`
        );
      })
  </script>
</html>
```

This is very small example and XSS is apparent, but once the project is composed of tens of thousands lines and many dependencies, such mistake can easily sneak through. In this case, if you execute this HTML code in the browser and you try searching for something, the result will be interpolated in the page. This allows the attacker to *prepare an evil URL* which they can then send to benign users to exploit.

For more examples, I recommend playing the *"XSS games"* [15] [16] [17] [18].

Finally, there are many cheatsheets with possible attack payloads and polyglots *Code that works for various environments, such as HTML, JS, CSS*. For example: [14].

1.4 Consequences of XSS

So far, we have talked about many types of XSS, saw a basic example and are aware that there are many more. What we haven't covered are the consequences, which can be caused by these attacks.

Citing cypress data defense article about XSS [2]:

> The impact of cross-site scripting vulnerabilities can vary from one web

application to another. It ranges from session hijacking to credential theft and other security vulnerabilities. By exploiting a cross-site scripting vulnerability, an attacker can impersonate a legitimate user and take over their account.

If the victim user has administrative privileges, it might lead to severe damage such as modifications in code or databases to further weaken the security of the web application, depending on the rights of the account and the web application.

Apart from these basic consequences, there are dozens of others. Citing [19]:

> Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content. An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company's stock price or lessen consumer confidence. An XSS vulnerability on a pharmaceutical site could allow an attacker to modify dosage information resulting in an overdose.

Also, sites might have more security vulnerabilities and even a benign XSS vulnerability might be used with combination with a different attack vector *e.g. CSRF* which has often more severe consequences.

1.5 Protection

There are no definitive measures to protect against XSS, however there are guidelines. XSS is caused by interpolating untrusted data into otherwise trusted environments. It is important to educate developers about the possible attack vectors and vulnerabilities in the underlying platforms *e.g. DOM and possible attack vectors with HTML/CSS.*

Applications, which need to interpolate uncontrolled user data into their applications need to make sure the input is safe. In pratice, this means **encoding** or **sanitizing** the inputs. It is also important to follow the new modern APIs which aim to prevent XSS - these include various security headers *Use Security Headers in [2]* or modern APIs such as Safe-Types [8] and Trusted Types [5]. The latter will be described in the rest of the paper.

Chapter 2

Introduction to Trusted Types

In this chapter we are going to introduce the concept of Trusted Types (abbreviated as TT). The threat model under which TT operate is DOM XSS *(subset of client side XSS)*. This vulnerability happens when untrusted, user controlled input reaches a *sink* which is function like *eval()* or *innerHTML* property of an HTML element.

The idea is to take all of the sinks and make them secure by default. For example, in DOM you can assign any string value to *innerHTML* property of an element and the browser will parse the string using HTML parser and render the parsed markup. Instead, we make this property accept only ***special branded objects*** and throw an error on any other value *e.g. string*.

This is a breaking change behaviour in DOM spec and is unfortunately not acceptable to change the DOM spec in this way. Instead, users who want to use TT have to explicitely opt-in to enforce this API. As of now, TT are supported in chromium based browser, but it is likely the support will increase [10].

Developers are asked to avoid the sinks if possible to reduce the attack surface, however, sometimes there is no way to avoid the sink. For such cases there should be a way for developers to create these *special branded objects* and make them explicitely **trusted** by the DOM APIs and sinks - hence the name *Trusted Types*. Of course, if creating TT would be as easy as calling wrapping the value in a function call, developers would quickly misuse this behaviour and just wrap the unsafe values in that function call instead of making sure the value can be trusted.

Refer to the recommended explainer article for more info [5] or the formal spec [12].

2.1 Enforcing Trusted Types

As mentioned, TT is backward incompatible change and must be enforced explicitely by the developers. You can enable TT by adding the following response header to the documents which should operate under TT enforced.

Listing 2.1: Restricting policy names

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

2.2 Creating Trusted Types

Most of the times, you can avoid using the sinks alltogether and use safe DOM APIs instead *e.g. create DOM node dynamically instead of innerHTML assignment*. However there are cases when using sinks can't be avoided, or the values are already sanitized/encoded. You can create trusted type which will be accepted by the DOM sink.

If your browser supports TT, you will have access to *trustedTypes* global object on *window* instance. You can use *trustedTypes.createPolicy* function to create a policy, which acts like a "factory" for creating the trusted values. For example, citing [5]:

Listing 2.2: Creating Trusted Types policy

```
if (window.trustedTypes && trustedTypes.createPolicy) { // Feature testing
const escapeHTMLPolicy = trustedTypes.createPolicy('myEscapePolicy', {
  createHTML: string => string.replace(/\</g, '&lt;')
});
}
```

and then you would use the policy simply by *(again citing [5])*:

Listing 2.3: Using the policy to create Trusted value

```
const escaped = escapeHTMLPolicy.createHTML('<img src=x onerror=alert(1)>');
console.log(escaped instanceof TrustedHTML); // true
el.innerHTML = escaped; // '&lt;img src=x onerror=alert(1)>'
```

However, as you noticed the policies are named. This is because you can restrict which policies might be created by your application. All you need to do is provide additional response header with a whitelist of allowed policy names.

Listing 2.4: Restricting policy names

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

2.3 Report only mode

Before switching to enforcing mode, which will cause an error when untrusted value is assigned to a sink, you can switch to report only mode, which will preserve the original DOM API behaviour *(no error will be thrown and standard sink behaviour will be used)*. However, there will be a warning in the console and CSP violation will be triggered. [11].

You will also have to use slightly different CSP response header [5]:

Listing 2.5: Report only CSP

```
Content-Security-Policy-Report-Only: require-trusted-types-for 'script';
    report-uri //my-csp-endpoint.example
```

2.4 Default policy

Creating your own policies and making sure *trusted* values reach the sinks is not always possible. Application code is largely composed of many dependencies and third party code, which the developer can't easily modify. For such cases, there is special ***default policy*** which will be called when untrusted value reaches is sink. It is the last place where the developer *(specifically, the handler function the developer implemented)* might want *bless* the value such that it will be accepted by the DOM API. [5]

Listing 2.6: Creating default policy

```
if (window.trustedTypes && trustedTypes.createPolicy) { // Feature testing
  trustedTypes.createPolicy('default', {
    createHTML: (string, sink) => DOMPurify.sanitize(string,
        {RETURN_TRUSTED_TYPE: true})
  });
}
```

Note, that *default policy* is still a named policy and must be listed in the allowlist specified by *trusted-types* response header *(of course, this only applies if the application uses this header)*.

2.5 Trusted Types integration

TT is being developed and used internally at Google, where they integrate it into their core products. However, many open source projects see the benefit of TT and started doing necessary steps needed for TT integration. The amount of work needed to integrate TT to application varies a lot. There are many factors which might contribute to this complexity:

- Bundlers - This is more a developer nuisance than production issue. Main purpose for bundlers is to transpile the code using latest *(and not widely supported)* features to standard version of JavaScript that all browsers understand. Nowdays, bundlers also take care of serving the app in the development phase and providing features such as code reloading *which refresh the web page once developer code is saved* or showing a popup with error message and stack trace. These development features often cause violations when TT are enforced.

- DOM framework - Modern web apps usually use some kind of framework *(examples include: React, Angular, Polymer, Vue, Svelte...)*. These frameworks often provide fine security measures, but some frameworks may be more *"TT friendly"* then others. For example, making React or Polymer TT compliant is much easier than integration with Angular [6] [4] [1].

- Other third party dependencies - Many third party dependencies might use sinks under the hood and fixing such violations might be hard.

- Application code - These are the violations in the source code of the developer, these are usually the easiest to fix. You either avoid using the sink or wrap the value in a policy.

Bibliography

[1] Angular integration. https://github.com/angular/angular/search?&q=trusted+types&type=commits.

[2] The impact of cross-site scripting vulnerabilities and their prevention. https://www.cypressdatadefense.com/blog/cross-site-scripting-vulnerability/#:~:text=Cross-site%20scripting%20(XSS),information%20(PII)%2C%20social%20security&text=Cross-site%20scripting%20(XSS),information%20(PII)%2C%20social%20security.

[3] Owasp top 10 vulnerabilities. https://snyk.io/learn/owasp-top-10-vulnerabilities/.

[4] Polymer integration. https://github.com/lit/lit/commit/2a719e6746fc0b28a714454be051808d7bf3f31d.

[5] Prevent dom-based cross-site scripting vulnerabilities with trusted types. https://web.dev/trusted-types/.

[6] React integration. https://github.com/facebook/react/pull/16157.

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

[8] The security reviewer's guide to safe html apis and strict templating. https://github.com/google/safe-html-types/blob/master/doc/security_reviewers_guide_safehtml.md.

[9] Top 10 most impactful and rewarded vulnerability types. https://snyk.io/learn/owasp-top-10-vulnerabilities/.

[10] Trusted types browser compatibility. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/trusted-types#browser_compatibility.

[11] Trusted types introduction - csp violation section. `https://web.dev/trusted-types/#prepare-for-content-security-policy-violation-reports`.

[12] Trusted types specification. `https://w3c.github.io/webappsec-trusted-types/dist/spec/#webidl-integration`.

[13] Types of cross site scriping. `https://owasp.org/www-community/Types_of_Cross-Site_Scripting`.

[14] Xss filter evasion cheat sheet. `https://owasp.org/www-community/xss-filter-evasion-cheatsheet`.

[15] Xss game 1. `https://xss-game.appspot.com/`.

[16] Xss game 2. `https://alf.nu/alert1`.

[17] Xss game 3. `http://prompt.ml/0`.

[18] Xss game 4. `https://xss-quiz.int21h.jp/`.

[19] KirstenS. Introduction to cross site scripting. `https://owasp.org/www-community/attacks/xss/`.

[20] J. R. R. Tolkien. *The Lord of The Rings, The Fellowship of the Ring*. 1954.