Trusted Types and the end of DOM XSS

Krzysztof Kotowicz, Google

@kkotowicz
github.com/koto
koto@google.com
Google Vulnerability Reward Program payouts in 2018

Non-web issues 49.1%
Mobile app vulnerabilities
Business logic (authorization)
Server /network misconfigurations
...

XSS 35.6%
Vulnerabilities by industry

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Consumer Goods</th>
<th>Financial services &amp; insurance</th>
<th>Government</th>
<th>Healthcare</th>
<th>Media &amp; Entertainment</th>
<th>Professional services</th>
<th>Retail &amp; Ecommerce</th>
<th>Technology</th>
<th>Telecom</th>
<th>Transportation</th>
<th>Travel &amp; Hospitality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross Site scripting (XSS)</td>
<td>25%</td>
<td>24%</td>
<td>26%</td>
<td>13%</td>
<td>28%</td>
<td>27%</td>
<td>24%</td>
<td>21%</td>
<td>23%</td>
<td>24%</td>
<td>55%</td>
</tr>
<tr>
<td>Information disclosure</td>
<td>17%</td>
<td>16%</td>
<td>18%</td>
<td>6%</td>
<td>15%</td>
<td>14%</td>
<td>16%</td>
<td>17%</td>
<td>15%</td>
<td>3%</td>
<td>3%</td>
</tr>
<tr>
<td>Improper authentication</td>
<td>3%</td>
<td>4%</td>
<td>5%</td>
<td>6%</td>
<td>5%</td>
<td>11%</td>
<td>8%</td>
<td>8%</td>
<td>6%</td>
<td>15%</td>
<td>10%</td>
</tr>
<tr>
<td>Violation of secure design principles</td>
<td>6%</td>
<td>5%</td>
<td>10%</td>
<td>10%</td>
<td>12%</td>
<td>9%</td>
<td>0%</td>
<td>13%</td>
<td>6%</td>
<td>4%</td>
<td>2%</td>
</tr>
<tr>
<td>Cross-site request forgery (CSRF)</td>
<td>4%</td>
<td>10%</td>
<td>8%</td>
<td>7%</td>
<td>5%</td>
<td>12%</td>
<td>5%</td>
<td>8%</td>
<td>2%</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td>Open redirect</td>
<td>6%</td>
<td>4%</td>
<td>5%</td>
<td>5%</td>
<td>3%</td>
<td>3%</td>
<td>4%</td>
<td>3%</td>
<td>2%</td>
<td>4%</td>
<td>3%</td>
</tr>
<tr>
<td>Privilege Escalation</td>
<td>5%</td>
<td>4%</td>
<td>1%</td>
<td>5%</td>
<td>3%</td>
<td>3%</td>
<td>2%</td>
<td>4%</td>
<td>5%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>Improper access control</td>
<td>12%</td>
<td>5%</td>
<td>3%</td>
<td>6%</td>
<td>7%</td>
<td>9%</td>
<td>4%</td>
<td>6%</td>
<td>2%</td>
<td>4%</td>
<td>2%</td>
</tr>
<tr>
<td>Cryptographic issues</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td>2%</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>1%</td>
</tr>
<tr>
<td>Denial of service</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>2%</td>
<td>1%</td>
</tr>
<tr>
<td>Business logic errors</td>
<td>4%</td>
<td>1%</td>
<td>1%</td>
<td>4%</td>
<td>5%</td>
<td>6%</td>
<td>4%</td>
<td>4%</td>
<td>3%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>Code injection</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>5%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
</tr>
<tr>
<td>SQL injection</td>
<td>4%</td>
<td>1%</td>
<td>5%</td>
<td>4%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>0%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>Source: HackerOne report, 2018</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Listed are the top 15 vulnerability types platform wide, and the percentage of vulnerabilities received per industry.
Paid bounties by vulnerability on Mozilla websites in 2016 and 2017

Source: @jvehent, Mozilla
Why do we still have DOM XSS?

Easy to introduce & Hard to detect
DOM XSS is a client-side XSS variant caused by the DOM API not being secure by default.

- User controlled **strings** get converted into code
- via dangerous and error-prone **DOM sinks** like `innerHTML`

Example: https://example.com/#<img src=x onerror=alert('xss')>

```javascript
var foo = location.hash.slice(1);
document.querySelector('#foo').innerHTML = foo;
```
Element.innerHTML
Is this DOM XSS?

```javascript
foo.innerHTML = location.href
```
Is this DOM XSS?

```
foo.innerHTML = location.href
```

```
foo.innerHTML = bar
```
Is this DOM XSS?

```javascript
foo.innerHTML = location.href
```

```javascript
foo.innerHTML = bar
```

```javascript
foo.innerText = bar
```
Is this DOM XSS?

foo.innerHTML = location.href

foo.innerHTML = bar

foo.innerText = bar

foo[baz] = bar
Growing problem

- DOM sinks can be used by your own code
  - ... or the libraries you use
  - ... or the scripts you load (analytics?)
  - ... or the script that they load at runtime.

- Each of those potentially introduces DOM XSS

- Static analysis for JS does not work reliably

- At Google, DOM XSS is already the most common XSS variant.
To fix DOM XSS we need to reduce the attack surface and make security decisions explicit.
We know how to address it!

Safe Types (link)

- 6+ years of experience
- Protects Gmail and almost all other Google applications
- Evidence of efficacy
- Securely produce values that end up in DOM

- Implemented as Java{Script}, Go, … libraries
- We're porting this approach directly to the browsers
Trusted Types
The idea behind Trusted Types

Strongly typed DOM API.

- Don't pass (HTML, URL, script URL) **strings** to the DOM sinks
- Use **objects** instead
- DOM already supports it:

  ```javascript
  el.innerHTML = {toString: () => 'hello'};
  el.innerHTML // 'hello'
  ```

- Web platform (or polyfill) provides **typed objects**: **TrustedHTML, TrustedScript, TrustedScriptURL**
- Security rules & controls are based on types
The idea behind Trusted Types

When Trusted Types are not enforced:

DOM sinks accepts strings as usual

```javascript
element.innerHTML = location.hash.slice(1) // a string
```

DOM sinks accept typed objects

```javascript
element.innerHTML = aTrustedHTML
```
The idea behind Trusted Types

When Trusted Types are **enforced:**

```
Content-Security-Policy: trusted-types myPolicy
```

DOM sinks **reject** **strings**

```
element.innerHTML = location.hash.slice(1) // a string
```

▶️ Uncaught TypeError: Failed to set the 'innerHTML' property on 'Element': This document requires TrustedHTML assignment.
   at demo2.html:9

DOM sinks **accept** **typed objects**

```
element.innerHTML = aTrustedHTML
```
The idea behind Trusted Types

When Trusted Types are in reporting mode:

```plaintext
```

DOM sinks accept & report strings

```javascript
element.innerHTML = location.hash.slice(1) // a string
```

DOM sinks accept typed objects

```javascript
element.innerHTML = aTrustedHTML
```
Creating Trusted Types

1. **Create** policies with validation rules

```javascript
const SanitizingPolicy = trustedTypes.createPolicy('myPolicy', {
  createHTML(s: string) => myCustomSanitizer(s)
}, false);
```
Creating Trusted Types

1. **Create** policies with validation rules

```javascript
const SanitizingPolicy = trustedTypes.createPolicy('myPolicy', {
    createHTML(s: string) => myCustomSanitizer(s)
}, false);
```

2. **Use** the policies to create Trusted Type objects

```javascript
// Calls myCustomSanitizer(foo).
const trustedHTML = SanitizingPolicy.createHTML(foo);
const element.innerHTML = trustedHTML;
```
Creating Trusted Types

1. **Create** policies with validation rules

   ```javascript
   const SanitizingPolicy = trustedTypes.createPolicy('myPolicy', {
       createHTML(s: string) => myCustomSanitizer(s)
   }, false);
   ```

2. **Use** the policies to create Trusted Type objects

   ```javascript
   // Calls myCustomSanitizer(foo).
   const trustedHTML = SanitizingPolicy.createHTML(foo);
   element.innerHTML = trustedHTML;
   ```

3. **Enforce** "myPolicy" by setting a Content Security Policy header

   ```html
   Content-Security-Policy: trusted-types myPolicy myOtherPolicy
   ```
Trusted Types - default policy

The "default" policy is called as a fallback when a string is assigned to a sink.

Good way to get started and to identify risky DOM assignments.

```javascript
trustedTypes.createPolicy('default', {
    createHTML(s) {
        console.log("Please fix! Insecure string assignment: ", s);
        return s;
        // Only return if you're OK with a value.
    }
});
```

Content-Security-Policy: trusted-types myPolicy default
Trusted Types - default policy

The "default" policy will be used when navigating to javascript: URLs.

By default, javascript: URLs stop working, but custom rules can be added.

```javascript
trustedTypes.createPolicy('default', {
    createScript(s) {
        if (s === 'void(0)') { // Allowing javascript:void(0)
            return s;
        }
        return; // ... and nothing else.
    }
});
```
1. Add Trusted Types to libraries that write to the DOM
2. Verify that your code uses *only* those libs (grep, linters, compiler plugins, …)
3. Add `Content-Security-Policy-Report-Only: trusted-types *`
4. Observe violations, debug
5. `goto 2` :)
6. Enforce Trusted Types
DEMO

Can we just have innerHTML do HTML sanitization by default as a browser security standard?

Jim Manico
@manicode
Control over policies

Control policy **creation:**

- Policy name whitelist:

```plaintext
Content-Security-Policy: trusted-types myPolicy myOtherPolicy
```

- No duplicate policy names

Control policy **usage:**

- Policies are JavaScript objects
- Lock them in a module, inside a local function variable etc.
Control over policies

```javascript
(function() {
    // Seemingly unsafe policy
    const unsafePolicy = trustedTypes.createPolicy('main-component', {
        createHTML: (s) => s,
    });

    // No XSS because of the usage limitation
    unsafePolicy.createHTML(`<div>My application component<div>`)
})();
```
Benefits

- **Reduced attack surface** - The risky data flow will always be:
  
  Source → ... → Policy → Trusted Type → ... → DOM sink

- **Isolates** security-sensitive code from the rest of the applications
- **No DOM XSS** if policies are secure and access restricted

- **Compile time & runtime** security validation
- **Gradual adoption** possible
Integration examples

DOMPurify

```javascript
DOMPurify.sanitize(html, { RETURN_TRUSTED_TYPE: true })
```

React

```
$ git clone https://github.com/facebook/react
$ cd react
$ yarn
Set enableTrustedTypesIntegration = true in ReactFeatureFlags.js
$ yarn build react/index,react-dom/index --type=UMD
```

More at [github.com/w3c/webappsec-trusted-types/wiki/Integrations](https://github.com/w3c/webappsec-trusted-types/wiki/Integrations)
Project status

- Active development, looking for your feedback!
- Support in Chromium browsers
  \[\text{--enable-blink-features=TrustedDOMTypes}\]
- Discussion group - trusted-types@googlegroups.com
- W3C specification draft - w3c.github.io/webappsec-trusted-types/
- Polyfills & documentation at bit.ly/trusted-types
Try Trusted Types now!

bit.ly/trusted-types

---

**Trusted Types**

First time here? This is a repository hosting the Trusted Types specification draft and the polyfill code. You might want to check out other resources about Trusted Types:

- [Introduction for web developers](#) - API description with examples.
- [Explainer](#) - introductory explainer (what problem is the API solving?).
- [Specification draft](#) - a more comprehensive and formalized description of the Trusted Types API.
- [Origin trial for Trusted Types](#) - The API is available natively in Chrome via origin trials.

**Polyfill**

This repository contains a polyfill implementation that allows you to use the API in all web browsers. The compiled versions are stored in the `dist/` directory.

**Browsers**

The E55 / ES5 builds can be loaded directly in the browsers. There are two variants of the browser polyfill - `api_only` (light) and `full`. The `api_only` variant defines the API, so you can create policies and types. `Full` version also enables the type enforcement in the DOM, based on the CSP policy it infers from the current document (see `src/polyfill/full.js`).

```html
<!-- API only -->
<script src="https://wicg.github.io/trusted-types/dist/es5/trustedtypes.api_only.build.js"></script>
<script>
    const p = TrustedTypes.createPolicy('foo', ...)
    document.body.innerHTML = p.createElement('foo'); // works
    document.body.innerHTML = 'foo'; // but this one works too (no enforcement).
</script>
```
THANKS FOR ATTENTION!

@kkotowicz